MasPar Fortran Reference Manual

Software Version 2.0

Document Part Number 9303-0000
Revision A5, July 1992

MasPar Computer Corporation
Sunnyvale, California

MasPar Fortran Reference Manual
Part Number 9303-0000, Revision A4
Software Version 2.0, 17 July 1992

MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale, California 94086
phone: 408-736-3300
FAX: 408-736-9560

Preface

The *MasPar Fortran Reference Manual* provides a complete description of the MasPar Fortran high-level programming language. MasPar Fortran is an implementation of the standard Fortran language for MasPar Computer Corporation's family of massively parallel, data parallel computers.

MasPar Fortran is based on the widely used Fortran 77 standard with extensions from Fortran 90 and Digital Equipment Corporation's Fortran. The most important of these extensions are the Fortran 90 array statements, which allow data parallel programs to be written simply in Fortran.

Be sure to refer to the *MasPar Fortran Release Notes* for details concerning any special restrictions in this release.

Who Should Use this Book

This manual is intended for programmers and engineers who have a basic understanding of the Fortran language. This book assumes you are familiar with programming in Fortran 77, and have some knowledge of parallel programming concepts.

How this Book Is Organized

The *MasPar Fortran Reference Manual* is organized into the following parts:

- Chapter 1 describes overall program structure, the lexical elements of the language, and the statement order.
- Chapter 2 describes data types, literal constants, names, variables, and array features.
- Chapter 3 describes specification statements.
- Chapter 4 describes expressions and assignments.
- Chapter 5 describes control statements.
- Chapter 6 describes input/output statements.
- Chapter 7 describes input/output editing.
- Chapter 8 describes program units and procedures.
- Chapter 9 describes intrinsic procedures.
- Chapter 10 describes compiler directives.
- Appendix A lists the ASCII character set.
- Appendix B briefly summarizes the major differences between MasPar Fortran and Fortran 77.

Conventions Used in this Book

Notation Conventions

UPPERCASE	Fortran keywords appear in UPPERCASE letters.
`typewriter`	Examples of Fortran code appear in `typewriter font`. Enter text exactly as shown.
italics	Parameters provided by users are shown in *italics*.
[ ]	Optional elements are enclosed in square brackets.
...	Ellipses indicate that an item can be repeated one or more times.

boldface	**Boldface font** highlights new terms and concepts when they are first defined.
Δ	Represents a blank space.

Presentation Conventions

[90]	New MasPar Fortran features that are derived from the proposed Fortran 90 standard are noted in the text with **[90]**.
[MP.EXT]	New MasPar Fortran extensions that are not based on Fortran 90 are noted in the text with **[MP.EXT]**.
[NYI]	Features described in this manual that are not implemented in the current release of MasPar Fortran are noted in the text with **[NYI]**.

Related Publications

MasPar Fortran is designed to run on the MasPar® family of computers with the DEC ULTRIX operating system as a front end. For complete documentation of Fortran for ULTRIX and the runtime libraries, refer to the documentation provided by Digital Equipment Corporation.

You should also have the following MasPar manuals:

- *MasPar Fortran User Guide*, PN 9303-0100
- *MasPar System Overview*, PN 9300-0100
- *MasPar Programming Environment (MPPE) User Guide*, PN 9305-0000
- *MasPar Commands Reference Manual*, PN 9300-0300
- *MasPar Programming Language (MPL) User Guide*, PN 9302-0101

The powerful parallel array processing extensions to Fortran 77 that are implemented in this version of MasPar Fortran are based on the Fortran 90 ISO standard. Future enhancements to MasPar Fortran may implement other aspects of the Fortran 90 standard. For more information about Fortran 90 (previously referred to as Fortran 8x), MasPar Computer Corporation recommends the following books:

- *Fortran 90 Explained*, 1st edition, by Michael Metcalf and John Reid, Oxford Science Publications, Oxford University Press, 1990.
- *Programmer's Guide to Fortran 90*, by Brainerd, Goldberg, and Adams, Intertext Publications, McGraw-Hill Book Co., 1990.

Getting Help

To get online help on the MPF compiler, type

```
man mpfortran
```

at the system prompt.

To get a list of all the available MasPar man pages, type

```
man -k maspar
```

or

```
apropos maspar
```

at the system prompt.

To get more general help on how to program the MasPar parallel processing system, study the examples in $MP_PATH/examples.

Contacting MasPar

To report defects or make comments on MasPar products and documentation, you can contact MasPar via email, phone, or FAX at these locations:

In North America and the Pacific Rim:

Customer Support
MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale
California
94086
USA

phone: 1-800-526-0916
hours: 8AM - 6PM PST

FAX: 408-736-9560
email: support@maspar.com

In Europe and the United Kingdom:

European Customer Support
MasPar Computer Ltd
8 Commerce Park
Brunel Road
Theale
Reading RG7 4AB
Berkshire, England

phone: +44-734-305444
hours: 9AM - 5PM BST

FAX: +44-734-305505
email: support@mpread.co.uk

Table of Contents

List of Tables

Chapter 1

Elements of a MasPar Fortran Program

An executable Fortran program is created by compiling and linking one or more Fortran source files. A **source file** contains one or more logical groupings of statements called **program units**. In an executable program one (and only one) of the program units must be a **main program**. If other program units are included in the executable program, they must be either procedural subprograms (functions or subroutines) or block data subprograms. A **block data subprogram** supplies initial values for data objects.

Procedural subprograms are either subroutine subprograms or function subprograms. A **subroutine** is a procedure invoked using a CALL statement. A **function** is a procedure invoked in an expression. When a function is invoked, the function returns a computational value that is used to evaluate the expression in which it appears.

All program units must end with an END statement. Program execution is terminated with the execution of the END statement in the main program or with the execution of a STOP statement in a main program or a procedural subprogram. In a subroutine or function, the END statement or a RETURN statement returns control to the calling program unit.

Character Set

The MasPar Fortran character set includes:

- the letters A through Z and their lowercase equivalents
- ten digits: 0 1 2 3 4 5 6 7 8 9
- the underscore _
- the special characters = + – * / ( ) , . $ ' : ! " % & ; < > ?
- the space (blank) and the tab

The compiler does not differentiate between uppercase and lowercase letters, except in character constants. Digits in numeric constants are interpreted as decimal numbers.

The underscore (_) can be used as a significant character in names. The special characters are used as operators and delimiters.

Each character is associated with a positive integer in a **collating sequence.** MasPar Fortran uses the ASCII equivalents for its collating sequence. Appendix A shows the ASCII character sequence and the corresponding numeric values.

Statements

A **statement** is either executable or nonexecutable. An **executable statement** performs a computational action when the program is executed. A **nonexecutable statement** describes data, specifies a statement function, or classifies a program unit.

A statement can be identified with a **statement label**, which enables other statements to refer to the labeled statement. A statement label must be a number (an unsigned integer). A statement can only refer to a label associated with an executable statement or with a FORMAT statement. Nonexecutable statements can have labels, but other statements cannot refer to them.

Line Format

MasPar Fortran source programs must follow a fixed format. Source program statements are written on **lines.** The format for the lines of a source program is summarized here:

- A character occupies one column in a source program line.
- Each statement must begin in column 7 or beyond, and can extend to column 72. The line can have as many as 80 characters, but columns 73 through 80 are ignored by the compiler.

- A single statement can consist of more than one line. The first line of a statement is called the **initial line**; it must have a blank space or a zero (0) in column 6. Any other line of a statement is called a **continuation line**, which is designated by any character other than a blank space or a zero (0) in column 6. The maximum number of continuation lines allowed for one statement can be controlled by the –continuations=*n* command-line option (see the *MasPar Fortran User Guide*). The default number of continuation lines allowed for one statement is 19.
- **Statement labels** can appear only in columns 1 through 5 of the first line of a statement. Blank spaces are allowed anywhere in the label; leading zeros are ignored. The label must be a number (an unsigned integer). Each statement label can be associated with only one statement in a program unit.
- **Comment lines** begin with the letter C, an asterisk (*), or an exclamation point (!) in column 1. Additionally, text that appears anywhere on a line following an exclamation point (!) is also considered a comment. The exclamation point can be placed in any column of the line. However, when an exclamation point appears in column 6 the compiler interprets it as a continuation line indicator, *not* as a comment indicator.
- **Compiler directives** are instructions that place restrictions on certain operations or instruct the compiler to perform a special task. They appear in the source program as structured comments. Compiler directives are discussed in Chapter 10 and in the *MasPar Fortran User Guide*.
- The letter D in column 1 identifies a **debugging statement**. These statements are treated as comment lines, unless the –d_lines option is specified when the program is compiled. If the –d_lines option is specified, these lines are treated as source text and are compiled. Options are described in the *MasPar Fortran User Guide*.

Blank lines are allowed anywhere in the source program and do not affect program execution. Spaces (or blanks) are significant only when they appear in directives and character constants. Spaces can (and should) be used freely to make the source program easier to read.

Tabular Formatting

The tab character can be used to simplify entering source lines. Any characters (five characters maximum) entered before the first tab character are treated as part of the statement label. The Fortran statement can be entered immediately after the first tab character. Since no Fortran statements can begin with a digit, any digit other than zero immediately after the first tab character is treated as a continuation indicator. The continuation of the statement follows the digit. Any tab characters in the statement portion of the line are ignored, except in character strings.

Tabular formatting is based on the compiler's interpretation of the first tab character; the spacing generated by pressing the tab key in the text editor is not significant to the compiler. The maximum length of the statement part of a line, including tab characters, is 72 characters.

Statement Order

Statements in a program unit must follow a prescribed order. A **program unit definition statement** comes first and can be a PROGRAM statement (optional), a SUBROUTINE statement, a FUNCTION statement, or a BLOCK DATA statement. The program unit definition statement is followed by other specification statements. Certain kinds of specification statements can be mixed with executable statements, but specification statements generally appear before executable statements.

Table 1-1 shows statement order within a program unit. Categories separated by vertical lines can be interspersed; categories separated by horizontal lines cannot. For example, IMPLICIT statements can be mixed with PARAMETER statements, but cannot be mixed with DATA statements. FORMAT, ENTRY, and DATA statements can be mixed with executable statements.

<table>
<tr><td colspan="3">Program Unit Definition Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td>PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>PARAMETER and DATA Statements</td><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Definitions</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Table 1-1 Required Statement Order

NOTE: INCLUDE lines and compiler directives can appear on any single source line where a Fortran statement can appear, before the END statement. However, in general, both should appear only before all executable statements.

Example of Source Program Format

The following program fragment shows examples of statements, statement labels, comment lines, and continuation lines.

```
C   This is a comment line.  Any text in this line is ignored by the
C   compiler.  Several comment lines can be used together for longer
C   descriptions.

      SUBROUTINE fix_it             ! Nonexecutable statement naming
                                    ! a program unit.

      PRINT *, '  Hello, world!'    ! Executable I/O statement.  Blanks and !
                                    ! are taken literally inside a character
                                    ! string, shown by single quotes.

100   FORMAT (1X, 4F15.5)           ! FORMAT statement with a label (100).

      IF ( I .LE. K .OR.            ! Executable IF statement with
     +     J .GT. L)                ! continuation lines (shown by '+') and
     +   GO TO 150                  ! label reference (150).
```

Chapter 2

Data Types, Constants, Variables, Names, and Arrays

This chapter describes the parts of a MasPar Fortran program that relate to data specification. These are

data types MasPar Fortran classifies data into types that do not depend on any particular representation. Each data type has a name, a set of associated values, and operations to manipulate and interpret these values.

constants A constant represents a data value that cannot change during program execution. Constants can be numeric or nonnumeric.

names The user can give unique names to data entities and program units.

variables A variable is a name that represents a storage location. The data value that is stored can change at different points in the execution of a program.

arrays An array is a set of data, all of the same type and type parameters, whose individual elements are logically arranged in a rectilinear pattern.

Data Types

MasPar Fortran classifies data into types that do not depend on any particular representation. Each **data type** has a name, a set of associated values, and operations to manipulate and interpret these values.

MasPar Fortran supports these data types:

- LOGICAL
- INTEGER
- REAL (REAL*4)
- DOUBLE PRECISION (REAL*8)
- COMPLEX (COMPLEX*8)
- DOUBLE COMPLEX (COMPLEX*16)
- CHARACTER

Some data types have a set of values predetermined by the language. For example, LOGICAL data can have only the value of either true or false. For other data types, the range of values depends on the capacity of the processor. The length of a CHARACTER variable can be specified when it is declared. This limits the possible values of that variable to character strings whose length equals the specified value.

For REAL and COMPLEX data types, there are two kinds of data: single and double precision.[1] The KIND= specifier can be used in declarations and as an argument to various intrinsics (such as CMPLX and REAL) to specify the kind of the result. The value of the KIND= specifier can be specified using the KIND intrinsic function or by using the predefined compiler constants _SINGLE_ and _DOUBLE_.

Table 2-1 lists the storage requirements for each data type. The number in the second column specifies the length in bytes of each object of that type.

[1]Multiple kinds of LOGICAL, INTEGER, and CHARACTER data are also possible within the Fortran 90 standard. Only the REAL and COMPLEX multiple kinds are currently supported by MasPar Fortran.

Data Type	Storage Requirements (in bytes)
LOGICAL	4
INTEGER	4
REAL	4
DOUBLE PRECISION	8
COMPLEX	8
DOUBLE COMPLEX	16
CHARACTER**length*	*length* (1 byte per character)

Table 2-1 Data Type Storage Requirements

Constants

A **constant** represents a data value that cannot change during program execution. A constant without a name is a **literal constant**. A constant with a name is a **named constant**. A constant is assigned a name with the PARAMETER statement or the PARAMETER attribute.

Constants are either numeric or nonnumeric. Numeric constants (integer, real, double precision, complex, and double-complex types) are used for computation; they are positive, negative, or zero. Nonnumeric constants include character and logical types.

Binary, hexadecimal, and octal constants are allowed in DATA statements. An unsigned binary, octal, or hexadecimal literal constant can correspond to an integer scalar or array entity.

Integer Constants

An **integer constant** represents the value of a mathematical whole decimal number that has a positive, negative, or zero value. If no leading sign is used, the value is interpreted as positive. Leading zeros are ignored.

An integer constant is composed of an optional leading plus (+) or minus (–) sign, followed by a string of any number of digits (0 through 9) that do not contain a decimal point or commas. Leading zeros and interspersed spaces do not affect the value of an integer constant. The value of an integer constant must be within the range –2147483648 to +2147483647.

The following list shows examples of valid integer constants:

```
182
0
-38643
+180421
13 000 000
00000756
```

Binary, octal, or hexadecimal integer constants are allowed only in DATA statements. The syntax for a binary constant is

B'*digit*[*digit*]...'

or

B"*digit*[*digit*]..."

digit can be either 0 or 1.

The syntax for an octal constant is

O'*digit*[*digit*]...'

or

O"*digit*[*digit*]..."

digit can be any value 0 through 7.

The syntax for a hexadecimal constant is

Z'*hex-digit*[*hex-digit*]...'

or

Z"*hex-digit*[*hex-digit*]..."

hex-digit can be any value 0 through 9, or any of these characters: A, B, C, D, E, or F.

Real Constants (REAL*4)

A **real constant** approximates the value of a mathematical real number. The value of real constants can be positive, negative, or zero. If no sign is used, a positive value is assumed. Leading zeros are ignored.

The basic form of a real constant consists of an optional plus (+) or minus (–) sign, followed by a string of decimal digits (0 through 9) that *can* contain a decimal point, but *cannot* contain commas. The decimal point can be anywhere in the string.

A real constant can include an exponent, which is indicated by adding the exponent letter E to the basic form of the real constant. If a decimal exponent is used, the string of decimal digits does not require a decimal point. The exponent letter E is followed by an integer constant, which can be positive or negative. The integer constant represents the power of 10 by which the preceding constant is to be multiplied. The sign is optional before a positive exponent, but must be included if the exponent is negative. If the letter E appears in a real constant, it must be followed by an integer exponent, which can be zero. The exponent letter E *cannot* be followed by a blank space.

The following list shows examples of valid real constants:

```
101.326
18.
+.00005
15.E04
4.3E-3
2E10
-.02E-8
```

Double-Precision Constants (REAL*8)

A **double-precision constant** represents a numeric value that can contain more significant digits than real constants. The range of values depends on the capacity of the processor. Double-precision constants have the same basic form as real constants, except that the exponent letter D must be included, followed by an optionally signed integer exponent. The exponent must be included; zero is a valid exponent.

The following list shows examples of valid double-precision constants:

```
5473209821D+4
-38.75D-12
1D0
+5.473209821D8
```

Complex Constants (COMPLEX*8)

A **complex constant** consists of a pair of values separated by a comma and enclosed in parentheses. The first value is defined as the real part, the second value as the imaginary part. The values can be integer or real constants.

The following list shows examples of complex constants:

```
(3, 4.E12)
(2.1, 0.03)
(-9.3E+2 ,+.00062E-2)
```

Double-Complex Constants (COMPLEX*16)

A double-complex constant consists of a pair of constants that represent a complex number. At least one constant must be double precision; the other must be an integer, real, or double precision. The two constants are separated by a comma and enclosed in parentheses. The first constant represents the real part of the complex number, the second the imaginary part.

The following list shows examples of double-complex constants:

```
(1.7039D0, -1.7039D0)
(+12739D3, 0.D0)
```

Character Constants

A **character constant** is a sequence of characters. Any character that the processor can represent can be included in the character string.

Apostrophes or quotation marks are used as delimiters; the same delimiter must start and end the sequence. Delimiters are not stored as part of the constant, but blanks and tabs are.

When apostrophes are used as the string delimiter, a literal apostrophe within a character constant is represented by two consecutive apostrophes (with no space or other character between them). The same is true of quotation marks.

The length of the character constant is the number of characters between the apostrophes (or quotation marks), except that two consecutive apostrophes (or quotation marks) represent a single apostrophe (or quotation mark). The length of a character constant must be in the range 1 to 2000. If not declared, the default length is 1.

The following list shows examples of valid character constants:

```
'HELP!'
'Today''s date is:'
"Today's date is:"
"Enter your name:"
```

Logical Constants

A **logical constant** represents the logical value of true or false. .TRUE. and .FALSE. are the only valid logical constants. The delimiting periods must always be included.

Names

A programmer identifies data entities and program units with **names.**[2] The maximum length of a name is 31 characters. It can include any combination of letters, digits, and the underscore (_); however, the first character must be a letter. The dollar sign ($) can be embedded within the name. Names cannot begin with a digit. Names beginning with the underscore (_) or the dollar sign ($) are reserved and should not be used. Spaces and tabs within names are ignored.

---

[2]Called symbolic names in Fortran 77.

The following list shows examples of valid names:

```
B12
fix_it
Whatnot
Pivot Point
```

MasPar Fortran is not case sensitive; lowercase letters are always converted to uppercase. In other words, the MasPar Fortran compiler will not distinguish between a routine named "Whatnot" and a routine named "WHATNOT".

In general, a name cannot be used to identify more than one entity in the same program unit. The one exception is in the case of a named common block. The name of the common block can be the same as the name of another local entity in the program unit. If an executable program has more than one program unit, names for the main program and subprograms must be unique.

MasPar Fortran uses **keywords** (such as IF, INTEGER, and PARAMETER) in statements. These are not reserved words. That is, they can also be used as names. Their meaning is derived from their context.

The name of a MasPar Fortran data entity can imply its data type, which defines the entity's set of values, storage allocation, and its set of valid operations.

The most common data types in MasPar Fortran are integer and real. Names beginning with the letters I, J, K, L, M, and N imply integer type; names beginning with any other letter imply real type. Names beginning with the dollar sign ($) or the underscore (_) must be explicitly typed if they name typed objects.

For example, entities with the following names would by default be treated as integers:

```
I
Number
KFOO
```

Entities with the following names would by default be treated as reals:

```
X
rvalue
AVERAGE
```

A programmer can, however, override the default data type implied in a name by using either an IMPLICIT statement or a type declaration statement. Names beginning with a dollar sign ($) or an underscore (_) cannot appear in the IMPLICIT statement.

Variables

A **variable** is a name that represents a storage location. The amount of storage allocated for a variable depends on its type (see Table 2-1). A variable with a valid value is considered **defined**. A variable must be defined before it can be referenced in a context that requires its value. A variable that ceases to have a predictable value becomes **undefined**.

Initial values are given to variables using type declaration statements or DATA statements. Ways of changing the values of variables during program execution include using assignment statements or READ statements. If the value assigned to the variable does not match its type, the value is converted to the type of the variable.

Two or more variables are **storage associated** when their values are stored in the same location. They are **partially associated** when part, not all, of one variable is associated with part, or all, of another variable.

A variable is either a **scalar** or an **array**. Scalar variables represent one unit of value. Array variables represent sets of values, all of the same type. Depending on how a variable is used (in a Fortran 90 or Fortran 77 context), the elements of the array will be stored either on the front end of the MasPar system or the DPU (where the data-parallel processing is done). Arrays that are stored on the front end are stored contiguously. On the DPU, arrays are stored in a more distributed fashion. The MasPar Fortran compiler determines where the array should be stored by the context in which it is used. For more information on where the compiler stores variables, see the *MasPar Fortran User Guide*.

Arrays provide a wide range of possibilities for representing and manipulating data. Arrays are described in the next sections.

Arrays

An **array** is a set of data, all of the same type and type parameters, whose individual elements are logically arranged in a rectilinear pattern. Each member of the array is an **array element**. All array elements must be of the same data type. Arrays can be divided into subsets, called **array sections**, which are also arrays. Each MasPar Fortran array has

- a data type,
- a fixed rank,
- a size,
- and a shape.

Data Type: The data type of the array is the same as the data type of its elements. If a value assigned to an individual array element is not of the declared type of the array, its value is converted to that of the array. The data type of an array is either implied by the first letter of its name or specified in a type declaration statement.

Rank: An array can have from one to seven dimensions. The number of dimensions is the **rank** of the array. The **bounds** of a dimension are defined by an array specification (described in the next section). A bound must resolve to a positive or negative integer, or to zero.

Size: The number of elements in any one dimension is the **extent** of that dimension. The value of the extent is calculated as follows:

max (*upper_bound* – *lower_bound* + 1,0)

The value of an extent is zero if the value of the lower bound is greater than the value of the upper bound. The **size** of an array is the total number of elements, which is the product of the array's extents. An array has a size of zero if the value of any of its extents is zero.

Shape: An array's rank and extents define its **shape**. The array's shape can be represented by a list of the array's extents, which itself is an array. For example, if a rank three array named X5 has extents of 10, 15, and 20, the shape of X5 could be expressed as [10, 15, 20], which itself is a rank one array. Two arrays with the same shape are said to be **conformable**. A scalar is conformable to an array of any shape.

The name and rank of an array are constant; they must be specified when the array is declared. The extent of each dimension can be constant, but does not need to be. The extents can be specified during program execution if the array is a dummy argument array or an automatic array.

A **dummy argument array** is used in the definition of a subprogram. Adjustable, assumed-size, and assumed-shape arrays are all kinds of dummy argument arrays. An **adjustable array** can reference scalar dummy arguments or common variables as part of its bound expressions. An assumed-size or assumed-shape array derives its extent information from the actual argument in the argument list when the subprogram is called.

An **automatic array [90]** is a local array declared in the specification part of a subprogram. The size of the array is specified with a variable expression and defined when the subprogram is called.

Array Specifications

An **array specification** declares the rank, or the rank and shape, of an array. The array specification is appended either to the name of the array or to the DIMENSION attribute when the array is declared. Array specifications are enclosed in parentheses.

Depending on how an array is declared, the array falls into one of the following categories:

- explicit-shape array
- assumed-shape array
- assumed-size array

Explicit-Shape Arrays

An **explicit-shape array** is declared with an explicit value for the upper bound of each dimension. The lower bound can be specified, but does not need to be. If the lower bound is not specified, the default value is 1. The declaration explicitly specifies the rank of the array and the extent of each dimension; thus, the size and shape of the array are also explicit. When both are specified, the lower and upper bounds are separated by a colon. The bounds for each dimension are separated by commas.

Following are some examples of specifications that declare explicit-shape arrays:

```
A (10, 2:10, 0:13)

SUBROUTINE TEST (I, J, X)
    REAL W (I:J, 15)          ! automatic array
    REAL X (-3:I+J)           ! adjustable array
    REAL U (-3:21)
```

The bounds can depend on variable expressions if the array is either an **automatic array** (W in the example above), or a **dummy argument array** (X in the example above). The expression must be either a specification expression or an initialization expression (described in Chapter 4). The bounds are determined on entry to the procedure when the expression is evaluated.

Assumed-Shape Arrays

[90]: The assumed-shape array is a Fortran 90 feature.

An **assumed-shape array** is a dummy argument array. It assumes the shape of the associated actual argument array. The array specification contains a colon for each dimension. The upper bound is never specified; the lower bound can be.

Following are some examples of specifications that declare assumed-shape arrays:

```
SUBROUTINE TEST (A, B, C, M)
   REAL A(:), B(0:), C(M:,:)
```

The extent of each dimension of an assumed-shape array is the extent of the corresponding dimension of the actual argument array. The value of the upper bound is calculated as follows:

extent + lower_bound – 1

The lower bound can be specified with a specification expression or an initialization expression (described in Chapter 4). If the lower bound is not specified, the default value is 1.

Assumed-Size Arrays

An **assumed-size array** is a dummy argument array. It assumes only the size of the associated actual argument. An assumed-size array does not have a shape or an extent in its last dimension. The rank and the extents can differ for the dummy array and the actual array. An array specification for an assumed-size array uses an asterisk (*) to represent the value of the upper bound of the last dimension. Bounds for other dimensions must be specified explicitly. Specification expressions or initialization expressions, described in Chapter 4, can be used for all but the last dimension. If the lower bound of the last dimension is omitted, the default value is 1.

The following example shows specifications that declare assumed-size arrays:

```
SUBROUTINE TEST (I, Y, Z)
     REAL Y (10, I, I:*)
     REAL Z (*)
```

Assumed-size arrays cannot be used in array-valued expressions because the upper bound of the last dimension is unknown. That is, in the above example `Z=0` would be invalid, but `Z(1)=0` is allowed.

Assumed-size arrays cannot be used in Fortran 90 array expressions. In most cases, assumed-size arrays can be replaced by assumed-shape arrays to take advantage of the Fortran 90 array features.

Subscripts

A **subscript** references an array element or an array section. For multidimensional arrays, subscripts can be combined into a subscript list, which has a form similar to the array specification. To reference an array element, the name of the array is followed by one or more subscripts. The list of subscripts is enclosed in parentheses. The number of subscripts must be the same as the number of dimensions declared for the array. The subscript must be an integer, or a variable expression that results in an integer value; its value must be within the declared bounds for the corresponding dimension.

For example, the following references would be valid for array elements in an array declared as REAL A5 (10, 10):

```
A5 (1, 2)
A5 (I, J)
A5 (I+J, L-M)
```

Subscript Triplet

[90]: Subscript triplet notation is a Fortran 90 feature.

A **subscript triplet** designates a range of subscript values and the increment, or **stride**, between each value. The stride can be a positive or negative integer value; it cannot be zero. The subscript values and stride are separated by colons.

If the stride is positive, the **subscript range** starts with the first subscript and is incremented by the value of the stride, until the largest value less than or equal to the second subscript is attained. For example, if an array has been declared as A4(16), the array section specified as A4(2:16:7) consists of three elements: A4(2), A4(9), and A4(16). The range is empty if the value of the first subscript is greater than the value of the second subscript.

If the stride is negative, the subscript range starts with the value of the first subscript and is decremented by the absolute value of the stride, until the smallest value greater than or equal to the second subscript is attained. For example, if an array has been declared as A4(16), the array section specified as A4(15:1:-4) consists of four elements: A4(15), A4(11), A4(7), and A4(3). The range is empty if the value of the second element is greater than the value of the first element.

When specifying array sections, values of the subscript triplet can be omitted. If the value of the first element of the triplet is omitted, the declared value of the lower bound of the dimension is assumed. If the value of the second element of the triplet is omitted, the declared value of the upper bound of the dimension is assumed. If the third element of the triplet is omitted, the stride defaults to a value of 1.

Vector-Valued Subscripts

A **vector-valued subscript** is a vector, or a rank one array-valued expression, that is used to index a dimension of an array. A vector-valued subscript can define a many-to-one section, where one or more elements of the vector expression have the same value. Vector-valued subscripts can be used on either the right or left of an assignment. However, a many-to-one subscript cannot appear on the left.

When an array is subscripted with a vector, the result shape has an extent due to the vector-valued subscript, which is the size of the vector. In the example below, the expression is conformant because the vector *v* is the same size as the lefthand side array *r*.

```
integer a(10)
integer v(5), r(5)
data a /21, 22, 23, 24, 25, 26, 27, 28, 29, 30/
data v /2, 4, 10, 8, 10/

  r = a(v)
end
```

If this code is compiled and executed *r* will contain [22, 24, 30, 28, 30].

Vector-valued subscripts can also be used together or with other sections to index higher rank arrays. For example:

```
integer, dimension( 5 ) :: v
integer, dimension( 5, 5 ) :: a, rslt

data v /5, 4, 3, 2, 1/

call init( a )
  rslt = a(v, 5:1:-1)
end
```

The expression *a(v, 5:1:-1)* is a rank 2 array with the shape [5,5].

Here *v* is initialized to [5, 4, 3, 2, 1]. If the array *a* is initialized to:

```
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

then *rslt* will contain

```
25  24  23  22  21
20  19  18  17  16
15  14  13  12  11
10   9   8   7   6
 5   4   3   2   1
```

In Fortran 77 the vector-valued subscript assignment above would be written as follows:

```
do j = 1, 5
   do i = 1, 5
     rslt(i, j) = a(v(i), 6-j)
   end do
end do
```

In contrast to the Fortran 77 code, the vector-valued subscript operation takes place in parallel. Vector-valued subscript operations use the global router of the DPU.

Array Constructors

[90]: The array constructor is a Fortran 90 feature.

An **array constructor** is a sequence of scalar values, interpreted as a one-dimensional array. The array element values are those specified in the sequence, as if an assignment were made for each element. The sequence of values can be specified as individual scalar values, ranges of values, or a one-dimensional array expression. Values in the sequence are separated by commas. An array constructor must be enclosed in parentheses, with a slash or bracket just inside each parenthesis.

The syntax for an array constructor is

(/ *array-constr-value list* /)

where

array-constr-value Is an *expression*, or is an *array-constr-implied-do*
Each *array-constr-value* expression in the sequence must have the same type and type parameters. The type of the array constructor is the same as the type of this *array-constr-value*.

array-constr-implied-do
Is (*array-constr-value-list, array-constr-implied-do*)
When the *array-constr-implied-do-control* is present, the *array-constr-value-list* must be a constant or a *do-variable*. The loop initialization and control are the same as for a DO construct; however, nested *array-constr-implied-do*(s) are not supported.

array-constr-implied-do-control
Is the following:
array-constr-do-variable = scalar-integer-expr, scalar-integer-expr[, *scalar-integer-expr*]

array-constr-do-variable
Is a *scalar-integer-variable*
This must be a named variable.

If an *array-constr-value* is a scalar expression, its value specifies an element of the array constructor. If its value is an array expression, the values of the elements of the expression (in array element order) specify the corresponding sequence of elements of the array constructor. If the *array-constr-value* is an *array-constr-implied-do*, it is expanded to form an *array-constr-value* sequence under the control of the *array-constr-do-variable*, just like in the DO construct. However, nested *array-constr-implied-do*(s) are not supported.

Following are examples of array declarations with different forms of array constructors:

```
REAL A4 (4)                              ! Each value of A4 is
  A4 = (/ 32.1, 83.5, 90.6, 22.1 /)      ! specified.

INTEGER I4 (6)                           ! Uses the form of an implied
  I4 = (/ (I, I=32, 42, 2) /)            ! do to specify a range of
                                         ! values and a stride.  The
                                         ! values of I4 are the even
                                         ! numbers 32 through 42.
```

Array constructors for arrays with more than one dimension can be created by using the intrinsic function RESHAPE. For example, the statement

```
Y = RESHAPE (SOURCE = (/ I, I=1,6 /), SHAPE = (/ 3,2 /))
```

creates the following value for array Y:

```
Y(1,1) = 1
Y(2,1) = 2
Y(3,1) = 3
Y(1,2) = 4
Y(2,2) = 5
Y(3,2) = 6
```

The data type of a constructor is the data type of the first item in the list; all elements of the constructor must have the same data type as the first element.

For compatibility with older versions of the array constructor, the following (now obsolete) syntax forms are also supported:

- Use of square brackets ([ ]) instead of the slash/parentheses sequence ((/ /)), as in

```
Y = RESHAPE (SOURCE =  [1:6], SHAPE = [3,2])
```

- Use of the colon-selector syntax instead of the implied-do syntax, as in

```
(/ J:K:L /)
```

  which would be written as follows using the implied-do syntax:

```
(/ (I, I=J,K,L) /)
```

- Use of the repeat count, as follows:

```
INTEGER I4 (6)
  I4 = (/ 6 (/ 0 /) /)
```

  which has the same meaning as:

```
(/ (0, I=1, 6) /)
```

Chapter 3

Specification Statements

This chapter describes the MasPar Fortran nonexecutable statements for data specification. These are

- type declaration statements
- type specifiers
- PARAMETER, INTENT, DIMENSION, and SAVE attributes
- PARAMETER statement
- DATA statement
- SAVE statement
- DIMENSION statement
- EQUIVALENCE statement
- COMMON statement
- IMPLICIT statement
- NAMELIST statement

This chapter also describes the INCLUDE line.

Type Declaration Statements

The syntax of a type declaration statement is

type_spec[[,*attribute_spec*] . . . ::] *object_decl*[,*object_decl*] . . .

where:

type_spec	Is one of the type specifiers supported by MasPar Fortran, followed by an optional KIND specifier, in parentheses. The KIND specifier is an integer initialization expression.
attribute_spec	Is one or more attributes, separated by commas. Certain combinations are not permitted.
object_decl	Is the data object being declared, which can include array specifications and initial values. If more than one name is specified, the names are separated by commas.

A type declaration statement specifies the type and other attributes of named data objects. These attributes determine the characteristics and uses of the data. A particular attribute for a data object can be specified only once within a program unit.

Declared non-dummy objects whose size parameters are variable expressions are referred to as **automatic data objects**.

These examples show type declaration statements:

```
REAL A
LOGICAL, DIMENSION (5,5) :: MASK1, MASK2
CHARACTER (LEN = 10) Name
CHARACTER *10, B, C *20
INTEGER, DIMENSION (X,Y) :: DYNAMIC_ARRAY      !automatic
```

Type Specifiers

The **type specifier** specifies the type of the associated data objects. This can confirm or override the first letter rule for data objects in MasPar Fortran (integer type for names beginning with letters I through N; real type for names beginning with any other letter). MasPar Fortran supports the following type specifiers:

INTEGER	Specifies that the associated objects are of type integer. If the KIND specifier is present, it specifies the integer representation method. If the KIND specifier is absent, the kind type parameter is KIND(0) and the objects declared are of type default integer. Therefore, an integer declared with the type specifier INTEGER(KIND(0)) is of the same kind as one declared with the type specifier INTEGER. The default is four bytes of storage for each associated object. (Only one KIND of INTEGER is currently supported.)

REAL	Specifies that the associated objects are of type real. If the KIND specifier is present, it specifies the real approximation method. If the KIND specifier is absent, the kind type parameter is KIND(0.0) and the objects declared are of type default real. An object declared with a type specifier REAL(KIND(0.0)) is of the same kind as one declared with the type specifier REAL. The default is four bytes of storage for each associated object.
DOUBLE PRECISION	Specifies that the associated objects are of type double precision. Parameter values for type double precision are considered different from parameter values for type real. Therefore, parameter values must agree when double precision values are defined. The kind parameter value is KIND(0.0D0). An object declared with the type specifier REAL(KIND(0.0D0)) is of the same kind as one declared with the type specifier DOUBLE PRECISION. The default is eight bytes of storage for each associated object.
COMPLEX	Specifies that the associated objects are of type complex. If the KIND specifier is present, it specifies the real approximation mehtod of the two real values making up the real and imaginary parts of the complex value. If the KIND specifier is absent, the kind type parameter is KIND(0.0) and the objects declared are of type default complex. An object declared with a type specifier COMPLEX(KIND(0.0)) is of the same kind as one declared with the type specifier COMPLEX. The default is eight bytes of storage for each associated object.
DOUBLE COMPLEX	Specifies that the associated objects are of type double complex. Parameter values for type double complex are considered different from parameter values for type complex. Therefore, parameter values must agree when double complex values are defined. The kind parameter value is KIND(0.0D0). An object declared with the type specifier COMPLEX(KIND(0.0D0)) is of the same kind as one declared with the type specifier DOUBLE COMPLEX. The default is sixteen bytes of storage for each associated object.
LOGICAL	Specifies that the associated objects are of type logical. If the KIND specifier is present, it specifies the representation method. If the KIND specifier is absent, the kind type parameter is KIND(.FALSE.) and the objects declared are of type default logical. The default is four bytes of storage for each associated object. (Only one KIND of LOGICAL is currently supported.)
CHARACTER	Specifies that the associated objects in this statement are of type character. Each individual character requires one byte of storage. The length of character objects can be declared in three ways: with an asterisk (*) followed by the number of characters, with the length specified in bytes, or with a length expression. These formats are as follows:

CHARACTER*(*)

or

CHARACTER**length*

or

CHARACTER**(length_expression)*

where:

length	Is the length in bytes (also the number of characters) of each associated object. If no length is specified in the declaration, the default length is one character.
length_expression	Is a scalar-valued initialization expression (described in Chapter 4). The expression must be of type integer.

When the length is specified for a statement function or a statement function dummy argument whose value is of type character, the length must be specified with a constant expression. Otherwise, length can be specified either with a constant expression or with a variable expression. If the parameter evaluates to a negative number, the length of the declared character object is zero.

The parameter value can be declared with an asterisk (*) in the following instances:

- *Declaring a dummy argument of a procedure.* When the procedure is invoked, the dummy character argument assumes the length of the associated actual argument.
- *Declaring a named constant.* The length is that of the constant.
- *Specifying an external function.* The function is declared in the program unit that references it with a value other than an asterisk (*). When the function is invoked, the length of the result variable is assumed from the declared length in the local program unit.

If the KIND specifier is present, it specifies the representation method. If the KIND specifier is absent, the kind type parameter is KIND('A') and the objects declared are of type default CHARACTER. (Only one KIND of CHARACTER is currently supported.)

Attributes

[90]: The attribute specifier is a Fortran 90 feature.

In addition to using specification statements (such as the PARAMETER statement or the DATA statement) to specify attributes of data objects, MasPar Fortran allows you to specify attributes of data objects in the type declaration statement using **attributes.** MasPar Fortran supports five attributes:

- PARAMETER
- INTENT

- DIMENSION
- OPTIONAL
- SAVE

These attributes are described in detail in the following sections.

The PARAMETER Attribute

The **PARAMETER attribute** specifies that any object named in the declaration statement is a constant. The constant is defined with the value of the initialization expression to the right of the equal sign (=). The initialization expression must be scalar-valued and can be of any data type. Restrictions that apply to initialization expressions are described in Chapter 4.

Following are examples of type declaration statements using the PARAMETER attribute:

```
REAL, PARAMETER :: C = 2.9979251, Y = (4.1 / 3.0)
INTEGER, DIMENSION(3), PARAMETER :: NUMBERS = (/ 33, 32, 31 /)
```

Defining a named constant in this way is the same as defining a named constant using the PARAMETER statement (described later in this chapter). **NOTE:** Named constants cannot be included in a format specification.

The INTENT Attribute

The **INTENT attribute** specifies the intended use of a dummy argument. It can appear either in the declaration section of a subprogram or in an Interface block. The INTENT attribute has three forms:

INTENT (IN)	Specifies that the dummy argument must *not* be redefined within the procedure. The associated actual argument can be either a constant or a variable.
INTENT (OUT)	Specifies that the dummy argument *must* be defined within the procedure. The associated actual argument must be a definable variable, which becomes undefined on entry to the procedure. In this instance, the argument is intended only to pass data out of the procedure.
INTENT (INOUT)	Specifies that the dummy argument is intended both to receive data from the calling program unit and to return data to the calling program unit. Therefore, the associated actual argument must be a definable variable. INTENT (INOUT) is the default attribute when not explicitly specified.

The INTENT attribute cannot be included in declarations using the PARAMETER attribute or the SAVE attribute.

This example shows type declaration statements using the INTENT attribute:

```
SUBROUTINE TEST (I, J)
  INTEGER, INTENT (IN) :: I
  INTEGER, INTENT (OUT), DIMENSION (I) :: J
```

The DIMENSION Attribute

The **DIMENSION attribute** specifies that the objects named in this declaration statement are arrays. The rank and shape of the array are specified with an array specification. Array specifications can contain specification expressions and initialization expressions (described in Chapter 4).

The DIMENSION attribute does not need to be specified explicitly in the declaration of an array. If it is included, the array specification immediately follows the attribute and applies to all the associated names in the declaration. If the DIMENSION attribute is not present, the array specification follows the name of the array or the array must appear in a DIMENSION statement (described later in this chapter). If the DIMENSION attribute is specified with an array specification for a list of objects, an array in the list can be declared with its own rank, or rank and shape. The array specification associated with the name takes precedence over the specification associated with the DIMENSION attribute.

These examples show type declaration statements using the DIMENSION attribute:

```
REAL, DIMENSION (10, 10) :: A, B, C (10, 15)
REAL, DIMENSION (:) :: E
```

NOTE: In previous releases of MasPar Fortran, the DIMENSION attribute was called the ARRAY attribute. The MasPar Fortran compiler accepts both the ARRAY and DIMENSION attributes; however, DIMENSION is the preferred term, in compliance with the Fortran 90 standard.

The OPTIONAL Attribute

The **OPTIONAL attribute** specifies that a dummy argument need not be associated with an actual argument in a reference to the procedure. The PRESENT intrinsic function can be used to determine whether an actual argument has been associated with a dummy argument that has the OPTIONAL attribute.

The OPTIONAL attribute *must* be specified in both the scoping unit of a subprogram and the Interface block, and can be specified only for dummy arguments.

These examples show type declaration statements using the OPTIONAL attribute:

```
REAL, OPTIONAL, DIMENSION(N) :: UPPER,LOWER
INTEGER, OPTIONAL :: A, B, C
```

The SAVE Attribute

The **SAVE attribute** specifies that objects named in the declaration are saved objects. A **saved object** is a variable that retains its value after the execution of a RETURN statement or an END statement in the program unit containing the declaration. The SAVE attribute can be specified in a declaration statement in a main program, but the attribute has no effect.

The SAVE attribute must *not* be specified for

- COMMON blocks (although they can appear in a SAVE statement)
- dummy arguments
- procedures
- function results
- automatic data objects

Variables with initial values specified when they are declared acquire the SAVE attribute automatically.

This example shows a type declaration statement using the SAVE attribute:

```
SUBROUTINE TEST ()
  REAL, SAVE :: X, Y
```

The PARAMETER Statement

Defining a named constant using the PARAMETER statement is the same as defining a named constant when it is declared with a type declaration statement that specifies the PARAMETER attribute (see preceding section).

The syntax of the PARAMETER statement is

PARAMETER (*c_name*=*i_expression* [,*c_name*=*i_expression*] . . .)

where:

c_name	Is the name of a constant whose type must have either been specified by a type declaration statement earlier in the same program unit or determined by the implicit typing rules in effect for the program unit.
i_expression	Is a scalar-valued initialization expression of any data type, which is subject to the restrictions described in Chapter 4.

The PARAMETER statement defines one or more named constants. The value of the initialization expression on the right of the equal sign (=) is assigned to the named constant on the left of the equal sign.

A named constant can appear to the right of the equal sign if it has been defined previously in the same PARAMETER statement, or defined in a prior PARAMETER statement or type declaration statement.

Following are examples of PARAMETER statements:

```
PARAMETER (MODULUS = MOD(28,3), SENATORS = 100)
PARAMETER (i = 3, j = i**3)
```

The DATA Statement

The **DATA statement** defines initial values for variables before program execution. While DATA statements can be interspersed with executable statements, the DATA statement normally appears in the specification part of the program, where data initialization logically belongs.

A variable can be initialized only once in an executable program. If an explicitly typed variable that appears in a DATA statement appears in a later type declaration, that subsequent declaration *must* confirm the implicit typing. All named variables and parts of named variables initialized using the DATA statement have the SAVE attribute, which can be reaffirmed by using a SAVE statement or a type declaration statement that specifies the SAVE attribute.

The DATA statement has two forms: a list-oriented form and an object-oriented form. Each form has its own syntax.

The List-Oriented DATA Statement

The syntax of the list-oriented DATA statement is

DATA *object_list* / *value_list* /[[,] *object_list* / *value_list* /] . . .

where:

object_list	Is a list of scalar names, array names, array element designators, or character substring designators, separated by commas. The list can also include implied DO loops.
value_list	Is a list of constants that will be assigned to the objects in the *object_list*. Each value in *value_list* can include a repeat count. The number of constants in *value_list* must equal the number of objects in *object_list*.

Values are assigned to objects in *object_list* in the order they appear in *value_list*, from left to right, as shown in this example:

```
DATA X, Y, Z /1.1, 2.2, 3.3/, I, J, K /1, 2, 3/
```

Here, X is assigned 1.1, Y is assigned 2.2, and Z is assigned 3.3.

The objects in *object_list* cannot be

- dummy arguments
- in a COMMON block, except in a block data program unit
- a zero-sized object
- an automatic object
- a function

A positive integer **repeat count** can specify the number of times a constant is repeated. The following example shows the form of the repeat count in the value list.

```
DATA I, J, K/3*0/     ! I, J, and K are initialized to zero.
```

Array element values can be initialized in three ways—by name, by element, or by an implied DO loop—as shown in the following examples. For these examples, assume the array has been declared as A(10,10).

- Here, the whole array is initialized by name:

```
DATA A/100*1.0/     ! All elements of A are
                    ! initialized to 1.0.
```

- Here, individual elements of the array are initialized:

```
DATA A(1,1), A(10,1), A(3,3)/2*2.5, 2.0/
```

  If a repeat count does not appear, a repeat count of one is assumed.

- In this example the array is initialized with an implied DO loop, which is interpreted in the same way as a DO construct.

```
DATA(( A(I,J), I = 1,5,2), J = 1,5)/15*1.0/  ! 15 elements, each
                                             ! initialized to 1.0.
```

  The syntax of the implied DO loop is

  (*dlist*,*int* = *i_expr*[,*i_expr*[,*i_expr*]])

  where:

dlist	Is one or more array element names, character substring names, or implied DO loops, separated by commas.
int	Is a named scalar integer variable.
int_expr	Is one or more scalar initialization expressions, which must be of type integer. The expressions can only contain constants and variables that appear in arrays containing implied DO loops.

  Each element in the sequence defined by the DO loop is assigned the corresponding value in the *value_list*, following the rules of assignment (see Chapter 4). The number of values in each *object_list* must equal the number of values in each *value_list*.

If the data type of a named object in the object list is either character or logical, the type of the corresponding constant must match. Character values are space-filled or truncated on the right as necessary to match the size of the corresponding object.

If the data type of a numeric value does not match the data type of the corresponding named object in the object list, the value is assigned according to the rules for mixed-mode expressions.

The Object-Oriented DATA Statement

The syntax of the object-oriented DATA statement is

[DATA] (*data-value-assignment* [,*data-value-assignment* . . .])

where *data-value-assignment* is

variable = *constant*

or

implied-DO-loop = *constant-array-constructor* **[90]**

The initial value of the variable is defined with the value of the constant expression to the right of the equal sign. If more than one variable is defined in the same DATA statement, the assignments are separated by commas. The DATA statement is quite flexible because the variable can be an array section or character substring. The subscript or substring expression can be a scalar initialization expression of type integer. (Initialization expressions are described in Chapter 4.)

Following are examples of the objected-oriented DATA statement:

```
REAL A, B(5)
DATA (A = 1.0, B = (/ 1.0, 7.9, 9.8, 10.2, 5.5 /) )

REAL C(300)
DATA (C(101:103) = (/ 1.1, 7.9, 9.8 /) )

CHARACTER*70 STRING
INTEGER, PARAMETER :: K = 24, L = 29
DATA (STRING(K+4:L-1) = 'X')
```

The object-oriented DATA statement can also use an implied DO loop to define an array section. An array constructor provides values for each array element. The syntax of the implied DO loop is

(*dlist*, *int* = *i_expr* [,*i_expr*[, *i_expr*]])

where:

dlist — Is one or more array element names, character substring names, or implied DO loops, separated by commas.

int Is a scalar integer variable.

i_expr Is one or more scalar initialization expressions, which must be of type integer. The expressions can only contain constants and variables that appear in arrays containing implied DO loops.

Each element in the sequence defined by the DO loop is assigned the corresponding value in the array constructor, following the rules of assignment. The number of elements must equal the number of values in the array constructor.

These examples show this second form of the object-oriented DATA statement:

```
INTEGER I
REAL A(5,5)
DATA ((A(I,1), I = 1,5,2) = (3 (1.1)))
DATA ((A(1,I), I = 1,5,2) = (3 (1.1)))

INTEGER J,K
REAL, DIMENSION (10, 10) :: B
DATA (((B (J, K), K = 1, 10), J = 1, 10) /100*42.0/
```

The SAVE Statement

The syntax of the SAVE statement is

SAVE [[::] *saved–object* [,*saved–object*] . . .]

where:

saved-object Is the name of a data object or the name of a COMMON block enclosed in slashes (*/common_block_name/*). The object must not be a dummy argument name, a procedure name, a function result name, or the name of an object in a COMMON block. All objects included within the COMMON block have the SAVE attribute.

The SAVE statement specifies that the data objects listed retain their value after the execution of a RETURN statement or an END statement in the procedure in which the specification appears. If the SAVE statement is specified without any objects, all allowed data objects in the procedure acquire the SAVE attribute. Therefore, no other occurrence of the SAVE attribute or another SAVE statement can appear in a program unit or interface block with a SAVE statement that has no explicitly specified objects.

If the name of a COMMON block appears in a SAVE statement in any one program unit of an executable program, the COMMON block name must be specified in a SAVE statement in every program unit of the executable program in which the COMMON block appears. The SAVE statement has no effect when specified in a main program.

Following is an example of a SAVE statement:

```
SAVE A, B, C, /BLOCKD/, E
```

The DIMENSION Statement

The syntax of the DIMENSION statement is

DIMENSION *array_name* (*array_spec*) [,*array_name* (*array_spec*)] . . .

where:

array_name — Is the name of the array-valued object being declared. The data type of the array's elements is inferred from either the default implicit typing rules, the user-defined implicit typing rules, or an array name that is declared with a type declaration statement.

array_spec — Is the rank, or rank and size, of the associated array.

The DIMENSION statement specifies one or more objects to have the DIMENSION attribute, and specifies the array properties that apply for each object.

Following is an example of the DIMENSION statement:

```
DIMENSION A (10), B (10, 75), C (-3:12, *), D (10)
```

The EQUIVALENCE Statement

[NYI]: In this release of MasPar Fortran, an array used in a Fortran 90 construct cannot be equivalenced. Only Fortran 77 style arrays can be equivalenced.

The syntax of the EQUIVALENCE statement is

EQUIVALENCE (*set_list*) [, (*set_list*)] . . .

where:

set_list — Is a list of two or more data objects, separated by commas and enclosed in parentheses, that can include variables, array elements, and substring names. Each subscript and substring range expression must be a scalar-valued initialization expression of type integer. (Initialization expressions are described in Chapter 4.)

The EQUIVALENCE statement specifies that a storage area is shared by two or more objects in a program unit. An EQUIVALENCE statement *cannot* specify the following objects:

- a dummy argument
- an automatic object
- a function
- an array that will be used in a Fortran 90 array construct

Data objects specified in the same object list are **storage associated** by sharing the same storage units. This initial association can cause storage association of other data objects. Equivalenced objects do not have to be of the same length or type. The EQUIVALENCE statement does not cause any type conversion. Character data objects, however, can be storage associated only with other objects of type character.

Array elements can be designated for association with or without subscripts. If an array element is referenced using subscripts, the number of subscripts must be the same as the number of dimensions in the array. If an array name is referenced without a subscript, the compiler interprets the reference as if the first element of the array had been specified explicitly. If an EQUIVALENCE statement storage-associates an element of one array with an element of another array, all corresponding elements become storage associated.

These are examples of valid EQUIVALENCE statements:

```
REAL A
INTEGER B
EQUIVALENCE (A, B)   ! A and B share the same storage.

REAL x
REAL y (3)
EQUIVALENCE (x, y)   ! x shares storage with the first
                     ! element of y.

DIMENSION A (5,4), B (20)
EQUIVALENCE (A (1,1), B (1))   ! All elements of A and B
                               ! are storage associated.
```

The same storage unit *cannot* occur more than once in a sequence, and consecutive storage units *cannot* be specified in a way that would make them nonconsecutive.

The COMMON Statement

NOTE: In this release of MasPar Fortran, if the –nofecommon command-line option is used, arrays and scalars cannot appear in the same COMMON block; a COMMON block must contain only arrays *or* scalars, not both. (The –nofecommon command-line option is discussed in the *MasPar Fortran User Guide*.)

The syntax of the COMMON statement is

COMMON [/[*cname*]/] *nlist*[[,] [/[*cname*]/] *nlist*] . . .

where:

cname — Is a COMMON block name, which can be the same as a variable name or an array name in the same program unit. The COMMON block name, however, *cannot* be the same as the name of a function, a subroutine, or the entry name of the executable program.

nlist Is a list that includes names of variables, or names of arrays, or both. Names are separated by commas. An array name can be followed by an explicit-shape specification with integer constant expressions to specify the bounds. The list *cannot* include dummy arguments, automatic objects, zero-sized arrays or character strings, and function names.

The COMMON statement specifies areas of physical storage, or **COMMON blocks**, that are accessible to any program unit in an executable program, thus creating a global data facility. COMMON blocks can be identified with a name, or remain unnamed as **blank COMMON**. The size of a COMMON block is the size of its storage sequence.

Data objects that appear in a COMMON statement using a COMMON block name are declared to be in that COMMON block. More than one COMMON statement can be specified in a program unit, and a COMMON block name can be specified more than once in the same or different COMMON statements. The list of associated names for a COMMON block is continued when a COMMON block name is specified more than once.

The same principle applies for statements that specify blank COMMON. The first COMMON statement without a specified COMMON block name declares the associated objects to be in blank COMMON. Subsequent declarations without a specified COMMON block name continue the list of objects associated with blank COMMON.

Blank COMMON can alternatively be declared using two slashes with no name between them. The following two statements are equivalent:

```
COMMON A, B, C
COMMON // A, B, C
```

Variables and arrays are shared among program units by inserting the COMMON definition in each program unit that requires access to one or more objects in the list. The variable names associated with a named COMMON block in one program unit can differ from the names associated with the same COMMON block declared in another program unit.

The size of a COMMON block must be the same in all program units that use it. Also, the size and shape of arrays within the COMMON block cannot be changed between program units. These restrictions hold for both named and blank COMMON. However, the names of the variables within the COMMON block *can* be changed between units.

Data objects in blank COMMON blocks cannot be initially defined. Data objects in named COMMON blocks can be initially defined using the DATA statement within a block data program unit.

Storage sequences from the same or different COMMON blocks can be storage associated using an EQUIVALENCE statement unless they are both stored on the front end. Assumptions regarding storage sequence cannot be made for arrays in COMMON blocks. Depending on the context in which a COMMON block is used and whether or not the –nofecommon command-line option is used, the COMMON array data can be

stored contiguously on the front end or distributed on the DPU. For example, in the following code fragment A and B are not necessarily contiguous.

```
INTEGER, DIMENSION (10) :: A,B
COMMON/PLEBEIAN/ A,B
```

Objects of type character cannot be included in COMMON statements with objects of other data types. See the *MasPar Fortran User Guide* for details on how the compiler stores variables.

The IMPLICIT Statement

The syntax of the IMPLICIT statement is

IMPLICIT *type_spec* (*letter*[–*letter*]) [,*type_spec* (*letter*[–*letter*])] . . .

or

IMPLICIT NONE

where:

type_spec	Is one of the data type specifiers supported by MasPar Fortran, including length parameters.
letter	Is any letter A-Z or a-z; the compiler ignores the case of the letter(s). If the hyphen and a second letter appear, the second letter must come alphabetically after the first letter. The form with the hyphen is equivalent to specifying all letters in between and including the first and second letters. For example, W-Z is equivalent to W, X, Y, Z.

The IMPLICIT statement changes the implicit data type for objects whose names begin with a letter specified in the statement. IMPLICIT NONE specifies that no implicit typing rules apply for this program unit, and all variables must be explicitly typed. If IMPLICIT NONE is specified, no other IMPLICIT statement can appear in the program unit. If IMPLICIT NONE is not specified, then each letter can appear in only one IMPLICIT statement in the program unit. The data type explicitly specified in a data type declaration statement takes precedence over any implicit typing rules for the program unit. Intrinsic function results retain their defined data type and are not affected by the IMPLICIT statement.

Objects which are not explicitly declared or are not intrinsic functions are implicitly declared to be the type mapped from the first letter of its name. Names beginning with a dollar sign ($) or underscore (_) cannot be implicitly typed. If no IMPLICIT statement appears in the program unit, implicit data typing assumes the default for letters and associated types, as if the program unit contained this statement:

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

The following are examples of the IMPLICIT statement:

```
IMPLICIT INTEGER (A-H)
IMPLICIT LOGICAL (X, Y), COMPLEX (K, N)
```

NAMELIST Statement

The NAMELIST statement defines a list of variables or array names and associates that list with a unique group-name. The group-name is used in the namelist I / O statement to identify the variables or arrays that are to be read or written.

The syntax of the NAMELIST statement is

NAMELIST */group-name/namelist*[[,]*/group-name/namelist*] ...

where:

group-name	Is a symbolic name.
namelist	Is a list of variable or array names, separated by commas, that is to be associated with the preceding group-name.

The namelist associates with a group of entities (variables or arrays) with a single group-name, which is used by namelist I / O statements instead of an I / O list. The unique group-name identifies a list whose entities can be modified or transferred.

You cannot include array element, character substrings, pointers, records, and record fields in a namelist, but you can use namelist I / O to assign values to elements of arrays or substrings of character variables that appear in namelists.

The namelist entities can have any data type and can be explicitly or implicitly typed.

Only the entities specified in the namelist can be read or written in namelist I / O. It is not necessary for the input records in a namelist input statement to define every entity in the associated namelist.

The order of entities in the namelist controls the order in which the values are written in the namelist output. Input of namelist values can be in any order.

A variable or an array name can appear in several namelists. Dummy arguments cannot appear in a namelist.

In the following example, the NAMELIST statement defines two group-names: INPUT, with the entities NAME, GRADE, and DATE; and OUTPUT, with the entities TOTAL and NAME.

```
CHARACTER*30 NAME(25)
NAMELIST /INPUT/ NAME, GRADE, DATE /OUTPUT/ TOTAL, NAME
```

The INCLUDE Line

Additional text can be incorporated into the source text of a program unit during processing using the INCLUDE line. An INCLUDE line is not an executable Fortran statement. The file that is included via the INCLUDE line can include Fortran statements.

The syntax of the INCLUDE line is

INCLUDE *file-spec*

where:

file-spec — Is a character string that specifies the name of a file to be included during compilation. The file specification must conform to the format required by the operating system.

The INCLUDE line directs the compiler to read statements from the specified file. The statements are incorporated in the compilation directly following the INCLUDE line. When the compiler reaches the end of the included file, the compiler resumes compilation with the statement directly following the INCLUDE line.

INCLUDE lines can be nested in the included file. Each statement must be completely contained in one file. An included file cannot begin with a continuation line. Any MasPar Fortran statement and comments can appear in an included file.

The INCLUDE line can appear anywhere within a program unit; however, in general, it should appear only before all executable statements. Restrictions on statement order must be observed when included statements are combined with the other statements in the compilation sequence (see Table 1-1).

The most important use of INCLUDE files is to use identical declaration statements in different program units. However, with the current release debugging information is not supported for executable statements within INCLUDE files.

The following code fragment shows an example of the INCLUDE line:

```
PROGRAM MAIN
INCLUDE 'INSERT.FOR'          file INSERT.FOR
DIMENSION B (M)                       PARAMETER (M = 100)
                                      INTEGER x, y
.                                     REAL A (M)
.
END
```

Chapter 4

Expressions and Assignments

This chapter describes how MasPar Fortran data entities can be combined in expressions and assignment statements. Topics covered include:

- expressions
- assignment statements
- masked array assignment, or WHERE statement **[90]**
- element array assignment, or FORALL statement **[MP.EXT]**

Expressions

An **expression** consists of one or more data references, called **operands**, that can be combined with predefined **operators** in computational procedures. An expression evaluates to a scalar value or to an array of like scalar values. The resultant value has a type and shape determined by the operators and the type and shape of the operands.

MasPar Fortran supports the following kinds of expressions:

- numeric expressions
- character expressions
- relational expressions
- logical expressions

Scalars, whole arrays **[90]**, and array sections **[90]** can be used as operands in expressions. For expressions with more than one array operand, the arrays must be **conformable**; that is, they must have the same shape. The operation is applied to corresponding elements of the arrays, and the result is an array of the same shape as the operands.

Scalars are considered conformable to arrays of any shape. If one operand of an expression is an array and another operand is a scalar, the value of the scalar is broadcast in the evaluation of the expression. It is as if the value of the scalar were duplicated to form an array of the same shape as the array operand. The result is an array of the same shape as the array operand.

Numeric Expressions

A **numeric expression** consists of numeric operands and numeric operators. Numeric operands can be any of the following data objects:

- numeric constants
- numeric scalar variables
- numeric arrays
- numeric function references
- numeric expressions

Operands for numeric expressions are of numeric data types, including INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX.

Numeric operators specify computations using the values of the operands. The result is a scalar numeric value or an array whose elements are scalar numeric values. The valid numeric operators are as follows:

Operator	Meaning	Example
**	Exponentiation	a ** 3
*	Multiplication	a * 3
/	Division	a / 3
–	Subtraction	a – 3
–	Negation	–3
+	Addition	a + 3
+	Identity	+3

Each numeric operator has an associated **precedence** that determines its order in the evaluation of an expression. Numeric operators are evaluated in the following order:

1. Exponentiation (**)
2. Multiplication and Division (* and /)
3. Negation and Identity (– and +)
4. Addition and Subtraction (+ and –)

Operators of equal precedence are evaluated in left-to-right order, except for exponentiation, which is evaluated in right-to-left order. For example, in the expression A**B**C, (B**C) is evaluated first, and A is raised to the resulting power.

The compiler maintains the integrity of parentheses by treating what is enclosed in parentheses as a single entity, evaluating the parts of expressions enclosed in parentheses first. If more than one part of an expression is enclosed in parentheses, the compiler evaluates them in order of precedence. With nested parentheses, the innermost parentheses are evaluated first. The examples that follow show how the order and the result change by using parentheses.

```
8 + 4 * 9 - 6 / 2
  [4*9=36]
  [6/2=3]
  [8+36=44]
  [44-3] = 41

(8 + 4) * 9 - 6 / 2
  [8+4=12]
  [12*9=108]
  [6/2=3]
  [108-3] = 105
```

```
(8 + 4 * 9 - 6) / 2
  [4*9=36]
  [8+36=44]
  [44-6=38]
  [38/2] = 19

(8 + 4 * (9 - 6)) / 2
  [9-6=3]
  [4*3=12]
  [8+12=20]
  [20/2] = 10
```

When operands in a numeric expression are of the same type, the result is the same type as the operands. Numeric operands, however, can differ in type.

The numeric data types and their ranking for mixed-mode expressions are as follows:

Data Type	Ranking
INTEGER	1
REAL	2
DOUBLE PRECISION	3
COMPLEX	4
DOUBLE COMPLEX	5

When operands of different numeric types are combined in a numeric expression, the compiler promotes the data type of lower ranking operands to the data type of the highest ranking operand appearing in the expression. The data type of the resultant value is also the type of the highest ranking operand. For example, if an expression contains one operand of type integer and one operand of type real, the integer value is promoted to real, and the resultant value is real. However, an operation involving a complex data type and a double precision data type produces a double complex result. The same rule applies to exponentiation. An integer raised to a power of type real produces a result of type real (e.g., the integer is promoted to the type real and the exponentiation is performed). REAL**INTEGER is an exception; the integer is not promoted, but the result is real.

When division involves two integer operands, the result is an integer truncated towards zero. For example, the result of 6/3 is 2; the result of 8/3 is also 2.

Character Expressions

A **character expression** consists of a character operator that concatenates two operands of type character. The evaluation of a character expression produces a result of type character. Character operations *cannot* be applied to whole arrays or array sections. Character operands can be any of the following data objects:

- character constants
- character variables
- character substrings
- character function references
- character expressions

The form of the character expression is

char_op [*//char_op*] . . .

The value of the first character operand is concatenated on the right with the value of the second character operand. The length of the resultant value is the sum of the lengths of the values of the operands. For example, the expression 'AB' // 'CDE' yields 'ABCDE', with a length of five. Parentheses do not affect the evaluation of a character expression.

Relational Expressions

A **relational expression** consists of two or more operands whose values are compared using relational operators. Relational operations can be applied to scalars, whole arrays **[90]**, and array sections **[90]**. The valid relational operators are as follows:

Operator[3]	Meaning
.LT. or <	less than
.LE. or <=	less than or equal to
.EQ. or ==	equal to
.NE. or /=	not equal to
.GT. or >	greater than
.GE. or >=	greater than or equal to

Relational operators are of equal precedence. Numeric operators take precedence over relational operators. The evaluation of a relational expression produces a result of type logical.

In a **numeric relational expression** the operands are numeric expressions. The result is .TRUE. if the relation specified by the operator is satisfied; the result is .FALSE. if the relation specified by the operator is not satisfied.

When either or both components of a relational expression involve a numeric expression, each component is evaluated separately. The difference of the results is then tested against zero. For example, the numeric relational expression

```
A + B .LE. I – J
```

is evaluated as

```
((A+B) – (I–J)) .LE. 0
```

The data types of the operands are converted in mixed-mode numeric relational expressions. Operands of type complex can only be compared using the equal operator (.EQ. or ==) or the not-equal operator (.NE. or /=).

A **character relational expression** uses operands only of type character; it cannot contain numeric operands. In character relational expressions, less than (<) means that the character precedes in the collating sequence; greater than (>) means that the character follows in the collating sequence. Two character operands are equal if every character of one operand is the same as the character in the corresponding position of the other operand. If two character operands are not the same length, the shorter operand is treated as if padded with blanks on the right.

---

[3]The symbols shown on the right of this column are derived from Fortran 90.

Logical Expressions

A **logical expression** consists of one or more operands of type logical combined with predefined logical operators to yield a result of either .TRUE. or .FALSE.. Operands can be constants, variables, functions, and expressions of type logical. Logical operations can be applied to scalars, whole arrays, and array sections. The logical operators are as follows:

Operator	Meaning
.NOT.	Logical negation (unary operator). The expression yields true if the operand is false, and yields false if the operand is true.
.AND.	Logical conjunction. The expression yields true if both operands are true and yields false if one or both operands are false.
.OR.	Logical disjunction. The expression yields true if one operand or both operands are true and yields false if both operands are false.
.NEQV.	Logical inequivalence. The expression yields true only if exactly one of the operands is true.
.EQV.	Logical equivalence. The expression yields true if both operands are true or false and yields false if one operand is true and the other is false.

Logical operators are evaluated in the following order:

1. .NOT.
2. .AND.
3. .OR.
4. .NEQV. and .EQV. have equal precedence and are evaluated in left-to-right order.

Relational operators have precedence over logical operators.

Summary of Operator Precedence

Following are the operators supported by MasPar Fortran, listed from highest precedence to lowest.

Category	Operator
Numeric	**
	* or /
	unary + or –
	binary + or –
Character	//
Relational	.EQ., .NE., .LT., .LE., .GT., .GE.,
	==, /=, <, <=, >, >=
Logical	.NOT.
	.AND.
	.OR.
	.EQV. or .NEQV.

Specification Expressions and Initialization Expressions

Specification expressions appear only in the declaration of arrays. Explicit-shape arrays can be declared with a specification expression in any bound of any dimension. Assumed-size arrays can be declared with a specification expression in any bound of any dimension, except the upper bound of the rightmost dimension. Assumed-shape arrays **[90]** can be declared with a specification expression in the lower bound of any dimension.

Specification statements must be scalar-valued and must be of type integer. MasPar Fortran has restricted the primaries that can appear in a specification expression. Valid specification expressions can only include primaries that are

- constants
- scalar variable names that are dummy arguments or in a common block
- **[90]:** specification expressions enclosed in parentheses
- **[90]:** references to any elemental intrinsic function (functions are explained in Chapter 9; each argument must conform to the restrictions listed here for specification expressions)
- **[90]:** references to character inquiry functions or array inquiry functions; each argument must be a specification expression

Initialization expressions must evaluate at compile time to a scalar-valued constant. In initialization expressions the second operand of any exponential operators must be of type integer. Valid initialization expressions can only include primaries that are

- constants
- initialization expressions enclosed in parentheses
- **[90]:** substring subobjects of a constant; the substring starting point and substring ending point must conform to the restrictions listed here for initialization expressions
- **[90]**: references to certain elemental intrinsic functions (each argument must conform to the restrictions listed here for initialization expressions); the references must be from the following list (associated specific names can also be used): MAX, MIN, DIM, LLT, LLE, LGT, LGE, ICHAR, CHAR, ABS, CONJG, MOD, CMPLX, NINT, DPROD, INT, IFIX, REAL, DBLE, FLOAT, and AIMAG.
- **[90]:** references to the following array inquiry functions: SIZE(ARRAY), SIZE(ARRAY,DIM), LBOUND(ARRAY,DIM), UBOUND(ARRAY,DIM).
- The bounds inquired about must conform to the restrictions listed here for initialization expressions. LBOUND(ARRAY) and UBOUND(ARRAY) are not permitted, because the function call would produce an array-valued result. (ARRAY is an array name.)
- **[90]:** references to the character inquiry function LEN(STRING); the string inquired about must have a length or bound that is an initialization expression (STRING is a scalar variable name).

Initialization expressions can be used anywhere that specification expressions can be used. In addition, expressions used in some other situations, such as the PARAMETER statement, must be initialization expressions. Instances in which an expression must be an initialization expression are so designated in the syntactical descriptions.

[NYI]: In this release, array-valued expressions cannot appear in any expression within a PARAMETER statement.

Assignments

Assignments cause variables to become defined or redefined. MasPar Fortran supports the following kinds of assignments:

- assignment statements
- **[90]:** masked array assignment—the WHERE statement
- **[MP.EXT]:** element array assignment—the FORALL statement

Assignment Statements

The general form of an **assignment statement** is

variable = expression

Before the assignment of a value to the variable is made, both the expression part of the assignment statement and any expressions within the variable are evaluated. No definition of expressions in the variable can affect or be affected by the evaluation of the expression part of the assignment statement. Any necessary type conversions are made before the expressions are evaluated.

For numeric and logical assignments, the variable can be a scalar, a whole array **[90]**, or an array section **[90]**. For character assignments, the variable must be a scalar.
[90]: If the expression is array-valued, the variable must also be an array or an array section. When the variable is an array, the assignment is made element by element. The array element values of the expression are assigned to corresponding elements of the array variable. Therefore, the variable and the expression must conform in shape. If a scalar expression is assigned to an array variable, the scalar value is broadcast to each element of the array variable. If the left side of an assignment statement uses a vector-valued subscript, the vector-valued subscript must not contain duplicate entries.

The expression can contain references to all or part of the variable. For example, both of these assignment statements are valid:

```
I = I + 1
STRING(1:4) = STRING(2:5)
```

A numeric assignment statement uses a numeric expression to define a variable of a numeric data type. If the expression is not the same numeric type as the variable, the expression is converted to the type of the variable before the assignment is made. If the variable is of lower precision than the expression, a loss of precision will occur, as the following examples illustrate:

```
INTEGER I
REAL A

I = 5.5                      ! the value of I is 5
A = 4.01934867E0             ! the value of A is 4.01935

REAL X(10)
...
X (1:10) = X (10:1:-1)   ! the entire expression is
                         ! evaluated before the assignment
                         ! is made, so the original values
                         ! X are reversed  [90]
```

A character assignment statement uses a character expression to define a variable of type character. If the variable and the expression have different lengths, the length of the expression is converted to the length of the variable. If the variable is longer than the expression, the expression is blank filled to the right until the length of the expression has

the same length as the variable. If the variable is shorter than the expression, the expression is truncated from the right until the expression has the same length as the variable. Array assignments are prohibited in character assignment statements.

Examples of character assignment statements follow:

```
CHARACTER(LEN=8) C1, C2, C3

C1 = 'language'              ! 8 characters
C2 = 'Hello, there'          ! 12 characters
C3 = 'help!'                 ! 5 characters
```

The value of C1 is 'language'; the value of C2 is 'Hello, t'; the value of C3 is 'help!␣␣' (3 blanks).

A logical assignment statement defines a variable of type logical. The expression can contain values that are numeric, logical, or character. The expression must evaluate to one of the logical values: .TRUE. or .FALSE..

Examples of logical assignment statements follow:

```
FLAG = .TRUE.
AZ = A .GT. B .AND. A .GT. C .AND. A .GT. D
L(1) = .TRUE.
```

Masked Array Assignment

[90]: The masked array assignment (the WHERE statement and WHERE construct) are Fortran 90 features.

In array assignment statements, the evaluation of expressions and the assignment of values can be masked according to the value of a logical expression appearing in a WHERE statement or in a WHERE construct. This is called a **masked array assignment**.

The syntax of the WHERE statement is

WHERE (*logical_expr*)*array_variable=expression*

The logical expression is an array expression that must conform to the shape of the array variable being defined. The logical expression is evaluated first. Only the elements of the array variable that correspond to the elements that have the value .TRUE. in the logical expression are defined. All other elements of the array variable are unchanged.

The following examples show the WHERE statement:

```
WHERE (A > 0.0) A = 1.0/A  ! A is an array of type REAL.
WHERE (X(:,1:N) < Y(:,1:N)) Y(:,1:N) = X(:,1:N)
```

In the first example, the WHERE ensures that a divide-by-zero will *not* be done, because the operation is "masked" by the condition `A > 0.0`.

The syntax of the WHERE construct is

WHERE (*logical_expr*)
    array_variable = *expression* ...

[ELSEWHERE
    array_variable = *expression* ...]

END WHERE

The execution of the WHERE construct causes the logical expression to be evaluated first. The assignment statements following the keyword WHERE are executed for elements that have the value .TRUE., and the assignment statements following the keyword ELSEWHERE are executed for elements having the value .FALSE.. The array variable must not evaluate to a character expression.

An example of the WHERE construct is

```
WHERE (PRESSURE <= 1.0)           ! PRESSURE, TEMP, and
   PRESSURE = PRESSURE + 1.0      ! RAINING are all arrays
   TEMP = TEMP - 5                ! of the same shape.
ELSEWHERE
   RAINING = .TRUE.
END WHERE
```

The WHERE statement can mask an array assignment statement that contains elemental function references. The function is evaluated only for elements corresponding to true values in the logical expression, as shown in the following example.

```
WHERE (Z > 0) Z = LOG(Z)   ! LOG is evaluated only when
                           ! the value of Z is positive.
```

The mask applies to all elemental function references. The mask does not apply to arguments of transformational or external functions, because the arguments can have a different shape.

Element Array Assignment

[MP.EXT]: The element array assignment (the FORALL statement) is a MasPar extension.

A parallel array assignment statement can be specified in terms of array elements or array sections using a FORALL statement. The FORALL statement can optionally have a mask. The syntax of the FORALL statement is

FORALL (*triplet_spec* [, *triplet_spec*] ... [, *mask_expr*]) *assignment_stmt*

where:

triplet_spec	Has the form *subscript_name = lowerbound : upperbound*[*:stride*] The *subscript_name* must be a scalar and must be of type integer. This name is valid only within the scope of the FORALL statement. The stride *cannot* be zero. If omitted, the stride has a default value of 1. A bound or stride cannot contain a reference to a *subscript_name*.
mask_expr	Must be a scalar logical expression, and can reference the *subscript_name*(s).
assignment_stmt	Is *array_element=expression*. The *assignment_stmt* must not contain a character expression. The *array_element* must reference all *subscript_name*(s) included in the *triplet_spec*(s).

When the FORALL statement is executed, the bounds and strides are evaluated first. Then the mask, the righthand-side expression of the *assignment_stmt*, and the lefthand-side expression of the *assignment_stmt* are evaluated, in that order. This way the mask can guard both the expression and the assignment, and all values of the expression are evaluated before any assignments are made to lefthand-side values. The mask, the expression, and the assignment are done as parallel operations to the extent possible in the current implementation (this will improve in future releases). The expression, the assignment, and the mask (if any) are evaluated over all possible combinations of the *subscript_name*(s) values; the expression evaluation and the assignment are further restricted to those combinations where the mask, if present, is TRUE.

The following examples show valid element array assignments:

```
FORALL(I=1:N,J=1:N) H(I,J) = 1.0 / REAL(I+J-1)
FORALL(I=1:N,J=1:N, A(I,J) .NE. 0.0) H(I,J)=1.0/A(I,J)
```

Generating Parallel Code with FORALL

The current implementation of the FORALL statement will generate parallel code for some cases; in the other cases it generates serial code and issues a warning at compile time.

To generate parallel code, the following restrictions apply.

Arrays indexed by FORALL subscripts must use all the FORALL subscripts exactly once, in the order in which they appear in the FORALL header. These subscripts must be "bare" —that is, not used in expressions—and they cannot appear as section bounds or strides. Additional scalar subscripts or sections not involving any FORALL *subscript_name*(s) can be used; any sections must follow the rightmost use of FORALL *subscript_name*(s) in the subscript list of any array reference.

An exception to the above appears when vector-valued subscripts are used in an array in a FORALL statement. The vector-valued-subscript arrays *must* be indexed as described above. However, the array being vector-valued-subscripted does *not* have the restriction

that all the FORALL index variables must appear once and only once and in FORALL header order. In other words, all the index variables might not be present, or they might be present multiple times, in any order. This allows many forms of FORALL statements, including the table-lookup performance technique described in the *User Guide*. Following are some examples of the FORALL statement with vector-valued subscripts:

```
FORALL (i=1:N)    A(i) = B(V(i),i)
FORALL (i=1:N, j=1:N)    A(X(i,j),Y(i,j)) = B(i,j)
```

There cannot be any transformational intrinsics nor any user-written function calls, only scalar intrinsics (not involving any FORALL subscripts) or elemental intrinsics (which can involve FORALL subscripts).

Chapter 5

Control Statements

This chapter describes the following executable constructs and statements that control the execution sequence of a MasPar Fortran program:

- executable constructs
- IF statement
- IF construct
- CASE construct **[90]**
- DO construct
- DO WHILE statement **[90]**
- unconditional GO TO statement
- computed GO TO statement
- assigned GO TO statement
- arithmetic IF statement
- CONTINUE statement
- PAUSE statement
- STOP statement
- RETURN statement

Executable Constructs

Sequences of executable statements can be grouped together and treated as a single unit. These groups of statements are called **blocks**. They appear in IF constructs, CASE constructs, and DO constructs. Control statements, external to the block, determine which block (if any) is executed or the number of times a block is repeated.

Each of these constructs begins with its own initial keyword statement and ends with a corresponding termination keyword statement. Some constructs have secondary keywords that form boundaries around the blocks of executable statements. A block can be empty. Such blocks have no effect. A block can contain a nested executable construct. In this case, the entire executable construct must be contained within the block.

Program execution cannot be transferred to the interior of a block from outside that block. Transfers of control from within a block to outside the block are allowed, as are transfers of control within a block itself. The execution of the block is completed when the last statement in the sequence has been executed, unless control is transferred out of the block before its last statement.

[90]: These constructs can be named. If a name is specified, it must immediately precede the construct and appear after column 6. The same name must be appended to the corresponding END statement of the construct.

The following is an example of a construct containing a block:

```
IF (A > 0.0) THEN
  B = SQRT (A)       ! These two statements
  C = LOG (A)        ! form a block.
END IF
```

The IF Statement

The IF statement and the IF construct provide control for program execution based on a condition determined by the resultant value of a logical expression. The IF statement controls the execution of a single statement. The IF construct is composed of blocks of statements and selects one (or none) of its blocks for execution.

The syntax of the IF statement is

IF (*logical_expr*) *statement*

The logical expression must be scalar valued and enclosed in parentheses. The statement can be any executable statement *except* another IF statement, a statement that begins or terminates an executable construct or block of statements (such as the ELSE statement, or ELSE IF statement), an END DO statement or an END statement.

When the IF statement is executed, the logical expression is evaluated first. If the resultant value is true, the associated statement is executed. If the resultant value is false, the associated statement is not executed.

Following are examples of IF statements:

```
IF (X - Y > 0.) X = 0.
IF (FLAG .OR. L .LT. M .AND. R .LE. 1.) S(I, J) = T(I, J)
```

The IF Construct

Like the IF statement, the IF construct provides control for program execution based on a condition determined by the resultant value of a scalar logical expression.

The syntax of the IF construct is

[*name*:] IF (*scalar_logical_expr*) THEN
    block
[ELSE IF (*scalar_logical_expr*) THEN [*name*]
    block]
    ...
[ELSE [*name*]
    block]
END IF [*name*]

In this syntax, the optional presence of [*name*] is derived from Fortran 90.

[90]: If a name is specified at the beginning of an IF construct, it must immediately precede the construct and appear after column 6. The same name must be appended to the corresponding END IF statement. The same name can optionally appear in the ELSE IF statement or in the ELSE statement of the same IF construct.

No more than one block in an IF construct is executed; none need be, unless an ELSE statement appears. The logical expressions are evaluated in the order in which the IF/ELSE IF statements appear, until a value of .TRUE. is found, an ELSE statement is encountered, or an END IF statement is encountered.

If a logical expression evaluates to .TRUE., or an ELSE statement is encountered, the block immediately following is executed. No remaining logical expressions in ELSE IF statements are evaluated. If none of the logical expressions evaluate to .TRUE. and no ELSE statement appears in the construct, no block in the construct is executed. Execution of the IF construct is then complete.

Statements within the IF construct *can* have labels, but control cannot be transferred to an ELSE statement or to an ELSE IF statement. Control can be transferred to an END IF statement, both from within the IF construct and from outside it.

Following are examples of IF constructs:

```
TEST: IF (A > B) THEN
          TEMP = A
          A = B
          B = TEMP
      END IF TEST

IF (X .LT. Y) THEN
      X = -X
ELSE
      Y = -Y
END IF

OUTER: IF (I < 0) THEN
  INNER: IF (J < 0 ) THEN
      X = 0.
      Y = 0.
   ELSE INNER
      Z = 0.
   END IF INNER
ELSE IF (K < 0) THEN OUTER
      Z = 1.
ELSE OUTER
      X = 1.
      Y = 1.
END IF OUTER
```

The CASE Construct

[90]: The CASE construct is a Fortran 90 feature.

The syntax of the CASE construct is

[*name*:] SELECT CASE (*case-expression*)
[CASE (*selector*) [*name*]
block]
...
[CASE DEFAULT[*name*]
block]
END SELECT [*name*]

The CASE construct enables the selection and execution of one statement block from a set. The selection is based on the value of the *case-expression*. The resultant value is called the **case index**; its data type can be integer or logical. Each statement block in the set is associated with a CASE statement, which has a **case selector** whose value or values are compared to the value of the *case-expression*. The expression must have a value that is included in at most one of the case selectors of the construct. When the value of the expression matches the value of a case selector, the associated statement block is executed, and the execution of the construct terminates.

If a name is specified, it must immediately precede the construct and appear after column 6. The same name must be specified in the corresponding END SELECT statement. The same name can optionally be included in any CASE statement in the construct.

The case selector is enclosed in parentheses and can be a scalar constant expression or a list of scalar values and ranges. The CASE statement can also have the form CASE DEFAULT.

For values of type integer or character, a range of values can be specified by constant expressions for the upper and lower values, separated by a colon. Either the upper value or the lower value can be omitted, but not both. A colon separating upper and lower values indicates a range that includes all values between the upper and lower values, including the upper and lower values. If the lower value is omitted, a colon specifies a range of values greater than or equal to the upper value. If the upper value is omitted, a colon specifies a range of values less than or equal to the lower value. The following examples are all valid case selectors.

```
CASE (1, 3, 6, 10:17, 23) ! Individual values as specified.

CASE DEFAULT              ! All possible values of the
                          ! expression not specified in
                          ! another selector of the construct.

CASE (:-1)                ! All integer values less than zero.

CASE (0)                  ! Only zero.

CASE (1:)                 ! All integer values greater than zero.
```

Values within one case selector can overlap, but values in one case selector cannot overlap the values of another case selector.

The CASE DEFAULT statement is optional and can appear anywhere in the construct. Only one CASE DEFAULT statement is permitted in any one CASE construct. The DEFAULT selector matches the value of the expression if no other CASE selector matches the value.

Examples of the CASE construct are

```
INTEGER FUNCTION SIGNUM
SELECT CASE (N)
CASE (:-1)
   SIGNUM = -1
CASE (0)
   SIGNUM = 0
CASE (1:)
   SIGNUM = 1
END SELECT
END
```

```
SELECT CASE (N)
CASE (1, 3:5, 8)
   CALL SUB1
CASE DEFAULT
   CALL SUB2
END SELECT
```

The DO Construct

The DO construct controls the repeated execution of a block of statements. The repeated sequence is called a **loop**. The DO construct has various forms, depending on whether a label is used, the type of loop control specified, and how the construct is terminated.

The syntax of the DO construct is

[*name*:] *do_statement*
block of executable statements
do_termination_statement

where:

do_statement is	DO [*label*] [[,]*loop_control*]
do_termination_statement is	END DO [*do_construct_name*]
	or
	[*label*] [*allowed executable statement**]

* See "Termination Statements" on page 5-8 for a list of statements that are *not* allowed as termination statements of a DO construct.

Paired [*labels*] *must* be used when terminating a DO construct with the CONTINUE statement. Otherwise they are not necessary.

90: In this syntax, the optional presence of [*name*] is derived from Fortran 90.

Loop Initiation and Control

A DO construct can be named. If a name is specified, it must immediately precede the *do_statement* and appear after column 6; this same name must appear in a corresponding END DO statement.

When the DO statement is executed, the DO construct becomes active.

loop_control usually has the following form:

variable = *scalar_numeric_expression1*, *scalar_numeric_expression2*
[,*scalar_numeric_expression3*]

The variable represents the actual value of the loop index. The first expression represents the initial value of the index, and the second expression represents its ending value. The third expression represents the value of the stride. This value can be negative, in which case the first value must be greater than the second value, or the loop is not executed. This value must not be zero. If no third expression appears, the default value of 1 is assumed.

The number of iterations for the loop is computed as follows:

MAX(INT((*scalar_numeric_expression2* – *scalar_numeric_expression1* + *scalar_numeric_expression3*) / *scalar_numeric_expression3*), 0)

If this value is zero, the loop is not executed. The variable must be scalar and can be integer, real, or double-precision data type. If the variable is not an integer, the number of iterations can be inaccurate, since the representation of real numbers is not exact. The expressions are converted to the type of the variable, if necessary; they must be numeric and scalar valued.

The loop is executed the number of iteration times, or until control is explicitly passed out of the loop. Each time one iteration completes, the DO variable is incremented by the stride value. The variable retains its last defined value after the loop terminates.

[MP.EXT]: Alternatively, *loop_control* can be specified as:

(*int_expr*) TIMES

The scalar integer expression specifies the number of times the loop is executed.

[90]: *loop_control* does not have to be specified at all. If not specified, execution continues until an explicit control flow leaves the loop.

The following examples are all valid.

```
DO

DO (1000) TIMES

DO I = 1,9,2

DO I = J + 4, M, -K(J)**2
```

Statement Labels

A statement label can be specified before the loop index variable. The label identifies the **termination statement** of the construct and must be associated with a statement that has the same label as specified in the DO construct. If no label appears in the DO statement, the DO loop must be terminated by the END DO statement. The termination statement must appear in the same program unit as the DO statement.

Termination Statements

The most common *do_termination* statements are the CONTINUE statement and the END DO **[90]** statement. If a DO statement specifies a name, the same name must appear in the corresponding END DO statement.

The following statements are *not* allowed as termination statements of a DO construct:

- another DO statement
- statements that mark the beginning or end of an executable construct, or statements that delineate blocks within constructs, including IF ... THEN, ELSE IF, ELSE, END IF, SELECT CASE, CASE, END SELECT, WHERE (construct), ELSEWHERE, and END WHERE
- branch statements, including unconditional GO TO, computed GO TO, assigned GO TO, and arithmetic IF
- a RETURN statement
- a STOP statement
- an END statement

EXIT Statement

[90]: A DO construct can also terminate from within the construct with an EXIT statement. The syntax of the EXIT statement is

EXIT [*name*]

When an EXIT statement is executed, control is transferred to the next executable statement after the *do_termination* statement.

In the case of nested DO constructs, a name can be specified in the EXIT statement to indicate which construct to exit from. If no name is specified, exit is taken from the innermost DO construct in which the EXIT statement appears.

(The EXIT statement *cannot* be used within a DO WHILE loop; see the discussion of DO WHILE on page 5-10.)

CYCLE Statement

[90]: The CYCLE statement abbreviates the execution of the current iteration of the loop. The syntax of the CYCLE statement is

CYCLE [*name*]

When the CYCLE statement is executed, the active iteration of the current construct, or the named construct, is terminated. The loop index is appropriately adjusted, and program execution continues with the next iteration of the current or named construct. The statement terminating the loop, if any, is not executed.

(The CYCLE statement *cannot* be used within a DO WHILE loop; see the discussion of DO WHILE on page 5-10.)

Range

DO constructs can be nested to any depth. The range of a DO construct includes all of the statements that follow the DO statement, up to and including the corresponding *do_termination* statement. The range of a nested DO construct must be fully contained within the range of another DO construct. Nested DO constructs can share the same termination statement of enclosing loops. The variable of the innermost DO loop varies most rapidly. Other constructs, such as an IF construct, can appear within a DO construct, provided the construct is completely within the loop.

Within the range of a DO loop, the DO variable cannot be assigned a value by the user program.

Examples

The following examples show DO constructs:

```
DO                              ! No loop index.  Loop
    IF (X .GT. Y) THEN          ! executes until X becomes
       Z = X                    ! greater than Y.
       EXIT
    END IF
    CALL NEWX (X)
END DO
```

```
N = 0
DO I = 1, 10
    J = 1
    DO K = 1, 5
       L = K
       N = N + 1                ! N = 50 at the end of
    END DO                      ! execution of both loops.
END DO
```

```
DO 10 (1000) TIMES
     DO 15 (50) TIMES
           .
           .
15   CONTINUE
10 CONTINUE
```

```
DO I =1,10
  A(I) = P + Q(I)
  IF (Q(I) .LT. 0) CYCLE ! If true, the next statement is
  A(I) = 0               ! not executed on this iteration.
END DO
```

Transfer of control directly to a labeled do_termination statement, for example with a GO TO statement, causes the statement to be executed on that iteration. Use of the CYCLE statement bypasses the do_termination statement.

The DO WHILE Statement

[90]: The DO WHILE statement is a Fortran 90 feature.

The DO WHILE statement is similar to the DO statement discussed in the preceding section. The difference between them is that the DO WHILE statement repeatedly executes a block of statements as long as the value of the specified logical expression remains true, whereas the DO statement executes for a fixed number of iterations.

A DO WHILE loop can only be terminated with an END DO statement.

The syntax of the DO WHILE statement is

DO [*label* [,]] WHILE (*scalar_logical_expr*)

The label must be associated with the corresponding END DO statement that appears in the same program unit as the DO WHILE statement.

When the DO WHILE statement is executed, the logical expression is tested first. If the value is true, the associated statements are executed one after another until the END DO statement is reached. Control then returns to the DO WHILE statement, which evaluates the logical expression again. The loop repeats until the logical expression evaluates to false; control is then transferred to the statement following the END DO statement.

DO WHILE loops can be nested. The range of a DO WHILE loop includes all statements after the DO WHILE statement, up to and including the corresponding END DO statement. The entire range of another executable construct can be placed within the range of a DO WHILE loop. DO WHILE statements can also be nested within other executable constructs.

Control can be transferred out of a DO WHILE loop; however, you cannot transfer control into the loop from elsewhere in the program.

NOTE: You *cannot* use the EXIT or CYCLE statements to stop execution of the DO WHILE loop before reaching the END DO statement. The EXIT and CYCLE statements can only be used in the DO loop structure. (See page 5-8.)

An example of the DO WHILE statement is

```
CHARACTER*132 LINE
I = 1
LINE(132:) = 'x'
DO WHILE (LINE(I:1) .EQ. ' ')    ! Find the first non-blank.
      I = I + 1
END DO
```

Branch Statements

A **branch statement** alters normal program execution sequence by transferring control to a labeled target statement within the same program unit. The **branch target statement** must be executable and can be the END statement or another termination statement of a program unit. In executable constructs, the branch target statement is subject to the following restrictions:

- An END SELECT statement can be a branch target statement only when branching from within its CASE construct.
- An END IF statement can be a branch target statement when branching from within its IF construct and also when branching from outside its IF construct.
- A DO termination can only be a branch target statement when branching from within its DO construct.

An example of a branch statement with a *do_termination* statement is

```
            DO 10 I=1,10
                  .
                  .
                  .
                     GOTO 10 ! Branch to statement labeled 10.
                  .
                  .
                  .
10          CONTINUE          ! Do termination statement.
```

The GO TO statement can be used within a DO construct to transfer control to another point within the DO construct, or to transfer completely out of the construct. However, a GO TO statement cannot be used to transfer into a DO construct except at the entry point of the DO construct.

The Unconditional GO TO Statement

The syntax of the unconditional GO TO statement is

GO TO *label*

When the unconditional GO TO statement is executed, control is transferred so that the branch target statement associated with the label becomes the next statement to be executed. The label must be associated with a branch target statement that appears in the same program unit as the GO TO statement. Additionally, this statement must be executable. A GO TO statement should *never* specify a branch into a block, though it *can* specify a branch

- from within a block to another executable statement in the block,

- to the terminal statement of its construct, or
- to an executable statement outside its construct.

The Computed GO TO Statement

The syntax for the computed GO TO statement is

GO TO (*label*[,*label*] . . .) [,] *scalar_int_expression*

When the computed GO TO statement is executed, the expression is evaluated first; it represents the ordinal value of a label in the associated list. Control is transferred to the statement associated with the label chosen in reference to its order, specified by the expression, in the list of labels. For example, if the statement specifies the labels (10, 20, 30, 40), and the value of the expression is 3, the statement associated with label 30 becomes the next statement to be executed.

Each label must be associated with a branch target statement that appears in the same program unit as the GO TO statement.

If the value of the expression is less than 1 or greater than the number of labels, control is transferred to the next executable statement in sequence.

An example of the computed GO TO statement is

```
GO TO (400, 425, 450, 525), INDEX + 1
```

The Assigned GO TO Statement

The syntax of the assigned GO TO statement is

GO TO *int_variable* [[,] (*label* [,*label*] . . .)]

When an assigned GO TO statement is executed, the statement associated with a specified label becomes the next statement to be executed. The label is represented with a scalar integer variable. If a parenthesized list of labels appears following the variable, the statement label assigned to the variable must be present in the list.

Before the GO TO statement is executed, the variable must be assigned a statement label value specified in an ASSIGN statement in the same program unit. The syntax of the ASSIGN statement is

ASSIGN *label* TO *int_variable*

When an ASSIGN statement is executed, the statement label is assigned to an integer variable. The statement label must appear in the same program unit as the ASSIGN statement and must be associated with a branch target statement or a FORMAT statement. While defined with a statement label value, the variable cannot be referenced in another context. The variable can be redefined with either a statement label value or an integer value and referenced accordingly.

Examples of the assigned GO TO statement are

```
ASSIGN 300 TO I
     .
     .
     .
GO TO I    ! Equivalent to GO TO 300.

ASSIGN 400 TO I
     .
     .
     .
GO TO I (200, 300, 400, 500)   ! Equivalent to GO TO 400.
```

The Arithmetic IF Statement

The syntax of the arithmetic IF statement is

IF (*numeric_expr*) *label1*, *label2*, *label3*

The numeric expression must be scalar and cannot be of type complex. Each label must be the label of a branch target statement that appears in the same program unit as the arithmetic IF statement.

When an arithmetic IF statement is executed, the numeric expression is evaluated first. If the value of the expression is less than zero, control is transferred to the statement associated with the first label. If the value of the expression is zero, control is transferred to the statement associated with the second label. If the value of the expression is greater than zero, control is transferred to the statement associated with the third label. The same label can appear more than once in an arithmetic IF statement.

An example of the arithmetic IF statement is

```
IF ((X + Y)/2) 10, 10, 15
```

The CONTINUE Statement

The syntax of the CONTINUE statement is

CONTINUE

The CONTINUE statement has no effect other than to transfer control to the next executable statement. The CONTINUE statement often serves as the termination statement for a labeled DO loop.

An example of a CONTINUE statement is

```
      DO 135 I = 1,30
       .
       .
135 CONTINUE
```

The PAUSE Statement

The syntax of the PAUSE statement is

PAUSE [*pause_msg*]

The PAUSE statement displays a message on the terminal and temporarily suspends program execution to permit you to take some action. The *pause_msg* can either be a scalar character constant or a string of one or more digits, five digits maximum. Leading zeros are ignored.

The *pause_msg* is optional. The contents in *pause_msg* are displayed on the terminal and the program is suspended until you enter a command. If you do not specify a value for *pause_msg*, the system displays the following message:

```
FORTRAN PAUSE
```

Examples of the PAUSE statement are

```
PAUSE

PAUSE 89

PAUSE 'ERROR DETECTED'
```

Execution will resume at the next executable statement if you respond by typing CONTINUE or any abbreviation of CONTINUE on the standard input device. Typing any other command will terminate execution of the program.

The STOP Statement

The syntax of the STOP statement is

STOP [*stop_msg*]

The STOP statement ends program execution and can optionally display a message at the time the program terminates. The message can either be a scalar character constant or a string of one or more digits, five digits maximum. Leading zeros are ignored.

Examples of the STOP statement are

```
STOP

STOP 98

STOP 'END OF PROGRAM'
```

The RETURN Statement

The syntax of the RETURN statement is

RETURN [*integer_expr*]

The RETURN statement transfers control from a subprogram to the program unit from which the subprogram was called. If no RETURN statement is encountered during the execution of a subprogram, execution of the END statement has the same effect. A subprogram can have more than one RETURN statement. The RETURN statement cannot appear in a main program.

When the RETURN statement appears in a function, control is transferred to the function reference in the calling program. In a subroutine, execution of the RETURN statement transfers control to the statement after the CALL statement in the calling program, unless an alternate return is specified. Alternate returns cannot be specified from a function.

A scalar *integer_expr* in the RETURN statement indicates an alternate return. The value of the expression represents the order number of an asterisk in the sequence of asterisks in the subroutine's dummy argument list. The CALL statement specifies one or more labels of executable statements in the calling program; each label is preceded by an asterisk in the actual argument list. Program execution continues in the calling program with the statement associated with the corresponding asterisk in the actual argument list. If the value of the expression is less than 1 or greater than the number of asterisks in the list, control is transferred as if the expression had not been specified.

Alternate returns are shown in the following example.

```
C   This is the calling program.

CALL SUB5 (A, *10, B, *20)
C Normal return here.
   .
   .
10 ... !Return 1 here.
   .
   .
20 ... !Return 2 here.


END
```

```
SUBROUTINE SUB5 (A1, *, B1, *)

IF (A1 .GT. 1.0) THEN
    RETURN 1
ELSE IF ( INT(A1) .EQ. 1) THEN
    RETURN
ELSE
    J = INT (A1 / Y*COS(A1))
    RETURN J
END IF
RETURN
END
```

The END Statement

The syntax of the END statement is

END

The END statement marks the end of a program unit. It must be the last source line of every program unit. In a main program, if control reaches the END statement program execution terminates. In a subprogram, a RETURN statement is implicitly executed when control reaches the END statement.

If an initial line contains END in the statement field and nothing else, it is treated as an END statement even if it is followed by continuation lines.

Chapter 6

Input and Output Statements

Input statements and output statements are used to transfer data. Input statements transfer data from external media to internal storage, or from an internal file to internal storage. Output statements transfer data from internal storage to external media, or from internal storage to an internal file.

This chapter provides an overview to I / O on the MasPar system, and describes the following statements:

- Data transfer statements:
  - READ statement
  - WRITE statement
  - PRINT statement
- OPEN statement
- CLOSE statement
- BACKSPACE statement
- ENDFILE statement
- REWIND statement
- INQUIRE statement

See Chapter 4 of the *MasPar Fortran User Guide* for details on high-performance I / O.

I / O Overview

External and internal file I / O in MasPar Fortran for the most part is the same as in standard Fortran 77. This section provides an overview of MasPar Fortran file I / O.

Records and Files

Input and output statements reference **records**. There are two kinds of records:

- data records
  - formatted
  - unformatted
- end-of-file records

A **data record** is a sequence of values. The values in a data record can be represented as formatted or unformatted. A **formatted** record consists of a sequence of characters that can be represented by the processor. Formatted records can be read or written only by formatted input or output statements. An **unformatted** record consists of a sequence of values, in a form dependent on the processor, and can contain data of any type or no data at all. Unformatted records can be read or written only by unformatted input or output statements. An **end-of-file record** can occur only as the last record of a file that is connected for sequential access. An end-of-file record has a length of zero and cannot appear elsewhere in the file. An end-of-file record is written explicitly by the ENDFILE statement.

A **file** is a collection of records. There are two kinds of files: external and internal. **External files** are files that are located on an external device, such as a disk, tape, CDROM, or a computer terminal. For each external file, there are allowed access methods, forms (formatted or unformatted), actions, and record lengths. **Internal files** are files stored in memory as values of character variables. The character values can be created using all the usual means of assigning character values, or they can be created with an output statement using the variable as an internal file. The file has only one record if the variable is a scalar. The file has one record for each element of the array if the variable is an array. Only formatted sequential access is permitted on internal files.

The use of these files is shown in Figure 6-1.

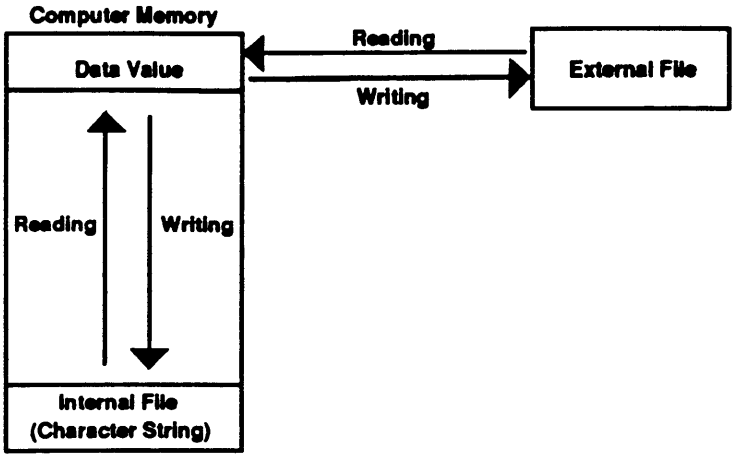

Figure 6-1 Internal and External Files

Accessing Files

External files can be accessed with sequential access or direct access. **Sequential access** to the records in the file must begin at the first record, proceed sequentially to the second record, and then to the next record, one at a time. When a file is accessed sequentially, the records are read and written sequentially (by record number).

When using **direct access** to access a file, records are selected by record number. Records thus can be read and written in any order. Some restrictions when using direct access are:

- All the records in the file to be accessed must be of the same length.
- Records cannot be deleted with direct access.
- List-directed or nonadvancing I / O is not allowed with direct access.
- Direct access cannot be used with internal files.

I / O statements refer to a particular file by specifying its **unit**. For instance, instead of referring to a file directly, READ and WRITE statements refer to a unit number, which must be connected to a file. Using an asterisk (*) as the unit number means the processor will determine the unit number (only for formatted sequential access). A specific unit number can refer to only one unit in a Fortran program.

To transfer data to or from an external file, the file must be connected to a unit. Once the connection is made, most I / O statements use the unit number instead of using the name of the file directly. An internal file always is connected to the unit that is the name of the character variable.

A **connection** between a unit and an external file can be established either by execution of an OPEN statement in the executing program, or preconnection by the operating system. Only one file can be connected to a unit at any given time. If a file is not connected to a unit, it cannot be used in any statement except the OPEN, CLOSE, or

INQUIRE statements. Units 5 and 6 are preconnected to the default input and default output files, respectively. The INQUIRE statement can be used to determine which unit numbers exist, or which files are connected to which units.

External File I / O

The most common type of external file I / O is formatted, sequential, and advancing. When an advancing sequential access I / O statement is executed, reading or writing of data begins with the next character in the file. **Advancing I / O** is record oriented. Completion of an I / O operation always positions the file at the end of a record.

Internal File I / O

Transferring data from machine representation to characters, or from characters back to machine representation, is done between two variables in an executing program. A formatted sequential access input or output statement is used. The format is used to interpret the characters. The internal file and the internal unit are the same character variable.

Some rules and restrictions for using internal files are

- The unit must be a character variable whose scope includes the data transfer statement.
- Each record of an internal file is a scalar character variable.
- If the file is an array or an array section, each element of the array or section is a scalar character variable, and thus a record. The order of the records is the order of the array elements (for arrays of rank two or greater, the first subscript varies most rapidly). The length, which must be the same for each record, is the length of one array element.
- If the number of characters written is less than the length of the record, the remaining characters are set to blank. If the number of characters is greater than the length of the record, the remaining characters are truncated.
- The records in an internal file are defined when the record is written. An internal file also can be defined by a character assignment statement, or some other means.
- To read a record in an internal file, it must have been defined.
- An internal file is always positioned at the beginning before a data transfer occurs.
- Only formatted sequential access is permitted on internal files. Namelist formatting is not allowed.
- File connection, positioning, and inquiry must not be used with internal files.
- The use of the IOSTAT, ERR, and END specifiers is the same as for external files.

Data Transfer Statements

The three data transfer statements in MasPar Fortran are READ, WRITE, and PRINT. These are discussed in the sections that follow.

The READ Statement

The READ statement causes values to be transferred from a file to the entities specified by an input list.

The syntax of a READ statement is

READ *format* [, *input-item* [, *input-item*]...]
(Reads from unit 5.)

or

READ (*io-control-spec* [, *io-control-spec*]...)
[*input-item* [, *input-item*]...]
(Reads from any specified unit.)

where *input-item* is the data whose values are to be transferred

and *io-control-spec* is

[UNIT=]	*scalar-integer-expression* or an asterisk (*)
[FMT=]	*format* or an asterisk (*)
REC=	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*
END=	*label*

Exactly one UNIT specifier must appear in any READ statement. If the "UNIT=" string is omitted, the unit value must appear as the first *io-control-spec*.

The FMT, REC, IOSTAT, ERR, and END specifiers are optional.

When a *format* specifier is present, if the "FMT=" string is omitted, the format label must appear as the second *io-control-spec*.

An example of a READ statement is

```
PRINT *, 'SIZE:'
READ (5, *) SIZE
```

In this example, the string 'SIZE:' is displayed, and the value that is read from external unit 5 is placed in the variable *SIZE*.

The READ statement input and output control specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by an asterisk (*) or a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The FMT Specifier

FMT is a parameter that is used to specify what kind of formatting should be used.

The syntax of the FMT specifier is

[FMT=] *format*

or

[FMT=] *

where

format	Is the statement label of a FORMAT statement, an integer variable that has been assigned a FORMAT statement label with an ASSIGN statement, or the name of an array, array element, or character expression containing a runtime format.
*	Indicates list-directed formatting.

The REC Specifier

The syntax of the REC specifier is

REC= *scalar-integer-expression*

The REC specifier indicates the number of the record to be read. The file must be connected for direct access to use the REC specifier.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the READ statement:

- Value of zero, if the READ statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

- Value of -1, if the processor reaches the end of the file and does not encounter an error (this condition can only occur during a sequential input statement).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the READ statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label*.

The END Specifier

The syntax of the END specifier is

END= *label*

The END specifier indicates the label of a statement in the same scoping unit. An end of file encountered during the execution of the READ statement with the END specifier produces the following results:

1. Execution of the statement terminates.
2. The file is positioned after the endfile record.
3. If the READ statement also contains an IOSTAT specifier, it becomes defined with a value of -1.
4. Execution continues with the statement specified by the *label*.

WRITE Statement

The WRITE statement causes values to be transferred to a file from entities specified by the output list and format specifications. When a WRITE statement is executed for a file that does not exist, the file is created, unless an error occurs.

If an error occurs and the ERR specifier is present, execution continues at the statement specified by the ERR specifier. If an error occurs and the IOSTAT specifier is present, execution continues at the next statement. If an error occurs and both the ERR and IOSTAT specifiers are present, execution continues at the statement specified by the ERR specifier, and a value is assigned to the IOSTAT specifier. If an error occurs and neither the ERR nor IOSTAT specifiers are present, the program terminates.

The syntax of a WRITE statement is

WRITE (*io-control-spec* [, *io-control-spec*]) [*output-item* [, *output-item*]...]

where *output-item* is

expression

or

output-implied-do

and *output-implied-do* is

(*output-item, do-variable=scalar-numeric-expr1, scalar-numeric-expr2* [, *scalar-numeric-expr3*])

and *io-control-spec* is

[UNIT=]	*scalar-integer-expression*
[FMT=]	*format*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*
REC=	*scalar-integer-expression*

Exactly one UNIT specifier must appear in any WRITE statement. If the "UNIT=" string is omitted, the unit value must appear as the first *io-control-spec*.

The FMT, REC, IOSTAT, and ERR specifiers are optional.

When a *format* specifier is present, if the "FMT=" string is omitted, the format label must appear as the second *io-control-spec*.

An example of a WRITE statement is

```
WRITE (NOUT, 100, IOSTAT=IOS, ERR=110)A
```

In this example, the value of *A* is written to the external unit defined by the variable *NOUT*. The value of *A* is formatted using the FORMAT statement at label 100, and the status is placed in the variable *IOS*. If the processor encounters an error, execution continues at the statement at label 110.

The following WRITE statement provides a simple example of the use of internal files. It converts the value of the integer variable N into the character string S of length 10:

```
CHARACTER*10 S
WRITE(S, "(I10)") N
```

If N = 999, the string S would be "*bbbbbbb*999", where "*b*" represents a blank character. To make the conversion behave a little differently, force the first character of S to be a sign and make the rest of the characters digits, using as many leading zeros as necessary.

```
WRITE (S, "(SP, I10.9)") N
```

Now if N=999, the string S will have the value "+000000999".

The WRITE statement I / O control specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by an asterisk (*) or a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The FMT Specifier

FMT is a parameter that is used to specify what kind of formatting should be used.

The syntax of the FMT specifier is

[FMT=] *format*

or

[FMT=] *

where

format	Is the statement label of a FORMAT statement, an integer variable that has been assigned a FORMAT statement label with an ASSIGN statement, or the name of an array, array element, or character expression containing a runtime format.
*	Indicates list-directed formatting.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that will contain one of the following values after the execution of the WRITE statement:

- Value of zero, if the WRITE statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the WRITE statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label.*

The REC Specifier

The syntax of the REC specifier is

REC= *scalar-integer-expression*

The REC specifier indicates the number of the record to be written. The file must be connected for direct access to use the REC specifier.

The PRINT Statement

The syntax of the PRINT statement is

PRINT *format* [, *output-item-list*]

or

PRINT * [, *output-item-list*]

The PRINT statement causes values to be transferred to a standard output device from the entities specified by the output list, if any, and the format specification. If the output list is empty, an empty record is output. When a PRINT statement is executed for a file that does not exist, the file is created. If an error occurs, execution terminates.

An example of a PRINT statement is

```
PRINT 10, A, S, J
```

This example prints the values of the variables *A*, *S*, and *J* as indicated by the FORMAT statement at label 10.

The OPEN Statement

A **unit** is a designator for devices (files or physical devices) and can be connected to a file by the execution of an OPEN statement. The OPEN statement can also modify the properties of a connection.

The syntax of an OPEN statement is

OPEN (*connect-spec* [, *connect-spec*]...)

where *connect-spec* is

[UNIT=]	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*
FILE=	*char-expression*
STATUS=	*scalar-char-expression*
ACCESS=	*scalar-char-expression*
FORM=	*scalar-char-expression*
RECL=	*scalar-integer-expression*
BLANK=	*scalar-char-expression*
POSITION=	*scalar-char-expression*
ACTION=	*scalar-char-expression*
DELIM=	*scalar-char-expression*

If the UNIT= string is omitted, the unit value must appear as the first *connect-spec*.

If the file exists, it can be connected to a unit with the execution of an OPEN statement. If an OPEN statement does not appear for a specified unit, an implicit OPEN statement is performed on the unit the first time that input or output is specified for the unit. This implicit OPEN operation is called a **preconnection**. If the file does not exist, but is preconnected to a unit, the properties specified by the OPEN statement become part of the connection. If the unit is connected to another file, the effect of the OPEN statement is as if a CLOSE statement occurred followed by an OPEN statement.

Executing a subsequent OPEN statement causes any new value of the BLANK specifier to be in effect; it does not cause changes in the values of any unspecified connection specifiers. The position of the file is also unaffected. If the file is already connected to the unit, the specifiers BLANK, ERR, and IOSTAT can have values that differ from the existing values. If the FILE specifier is omitted, the default is the name of the connected file. If an error occurs during the execution of an OPEN statement and neither IOSTAT nor ERR is specified, the program terminates.

An example of an OPEN statement is

```
OPEN (10, FILE = 'employee.names')
```

In this example, the external unit 10 is connected to a file named employee.names.

The OPEN connection specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the OPEN statement:

- Value of zero, if the OPEN statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).
- Value of -1, if the processor reaches the end of the file and does not encounter an error. This condition only occurs during a sequential input statement.

The ERR Specifier

The syntax of the ERR specifier is

ERR = *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the OPEN statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label*.

The FILE Specifier

The syntax of the FILE specifier is

FILE= *char-expression*

The FILE specifier indicates the filename to be connected to a unit. The filename can be any combination of 255 printable ASCII characters, but should always begin with an alphanumeric. Following are examples of valid filenames:

```
file1
FILE1
Long.File_name
```

The *char-expression* is case-sensitive; file1 and FILE1 are recognized as different files. Some printing characters have special meaning to the UNIX shell and so should be avoided (* , ; , $, !). Limit the characters you use in filenames to the alphanumerics, the period (.), and the underscore (_).

If the FILE specifier is omitted, the unit is connected to a file determined by the processor.

The STATUS Specifier

The syntax of the STATUS specifier is

STATUS= *scalar-char-expression*

The STATUS specifier determines the status of the file. The *scalar-char-expression* must evaluate to OLD, NEW, SCRATCH, or UNKNOWN.

- If OLD is specified, the file must exist.
- If NEW is specified, the file must not exist. NEW creates the file and changes the status to OLD.
- If SCRATCH is specified, the file cannot be a named file. SCRATCH creates the file and connects it to the specified unit. The file is deleted when an executing CLOSE statement refers to the same unit, or when the program terminates.
- If UNKNOWN is specified, the status is determined by the processor. UNKNOWN is the default value.

The ACCESS Specifier

The syntax of the ACCESS specifier is

ACCESS= *scalar-char-expression*

The ACCESS specifier indicates the access method for the file connection. The *scalar-char-expression* must evaluate to SEQUENTIAL or DIRECT. SEQUENTIAL is the default value.

[MP.EXT]: The *scalar-char-expression* can also evaluate to APPEND. This has the same meaning as if SEQUENTIAL was specified for ACCESS=, and POSITION='APPEND' was also specified.

The FORM Specifier

The syntax of the FORM specifier is

FORM= *scalar-char-expression*

The FORM specifier indicates whether the file is being connected for formatted or unformatted input or output. The *scalar-char-expression* must evaluate to FORMATTED or UNFORMATTED. UNFORMATTED is the default value if the file is connected for sequential access. FORMATTED is the default value if the file is connected for direct access.

The RECL Specifier

The syntax of the RECL specifier is

RECL= *scalar-integer-expression*

The RECL specifier indicates the length of each record connected for direct access, or it specifies the maximum length of a record connected for sequential access. The length is measured in characters. If the file exists, the specified length must be one of the allowed record lengths for the file. The record length defined by the RECL specifier in an OPEN statement cannot have a value larger than 16383.

The BLANK Specifier

The syntax of the BLANK specifier is

BLANK= *scalar-char-expression*

The BLANK specifier indicates whether blank input characters are ignored or treated as zeros. The *scalar-char-expression* must evaluate to NULL or ZERO.

- NULL specifies that all blank characters in the specified unit's numeric formatted input fields are ignored; a field of all blanks, however, has a value of zero. NULL is the default value.
- ZERO specifies that all blanks, other than leading blanks, are treated as zeros.

The BLANK specifier can only be specified for files being connected for formatted input or output.

The POSITION Specifier

The syntax of the POSITION specifier is

POSITION= *scalar-char-expression*

The POSITION specifier indicates where to start a file for sequential access. The *scalar-char-expression* must evaluate to ASIS, REWIND, or APPEND.

- REWIND positions an existing file at its initial point.
- APPEND positions an existing file such that, if present, the endfile record is the next record. If there is no endfile record, APPEND positions the file at its terminal point.
- ASIS leaves the position of the file unchanged if the file exists and is connected. If the file exists but is not connected, ASIS leaves the position unspecified.

ASIS is the default value. A file that did not exist previously (a new file, either specified explicitly or by default) is positioned at its initial point.

The ACTION Specifier

The syntax of the ACTION specifier is

ACTION= *scalar-char-expression*

The ACTION specifier indicates what kind of I / O is permitted on the connected file. The *scalar-char-expression* must evaluate to READ, WRITE, or READWRITE.

- READ specifies that the WRITE, PRINT, and ENDFILE statements are not allowed.
- WRITE specifies that READ statements are prohibited.
- READWRITE indicates that any I / O statement is allowed.

If ACTION is not specified, the default is READWRITE. However, the default for the standard input file (STDIN) is READ, and the default for the standard output and error files (STDOUT and STDERR, respectively) is WRITE.

Including READWRITE in the set of allowable actions for a file assumes that READ and WRITE are also allowed for that file.

The DELIM Specifier

The syntax of the DELIM specifier is

DELIM= *scalar-char-expression*

The DELIM specifier indicates what characters will be used as delimiters for character constants written with list-directed or namelist formatting. The *scalar-char-expression* must evaluate to APOSTROPHE, QUOTE, or NONE.

- APOSTROPHE specifies that the apostrophe will be used as the delimiter. All internal apostrophes will be doubled.
- QUOTE specifies that quotation marks will be used as the delimiter. All internal quotation marks will be doubled.
- NONE specifies that neither apostrophes or quotation marks will be used to delimit character constants, and internal apostrophes and quotation marks will not be doubled.

If DELIM is not specified, the default is NONE. The DELIM specifier is allowed only for a file being connected for formatted I / O. During input of a formatted record, this specifier is ignored.

The CLOSE Statement

A CLOSE statement disconnects an external unit from a file.

The syntax of a CLOSE statement is

CLOSE (*close-spec* [, *close-spec*]...)

where *close-spec* is

[UNIT=]	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*
STATUS=	*scalar-char-expression*

If the UNIT= string is omitted, the unit value must appear as the first *close-spec*.

A CLOSE statement can appear in any program unit of an executable program; the CLOSE statement does not have to occur in the same program unit as an OPEN statement referring to the same unit.

After the unit or file has been disconnected by a CLOSE statement, the unit (or file) can be connected again within the same executable program. If an error occurs during the execution of a CLOSE statement, and neither IOSTAT nor ERR is specified, the program terminates.

An example of a CLOSE statement is

```
CLOSE (10, IOSTAT= EFLAG)
  IF (EFLAG .NE. 0)  WRITE (*,*) 'Error closing file'
```

In this example, the file connected to external device 10 is closed. If the processor encounters an error while closing the file, the message `'Error closing file'` is displayed on the user's preconnected output device (such as a terminal screen).

The CLOSE specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the CLOSE statement:

- Value of zero, if the CLOSE statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the CLOSE statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label*.

The STATUS Specifier

The syntax of the STATUS specifier is

STATUS= *scalar-char-expression*

The STATUS specifier determines the status of the file. The *scalar-char-expression* must evaluate to KEEP or DELETE.

- If KEEP is specified for a file that exists, the file will still exist after the execution of a CLOSE statement. If KEEP is specified for a file that does not exist, the file will not exist after the execution of a CLOSE statement. KEEP must not be specified for a file whose status is SCRATCH prior to the execution of a CLOSE statement. The default value is KEEP unless the file status prior to the execution of the CLOSE statement is SCRATCH, in which case the default value is DELETE.
- If DELETE is specified, the file will not exist after the execution of the CLOSE statement.

The BACKSPACE Statement

The BACKSPACE statement causes the file connected to the specified unit to be positioned at the beginning of the preceding record. If there is no preceding record, the position of the file remains unchanged. If the preceding record is an endfile record, the file is positioned before the endfile record.

The BACKSPACE statement can be specified only if the file has been opened for sequential access. If an error occurs during the execution of a BACKSPACE statement and neither IOSTAT nor ERR is specified, the program terminates.

The syntax of a BACKSPACE statement is

BACKSPACE *scalar-integer-expression*

or

BACKSPACE (*position-spec* [, *position-spec*]...)

where *scalar-integer-expression* is an external file unit

and *position-spec* is

[UNIT=]	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*

If the UNIT= string is omitted, the unit value must appear as the first *position-spec*.

The BACKSPACE statement cannot be used for files that do not have any records written to them or for files that contain records written using list-directed or namelist formatting.

An example of a BACKSPACE statement is

```
BACKSPACE (10, ERR = 20)
```

In this example, the file connected to external device 10 is positioned at the start of the preceding record. If there was an error in the BACKSPACE statement, execution would resume with the statement at label 20.

The BACKSPACE position specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the BACKSPACE statement:

- Value of zero, if the BACKSPACE statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the BACKSPACE statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label*.

The ENDFILE Statement

The ENDFILE statement truncates the file at the current position and writes an endfile record as the next record of the file. The file is then positioned after the endfile record, which becomes the last record of the file. If the file can also be connected for direct access, only those records before the endfile record are considered as written. Therefore, only those records can be read during subsequent direct access connections to the file.

When an ENDFILE statement is executed for a file that is connected, but does not exist, the file is created.

After execution of an ENDFILE statement, a BACKSPACE or REWIND is needed to reposition the file before execution of any data transfer input/output statement or ENDFILE statement. If an error occurs during the execution of an ENDFILE statement and neither IOSTAT nor ERR is specified, the program terminates.

The syntax of an ENDFILE statement is

ENDFILE *scalar-integer-expression*

or

ENDFILE (*position-spec* [, *position-spec*]...)

where *scalar-integer-expression* is an external file unit

and *position-spec* is

[UNIT=]	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*

If the UNIT= string is omitted, the unit value must appear as the first *position-spec*.

An example of an ENDFILE statement is

```
ENDFILE K
```

In this example, the variable K contains the unit number of the file. The ENDFILE record is written to the file whose unit number is specified by K.

The ENDFILE position specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the ENDFILE statement:

- Value of zero, if the ENDFILE statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the ENDFILE statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label*.

The REWIND Statement

The REWIND statement positions a file at its initial point. If the file is already positioned at its initial point, the REWIND statement has no effect.

The REWIND statement can only be specified if the file has been opened for sequential access. If an error occurs during the execution of a REWIND statement and neither IOSTAT nor ERR is specified, the program terminates.

The syntax of a REWIND statement is

REWIND *scalar-integer-expression*

or

REWIND (*position-spec* [, *position-spec*]...)

where *scalar-integer-expression* is an external file unit

and *position-spec* is

[UNIT=]	*scalar-integer-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*

If the UNIT= string is omitted, the unit value must appear as the first *position-spec*.

An example of a REWIND statement is

```
REWIND 10
```

In this example, the file that was opened on unit number 10 is positioned at its initial point.

The REWIND position specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression.* The *scalar-integer-expression* must have a value between 0 and 99.

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the REWIND statement:

- Value of zero, if the REWIND statement executes correctly.
- A positive integer value, if the processor encounters an error (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the REWIND statement produces the following results:

1. Execution of the statement terminates.
2. The position of the file becomes indeterminate.
3. Execution continues with the statement specified by the *label.*

The INQUIRE Statement

The INQUIRE statement is used to inquire about the properties of a named file or of the connection of a unit. The INQUIRE statement allows inquiry by file or by unit.

The syntax of an INQUIRE statement is

INQUIRE (*inquire-spec* [, *inquire-spec*]...)

where *inquire-spec* is

[UNIT=]	*scalar-integer-expression*
FILE=	*char-expression*
IOSTAT=	*scalar-integer-variable*
ERR=	*label*
EXIST=	*scalar-logical-variable*
OPENED=	*scalar-logical-variable*
NUMBER=	*scalar-integer-variable*
NAMED=	*scalar-logical-variable*
NAME=	*scalar-char-variable*
ACCESS=	*scalar-char-variable*
SEQUENTIAL=	*scalar-char-variable*
DIRECT=	*scalar-char-variable*
FORM=	*scalar-char-variable*
FORMATTED=	*scalar-char-variable*
UNFORMATTED=	*scalar-char-variable*
RECL=	*scalar-integer-variable*
NEXTREC=	*scalar-integer-variable*
BLANK=	*scalar-char-variable*
POSITION=	*scalar-char-variable*
ACTION=	*scalar-char-variable*
DELIM=	*scalar-char-variable*

If you specify a unit but omit the UNIT= string, then the unit value must appear as the first *inquire-spec*.

An example of an inquiry by file is

```
INQUIRE(FILE = 'filex', EXIST = ex)
```

In this example, *ex* is set to the value .TRUE. if the file *filex* exists. Otherwise, *ex* is set to .FALSE..

An example of an inquiry by unit is

```
INQUIRE(3, EXIST = ex)
```

In this example, *ex* is set to .TRUE. if the file associated with the unit number 3 exists. Otherwise, *ex* is set to .FALSE..

Table 6-1 indicates the values assigned to the INQUIRE specifier variables.

Specifier	INQUIRE by File		INQUIRE by Unit	
	Unconnected	Connected	Connected	Unconnected
EXIST=	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
OPENED=	.FALSE.	.TRUE.		.FALSE.
NUMBER=	-1	unit no.		-1
NAMED=	.TRUE. if file named, .FALSE. otherwise			.FALSE.
NAME=	filename (might not be the same as FILE=value)		filename if named, else undefined	undefined
ACCESS=	undefined	SEQUENTIAL or DIRECT		undefined
SEQUENTIAL=	YES, NO, or UNKNOWN			UNKNOWN
DIRECT=	YES, NO, or UNKNOWN			UNKNOWN
FORM=	undefined	FORMATTED or UNFORMATTED		undefined
FORMATTED=	YES, NO, or UNKNOWN			UNKNOWN
UNFORMATTED=	YES, NO, or UNKNOWN			UNKNOWN
RECL=	undefined	if maximal record length n; else characters		undefined
NEXTREC=	undefined	if direct access, next record number; else undefined		undefined
BLANK=	undefined	NULL, ZERO, or UNDEFINED		undefined
DELIM=	undefined	APOSTROPHE, QUOTE, NONE, or UNDEFINED		undefined
POSITION=	undefined	REWIND, APPEND, ASIS, or UNDEFINED		undefined
ACTION=	undefined	READ, WRITE, or READWRITE		undefined

Table 6-1 Values Assigned to INQUIRE Specifier Variables

The INQUIRE specifiers are described in the following sections.

The UNIT Specifier

The syntax of the UNIT specifier is

[UNIT=] *scalar-integer-expression*

A unit is an external unit. An external unit refers to a file or device and is specified by a *scalar-integer-expression*. The *scalar-integer-expression* must have a value between 0 and 99.

The FILE Specifier

The syntax of the FILE specifier is

FILE= *char-expression*

The FILE specifier specifies the name of the file being inquired about. The named file does not have to exist or be connected to a unit. The filename, however, must be in a form acceptable to the processor. (See the description of file-naming conventions under "The FILE Specifier," page 6-12.)

The IOSTAT Specifier

The syntax of the IOSTAT specifier is

IOSTAT = *scalar-integer-variable*

The IOSTAT specifier defines a variable that contains one of the following values after the execution of the INQUIRE statement:

- Value of zero, if the INQUIRE statement executes correctly.
- A positive integer value, if the processor encounters an error executing the INQUIRE statement (a list of specific values are included in Table 6-2).

The ERR Specifier

The syntax of the ERR specifier is

ERR= *label*

The ERR specifier indicates the label of a statement in the same program unit. If the ERR specifier is present, an error during the execution of the INQUIRE statement produces the following results:

1. Execution of the statement terminates.
2. All of the INQUIRE specifier values become undefined except IOSTAT (if it is specified).
3. Execution continues with the statement specified by the *label*.

The EXIST Specifier

The syntax of the EXIST specifier is

EXIST= *scalar-logical-variable*

The EXIST specifier's *scalar-logical-variable* is assigned the value .TRUE. if the file or unit exists, or .FALSE. if the file or unit does not exist.

The OPENED Specifier

The syntax of the OPENED specifier is

OPENED= *scalar-logical-variable*

The OPENED specifier indicates whether the specified file (or unit) is connected to a unit (or file). The *scalar-logical-variable* is assigned the value .TRUE. if the connection exists, or .FALSE. if the connection does not exist.

The NUMBER Specifier

The syntax of the NUMBER specifier is

NUMBER= *scalar-integer-variable*

The NUMBER specifier's *scalar-integer-variable* is assigned the value of the external unit number that is currently connected to the file. If there is no unit connected to a file, the NUMBER specifier is assigned the value -1.

The NAMED Specifier

The syntax of the NAMED specifier is

NAMED= *scalar-logical-variable*

The NAMED specifier's *scalar-logical-variable* is assigned the value .TRUE. if the file has a name and the value .FALSE. if the file does not have a name.

The NAME Specifier

The syntax of the NAME specifier is

NAME= *scalar-char-variable*

The NAME specifier's *scalar-char-variable* is assigned the name of the file (if the file has a name); otherwise, it becomes undefined. The value of *scalar-char-variable* is not necessarily the same name that appears in the FILE specifier. The name, however, is valid for use in a subsequent OPEN statement.

The ACCESS Specifier

The syntax of the ACCESS specifier is

ACCESS= *scalar-char-variable*

The ACCESS specifier indicates whether the file is connected for sequential access or direct access. The *scalar-char-variable* is assigned the value SEQUENTIAL or DIRECT. If the file is not connected, the *scalar-char-variable* is assigned the value UNDEFINED.

The SEQUENTIAL Specifier

The syntax of the SEQUENTIAL specifier is

SEQUENTIAL= *scalar-char-variable*

The SEQUENTIAL specifier indicates whether SEQUENTIAL is included in the set of allowed access methods for the file. The *scalar-char-variable* is assigned the value YES or NO. If the processor cannot determine that SEQUENTIAL is an allowed access method, the *scalar-char-variable* is assigned the value UNKNOWN.

The DIRECT Specifier

The syntax of the DIRECT specifier is

DIRECT= *scalar-char-variable*

The DIRECT specifier indicates whether DIRECT is included in the set of allowed access methods for the file. The *scalar-char-variable* is assigned the value YES or NO. If the processor cannot determine that DIRECT is an allowed access method, the *scalar-char-variable* is assigned the value UNKNOWN.

The FORM Specifier

The syntax of the FORM specifier is

FORM= *scalar-char-variable*

The FORM specifier indicates if the file is connected for formatted or unformatted input or output. The *scalar-char-variable* is assigned the value FORMATTED or UNFORMATTED. If the file is not connected, the *scalar-char-variable* is assigned the value UNDEFINED.

The FORMATTED Specifier

The syntax of the FORMATTED specifier is

FORMATTED= *scalar-char-variable*

The FORMATTED specifier indicates whether FORMATTED is included in the set of allowed forms for the file. The *scalar-char-variable* is assigned the value YES or NO. If the processor cannot determine that FORMATTED is an allowed form, the *scalar-char-variable* is assigned the value UNKNOWN.

The UNFORMATTED Specifier

The syntax of the UNFORMATTED specifier is

UNFORMATTED= *scalar-char-variable*

The UNFORMATTED specifier indicates whether UNFORMATTED is included in the set of allowed forms for the file. The *scalar-char-variable* is assigned the value YES or NO. If the processor cannot determine that UNFORMATTED is an allowed form, the *scalar-char-variable* is assigned the value UNKNOWN.

The RECL Specifier

The syntax of the RECL specifier is

RECL= *scalar-int-variable*

The RECL specifier's *scalar-int-variable* is assigned the value of the maximum record length allowed for the file. If the file is connected for formatted and unformatted input or output, the length is measured in bytes. If the file does not exist, the *scalar-int-variable* becomes undefined.

The NEXTREC Specifier

The syntax of the NEXTREC specifier is

NEXTREC= *scalar-int-variable*

The NEXTREC specifier's *scalar-int-variable* is assigned the value of the last record plus one (n+1). If no records have been read since the connection, the *scalar-int-variable* is assigned the value 1. If the file is not connected for direct access, or if the file position is undetermined due to a previous error, the *scalar-int-variable* becomes undefined.

The BLANK Specifier

The syntax of the BLANK specifier is

BLANK= *scalar-char-variable*

The BLANK specifier's *scalar-char-variable* is assigned the value of NULL if null blank control is in effect for a file opened for formatted I / O; it is assigned the value ZERO if zero blank control is in effect. If the file is not opened or if it is opened, but not for formatted I / O, the *scalar-char-variable* is assigned the value UNKNOWN.

The POSITION Specifier

The syntax of the POSITION specifier is

POSITION= *scalar-char-variable*

The POSITION specifier's *scalar-char-variable* indicates where to start a file for sequential access. The *scalar-char-variable* evaluates to one of the following values:

- The value REWIND is assigned if the file is connected by an OPEN statement for positioning at its initial point.
- The value APPEND is assigned if the file is connected for positioning before its endfile record or at its terminal point.
- The value ASIS is assigned if the file is connected without changing its position.
- The value UNDEFINED is assigned if there is no connection or the file is connected for direct access.
- The value UNDEFINED is assigned if the file has been repositioned since the connection.

The ACTION Specifier

The syntax of the ACTION specifier is

ACTION= *scalar-char-variable*

The ACTION specifier indicates what kind of I / O is permitted on the connected file. The *scalar-char-variable* is assigned one of the values READ, WRITE, READWRITE, or UNDEFINED.

- READ specifies that the file is connected for input only.
- WRITE specifies that the file is connected for output only.
- READWRITE indicates that the file is connected for both input and output.

If there is no connection, the *scalar-char-variable* is assigned the value UNDEFINED.

The DELIM Specifier

The syntax of the DELIM specifier is

DELIM= *scalar-char-variable*

The DELIM specifier indicates which characters will be used to delimit character data written by list-directed or namelist formatting.

- The value APOSTROPHE is assigned if the apostrophe is used.
- The value QUOTE is assigned if the quotation mark is used.
- The value NONE is assigned if neither the apostrophe or the quotation mark is used.
- The value UNDEFINED is assigned if there is no connection or if the connection is not for formatted I / O.

Value	Condition
-1	end of file
00	no error or end of file
01	not a Fortran error
02	not implemented
03	ignored requested disposition
04	ignored requested disposition, file not deleted
17	syntax error in NAMELIST input
18	too many values for NAMELIST variable
19	invalid reference to variable
20	REWIND error
21	duplicate file specifications
22	input record too long
23	BACKSPACE error
24	end of file encountered during READ
25	record number outside range
26	OPEN or DEFINE FILE required
27	too many records in I / O statement
28	CLOSE error
29	file not found
30	open failure
31	mixed file-access modes
32	invalid logical unit number
33	ENDFILE error
34	unit already open
35	segmented record format error
36	attempt to access nonexistent record
37	inconsistent record length
38	error encountered during WRITE
39	error encountered during READ
40	recursive I / O operation
41	insufficient virtual memory

Table 6-2 Defined Values for the IOSTAT Specifier

Value	Condition
42	no such device
43	filename specification error
44	inconsistent record type
45	keyword value error in OPEN statement
46	inconsistent OPEN/CLOSE parameters
47	WRITE to read-only file
48	invalid argument to Fortran runtime library
49	invalid key specification
50	inconsistent key change or duplicate key
51	inconsistent file organization
52	specified record locked
53	no current record
54	REWRITE error
55	DELETE error
56	UNLOCK error
57	FIND error
59	list-directed I / O syntax error
60	infinite FORMAT loop
61	FORMAT/variable-type mismatch
62	FORMAT syntax error
63	output conversion error
64	input conversion error
66	output statement overflows record
67	input statement requires too much data
68	variable format expression value error
70	integer overflow
71	integer divide by zero
72	floating overflow
73	floating/decimal divide by zero
74	floating underflow

Table 6-2 Defined Values for the IOSTAT Specifier

Chapter 7

Input and Output Editing

This chapter describes the following input and output editing structures:

- the FORMAT statement
- data edit descriptors
- control edit descriptors
- character string control edit descriptors
- list-directed formatting

FORMAT statements can be used with input/output statements to edit the internal representation of data. The part of a record that is read or written and formatted with edit descriptors is called a **field**. There are two types of formatting: explicit and list-directed.

Explicit format specifications can appear in a FORMAT statement or a character expression that evaluates to a valid format specification.

List-directed formatting uses data types to control the editing. List-directed formatting is specified with an asterisk (*) in the READ, WRITE, and PRINT statements.

The FORMAT Statement

A FORMAT statement is a nonexecutable statement associated with formatted I/O statements. A FORMAT statement describes the format of the data to be transferred, and determines the data conversion and editing required to achieve that format.

The FORMAT statement can be specified as follows:

FORMAT *format-specification*

where *format-specification* is

([*format-item-list*])

where *format-item-list* is

format-item [, *format-item*...]

where *format-item* is one of the following:

[*r*] *data-edit-desc*

or

control-edit-desc

or

char-string-edit-desc

or

[*r*] (*format-item-list*)

The repeat specification, *r*, is a positive integer literal constant.

The FORMAT statement has the following constraints:

- The FORMAT statement must be labeled.
- The comma used to separate *format-items* in a *format-item-list* can be omitted in the following cases:
  - between a P edit descriptor and immediately following F, E, EN, D, or G edit descriptors
  - before or after the slash edit descriptor when the *r* repeat specification is not present
  - before or after the colon edit descriptor

The list of data edit descriptors and field separators, including the parentheses, is called a **format specification.**

An example of a FORMAT statement is

```
      PRINT 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
10    FORMAT (I12, /, ' Dates:', 2 (2I3, I5))
```

The output of this example is shown

```
           1
Dates:  2  3    4  5  6    7
 8  9   10 11 12
```

In this example, `I12` specifies the output as a right-justified integer value that uses the first 12 spaces on the line. The numeral 1 appears in the twelfth space on the first line.

The slash descriptor (/) indicates that the output should continue on a second line.

The string ' Dates:' and the values 2, 3, 4, 5, 6, and 7 are specified by the string `' Dates:', 2(2I3, I5)` and output on the second line.

The parenthetical expression, `(2I3, I5)`, indicates that the remaining values should be handled through **reversion**. Through reversion, the FORMAT statement outputs the remaining values (8, 9, 10, 11, and 12), continuing on a third line as necessary. Reversion continues until there are no remaining values.

Data Edit Descriptors

Data edit descriptors describe the size and format of a data item or of several data items. This section describes the following data edit descriptors:

- I data edit descriptor
- F data edit descriptor
- E data edit descriptor
- D data edit descriptor
- G data edit descriptor
- L data edit descriptor
- A data edit descriptor

In a format specification, a data edit descriptor has one of the following forms:

r c

r cw

r cw.m

r cw.d [*Ee*]

where:

r Is the repeat count for the data edit descriptor. The default value is 1. The range for the repeat count is 1 to 65535.

c Is the format code (I, F, E, D, G, L, A).

w Is the external width, specified in the number of characters. The range for the external width is 1 to 65535.

m Is the minimum number of characters that must be in the field (including zeros). The range for the minimum number of characters is 0 to 65535.

d Is the number of significant digits. The number of significant digits will be affected if a scale factor is specified with a data edit descriptor. The range for significant digits is 0 to 255.

E Is the exponent field.

e Is the number of characters in the exponent. The range for the number of characters in the exponent is 1 to 255.

The terms *r*, *m*, and *d*, must be unsigned integer constants. The *r* term is optional and can only appear in certain edit descriptors.

The I Data Edit Descriptor

The I data edit descriptor transfers decimal integer values. The syntax of the I data edit descriptor is

I*w*[.*m*]

On input, the I data edit descriptor transfers the specified number of characters (*w*) from an external field and assigns their integer value to a corresponding I/O list element that is a variable. (*Iw.m* is treated like *w*.) Each data item in an external medium is called an **external field.** The I/O list elements must be integer data types. The external field must be an integer constant.

Examples of input using the I data edit descriptor are

Format	External Field	Internal Value
I4	Δ-26	–26
I4	ΔΔΔΔ	0
I4	Δ+38	38
I4	2788	2788

On output, if *m* is zero or not present, and the internal representation is zero, the external field is filled with blanks. A field of blanks is treated as a value of zero. If the value of the external field exceeds the range of the corresponding I/O element, the external field is filled with asterisks (*).

Examples of output using the I data edit descriptor are

Format	Internal Value	External Representation
I4	284	Δ 284
I4.4	1	0001

The F Data Edit Descriptor

The F data edit descriptor transfers real values. The syntax of the F data edit descriptor is

F*w.d*

On input, the F data edit descriptor transfers the specified number of characters (*w*) from an external field and assigns their real value to corresponding I/O list elements. The I/O list elements must be real, double precision, or the real or imaginary part of a complex or double complex data type. A field of blanks is treated as a value of zero.

If the field does not contain a decimal point or an exponent, the field is treated as a real number of *w* digits, and the *d* digits are to the right of the decimal point. If the field contains a decimal point, the location of the decimal point overrides the location specified by the data edit descriptor. If the field contains an exponent, that exponent establishes the magnitude of the value before it is assigned to the list element.

Examples of input using the F data edit descriptor are

Format	External Field	Internal Value
F8.5	2477E+2	24.7
F8.5	123456789	123.45678
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

On output, the F data edit descriptor transfers the value of the corresponding I/O list element, rounded to *d* decimal positions and right-justified, to an external field that is a specified number of characters (*w*) long. If the value does not fill the field, leading spaces are inserted. If the value is too large for the field, the entire field is filled with asterisks.

The term *w* must be greater than or equal to *d+3* to allow an optional sign, at least one digit to the left of the optional decimal point, a required decimal point, and *d* digits to the right of the decimal point. On output, if the number is too large, the field is filled with asterisks (*).

Examples of output using the F data edit descriptor are

Format	Internal Value	External Representation
F2.1	51.44	**
F5.2	–.2	–0.20
F5.2	325.013	*****

The E Data Edit Descriptor

The E data edit descriptor transfers real values in an exponential form. The syntax of the E data edit descriptor is

E*w.d*[*Ee*]

On input, the E data edit descriptor transfers the specified number of characters (*w*) from an external field and assigns their real value to corresponding I/O list elements. The I/O list elements must be real, double precision, or the real or imaginary part of a complex or double complex data type.

Examples of input using the E data edit descriptor are

Format	External Field	Internal Value
E6.2E2	732145	73.21E45
E6.2	732145	7321.45
E9.3	732.322E3	732322.0
E12.4	ΔΔ 1022.43E–6	1022.43E–6

NOTE: On input, the E data edit descriptor and the F data edit descriptor are equivalent.

On output, the exponent has one of the following forms:

E + *nn* (Ew.d for $0 \leq exponent \leq 99$)
E – *nn* (Ew.d for $-1 \geq exponent \geq -99$)

+ *nnn* (Ew.d for $99 < exponent \leq 999$)
– *nnn* (Ew.d for $99 > exponent \geq -999$)

$E + n_1 n_2 \ldots n_e$ (Ew.dEe)
$E - n_1 n_2 \ldots n_e$ (Ew.dEe)

The exponent width specification is optional; the default value is 2. If the exponent value is too large to be output in one of the exponent forms, the external field is filled with asterisks (*).

On output, without scaling, the term *w* must be greater than or equal to *d+6*, or to *d+e+2* if *e* is present. This allows a decimal point, at least one digit, and the exponent. Scaling is discussed in the section "The Scale Factor," later in this chapter.

Examples of output using the E data edit descriptor are

Format	Internal Value	External Representation
E14.5E4	1.001	Δ 0.10010E+0001
E9.2	475867.222	Δ 0.48E+06

The D Data Edit Descriptor

The D data edit descriptor transfers real values in an exponential form. The syntax of the D data edit descriptor is

D*w.d*

On input, the D data edit descriptor transfers the specified number of characters (*w*) from an external field and assigns their real value to corresponding I/O list elements. The I/O list elements must be the real or imaginary part of a complex or double complex data type or a real or double-precision data type.

Examples of input using the D data edit descriptor are

Format	External Field	Internal Value
BZ,D10.2	12345 ΔΔΔΔ	12345000.0D0
D15.3	367.4981763D+04	3.674981763D+06

On output, the D data edit descriptor has the same effect as the E data edit descriptor.

Examples of output using the D data edit descriptor are

Format	Internal Value	External Representation
D9.6	1.2	*********
D14.3	0.0363	ΔΔΔΔΔ 0.363D–01

The G Data Edit Descriptor

The G data edit descriptor transfers real values. The syntax of the G data edit descriptor is

G*w.d*[E*e*]

On input, the G data edit descriptor and the F data edit descriptor are equivalent.

On output, the G data edit descriptor transfers the value of the corresponding I/O list element, rounded to *d* decimal positions and right-justified, to an external field that is a specified number of characters (*w*) long. The G data edit descriptor uses the magnitude of the value to determine the output form as indicated in the following table.

Value Magnitude	G Output Form
m < 0.1	E*w.d*[Ee]
0.1 ≤ m < 1.0	F(*w*–4).*d*, *n*Δ
1.0 ≤ m < 10.0	F(*w*–4).(*d*–1), *n*Δ
.	.
.	.
.	.
10**(*d*–2) ≤ m < 10***d*–1	F(*w*–4).1, *n*Δ
10**(*d*–1) ≤ m < 10***d*	F(*w*–4).0, *n*Δ
m ≥ 10***d*	E*w.d*[Ee]

In the above table the *n*Δ descriptor specifies that four or *e*+2 spaces are to follow the numeric data representation. The term *w* must be large enough to include all of the following: a minus sign when necessary (plus signs are optional); a decimal point; one digit to the left of the decimal point; *d* digits to the right of the decimal point; and either a 4-character or *e*+2-character exponent. Therefore *w* must be greater than or equal to 1+*d*+7 or 1+*d*+5+*e*.

Examples of output using the G data edit descriptor are

Format	Internal Value	External Representation
G13.6	0.01234567	Δ 0.123457E–01
G13.6	1234567.89012345	Δ 0.123457E+07