PGI® Fortran Reference

The Portland Group™
STMicroelectronics
Two Centerpointe Drive, Suite 320
Lake Oswego, OR 97035
www.pgroup.com

PGI Fortran Reference

Printed in the United States of America

First Printing:Release 6.0, March, 2005
Second Printing:Release 6.1, December, 2005
Third Printing:Release 6.1-3, February, 2006
Fourth Printing: Release 7.0-1, December, 2006
Fifth Printing: Release 7.0-2, February, 2007

Technical support: http://www.pgroup.com/support/
Sales: sales@pgroup.com
Web: http://www.pgroup.com

Contents

Tables

Figures

Preface

This help collection describes The Portland Group's implementation of the FORTRAN 77, Fortran 90/95 languages. Collectively, The Portland Group compilers that implement these languages are referred to as the PGI Fortran compilers. This help collection is part of a set of other documents describing the Fortran language and the compilation tools available from The Portland Group. This help collection presents the Fortran language statements, intrinsics, and extension directives. The Portland Group's Fortran compilation system includes a compilation driver, multiple Fortran compilers, associated runtime support and mathematical libraries, and associated software development tools for debugging and profiling the performance of Fortran programs. Depending on the target system, The Portland Group's Fortran software development tools may also include an assembler or a linker. You can use these tools to create, debug, optimize and profile your Fortran programs. '' Related Publications'' lists other manuals in the PGI documentation set.

This manual describes The Portland Group's implementation of the FORTRAN 77, Fortran 90/95 and High Performance Fortran (HPF) languages. Collectively, The Portland Group compilers that implement these languages are referred to as the PGI Fortran compilers. This manual is part of a set of other documents describing the Fortran language and the compilation tools available from The Portland Group. This manual presents the Fortran language statements, intrinsics, and extension directives. The Portland Group's Fortran compilation system includes a compilation driver, multiple Fortran compilers, associated runtime support and mathematical libraries, and associated software development tools for debugging and profiling the performance of Fortran programs. Depending on the target system, The Portland Group's Fortran software development tools may also include an assembler or a linker. You can use these tools to create, debug, optimize and profile your Fortran programs. '' Related Publications'' lists other manuals in the PGI documentation set.

Audience Description

This help collectionmanual is intended for people who are porting or writing Fortran programs using the PGI Fortran compilers. To use Fortran you should be aware of the role of Fortran and of source-level programs in the software development process and you should have some knowledge of a particular system or workstation cluster. To use the PGI Fortran compilers, you need to be familiar with the Fortran language, either FORTRAN77, or Fortran 90/95 or HPF, and the basic commands available on your host system.

Compatibility and Conformance to Standards

The PGI Fortran compilers, PGF77, or PGF95, run on a variety of 32-bit and 64-bit x86 processor-based host systems. The PGF77 compiler accepts an enhanced version of FORTRAN 77 that conforms to the ANSI standard for FORTRAN 77 and includes various extensions from VAX/VMS Fortran, IBM/VS Fortran, and MIL-STD-1753. The PGF95 compiler accepts a similarly enhanced version of the ANSI standard for Fortran 90/95.

The PGI Fortran compilers, PGF77, PGF95 and PGHPF, run on a variety of 32-bit and 64-bit x86 processor-based host systems. The PGF77 compiler accepts an enhanced version of FORTRAN 77 that conforms to the ANSI standard for FORTRAN 77 and includes various extensions from VAX/VMS Fortran, IBM/VS Fortran, and MIL-STD-1753. The PGF95 compiler accepts a similarly enhanced version of the ANSI standard for Fortran 90/95. The PGHPF compiler accepts the HPF language and is largely, though not strictly, a superset of Fortran 90/95. The PGHPF compiler conforms to the High Performance Fortran Language Specification Version 1.1, published by the Center for Research on Parallel Computation, at Rice University (with a few limitations and modifications, consult the PGHPF Release Notes for details).

For further information on the Fortran language, you can also refer to the following:

- American National Standard Programming Language FORTRAN, ANSI X3. -1978 (1978).
- ISO/IEC 1539 : 1991, Information technology – Programming Languages – Fortran, Geneva, 1991 (Fortran 90).
- ISO/IEC 1539 : 1997, Information technology – Programming Languages – Fortran, Geneva, 1997 (Fortran 95).
- Fortran 95 Handbook Complete ISO/ANSI Reference, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- High Performance Fortran Language Specification, Revision 1.0, Rice University, Houston, Texas (1993), http://www.crpc.rice.edu/HPFF.
- High Performance Fortran Language Specification, Revision 2.0, Rice University, Houston, Texas (1997), http://www.crpc.rice.edu/HPFF.
- OpenMP Fortran Application Program Interface, Version 1.1, November 1999, http://www.openmp.org.
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).

- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).

Organization

This manual is divided into the following chapters and appendices:

Chapter 1, "Language Overview", provides an introduction to the Fortran language.

Chapter 2, "Fortran Data Types", describes the data types supported by PGI Fortran compilers and provides examples using various data types. Memory allocation and alignment issues are also discussed.

Chapter 3, "Fortran Statements", describes each Fortran and HPF statement that the PGI Fortran compilers accept. Many HPF statements are in the form of compiler directives which can be ignored by non-HPF compilers.

Chapter 4, "Fortran Arrays", describes special characteristics of arrays in Fortran 90/95.

Chapter 5, "Input and Output Formatting", describes the input, output, and format statements that allow programs to transfer data to or from files.

Chapter 6, "Fortran Intrinsics", lists the Fortran intrinsics and subroutines supported by the PGI Fortran comilers.

Chapter 7, "3F Functions and VAX Subroutines", describes the functions and subroutines in the Fortran run-time library and discusses the VAX/VMS system subroutines and the built-in functions supported by the PGI Fortran compilers.

Chapter 8, "OpenMP Directives for Fortran", lists the language extensions that the PGI Fortran compilers support.

Chapter 9, "HPF Directives", describes the HPF directives which support data distribution and alignment, and influence data parallelism by providing additional information to the PGHPF compiler.

Appendix A., "HPF_LOCAL", lists the HPF_LOCAL_LIBRARY procedures supported by the PGHPF compiler.

Hardware and Software Constraints

The PGI compilers operate on a variety of host systems and produce object code for a variety of target systems. Details concerning environment-specific values and defaults and host-specific features or limitations are presented in the PGI User's Guide, the man pages for each compiler in a given installation, and in the release notes and installation instructions included with all PGI compilers and tools software products.

Conventions

This PGI Fortran Reference manual uses the following conventions:

italic	is used for commands, filenames, directories, arguments, options and for emphasis.
`Constant Width`	is used in examples and for language statements in the text.
[item]	square brackets indicate optional items. In this case item1 is optional.
{ item2 | item3}	braces indicate that a selection is required. In this case, you must select either item2 or item3.
filename ...	ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.
FORTRAN	Fortran language statements are shown using upper-case characters and a reduced point size.
<TAB>	non-printing characters, such as TAB, are shown enclosed in greater than and less than characters and a reduced point size.
§	this symbol indicates an area in the text that describes a Fortran 90/95 Language enhancement. Enhancements are features that are not described in the ANSI Fortran 90/95 standards.
@	This symbol indicates an area in the text that describes a FORTRAN 77 enhancement. Enhancements may be VAX/VMS Fortran enhancements, IBM/VM enhancements, or military standard MIL-STD-1753 enhancements.

Related Publications

The following documents contain additional information related to HPF and other compilers and tools available from The Portland Group, Inc.

- The PGI User's Guide describes the general features and usage guidelines for all PGI compilers, and describes in detail various available compiler options in a user's guide format.
- The PGHPF User's Guide describes the PGHPF compiler and describes some details concerning the PGI implementation of HPF in a user's guide format.
- Fortran 95 Handbook, from McGraw-Hill, describes the Fortran 95 language and the statements, data types, input/output format specifiers, and additional reference material that defines ANSI/ISO Fortran 95.
- System V Application Binary Interface Processor Supplement by AT&T UNIX System Laboratories, Inc, (available from Prentice Hall, Inc.)
- The High Performance Fortran Handbook, from MIT Press, describes the HPF language in detail.
- High Performance Fortran Language Specification, Rice University, Houston Texas (1993), is the specification for the HPF language and is available online at http://www.crpc.rice.edu/HPFF.
- American National Standard Programming Language Fortran, ANSI x.3-1978 (1978).
- Programming in VAX FORTRAN, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS FORTRAN, IBM Corporation, Rev. GC26-4119.
- Military Standard, FORTRAN, DOD Supplement to American National Standard Programming Language FORTRAN, ANSI X3.-1978, MIL-STD-1753 (November 9, 1978).

1 Language Overview

This chapter describes the basic elements of the Fortran language, the format of Fortran statements, and the types of expressions and assignments accepted by the PGI Fortran compilers.

The PGF77 compiler accepts as input FORTRAN 77 and produces as output assembly language code, binary object code or binary executables in conjunction with the assembler, linker and libraries on the target system. The input language must be extended FORTRAN 77 as specified in this reference manual. The PGF95 compiler functions similarly for Fortran 90/95. The PGF95 and PGHPF compilers function similarly for Fortran 90/95 and HPF respectively.

This chapter is not an introduction to the overall capabilities of Fortran. Rather, it is an overview of the syntax requirements of programs used with the PGI Fortran compilers. The Fortran 95 Handbook provides details on the capabilities of Fortran 90/95 language.The Fortran 95 Handbook and The High Performance Fortran Handbook provide details on the capabilities of Fortran 90/95 and HPF languages.

Elements of a Fortran Program Unit

A Fortran program is composed of SUBROUTINE, FUNCTION, MODULE, BLOCK DATA, or PROGRAM program units.

Fortran source code consists of a sequence of program units which are to be compiled. Every program unit consists of statements and optionally comments beginning with a program unit statement, either a SUBROUTINE, FUNCTION, or PROGRAM statement, and finishing with an END statement (BLOCK DATA and MODULE program units are also allowed).

In the absence of one of these statements, the PGI Fortran compilers insert a PROGRAM statement.

Statements

Statements are either executable statements or nonexecutable specification statements. Each statement consists of a single line or source record, possibly followed by one or more continuation lines. Multiple statements may appear on a single line if they are separated by a semicolon (;). Comments may appear on any line following a comment character (!).

Free and Fixed Source

Fortran permits two types of source formatting, fixed source form and free source form. Fixed source form uses the traditional Fortran approach where specific column positions are reserved for labels, continuation characters, and statements and blank characters are ignored. The PGF77 compiler supports only fixed source form. The PGF77 compiler also supports a less restrictive variety of fixed source form called tab source form. Free source form introduced with Fortran 90 places few restrictions on source formatting; the context of an element, as well as the position of blanks, or tabs, separate logical tokens. Using the compiler option –Mfreeform you can select free source form as an option to PGF95 or PGHPF.

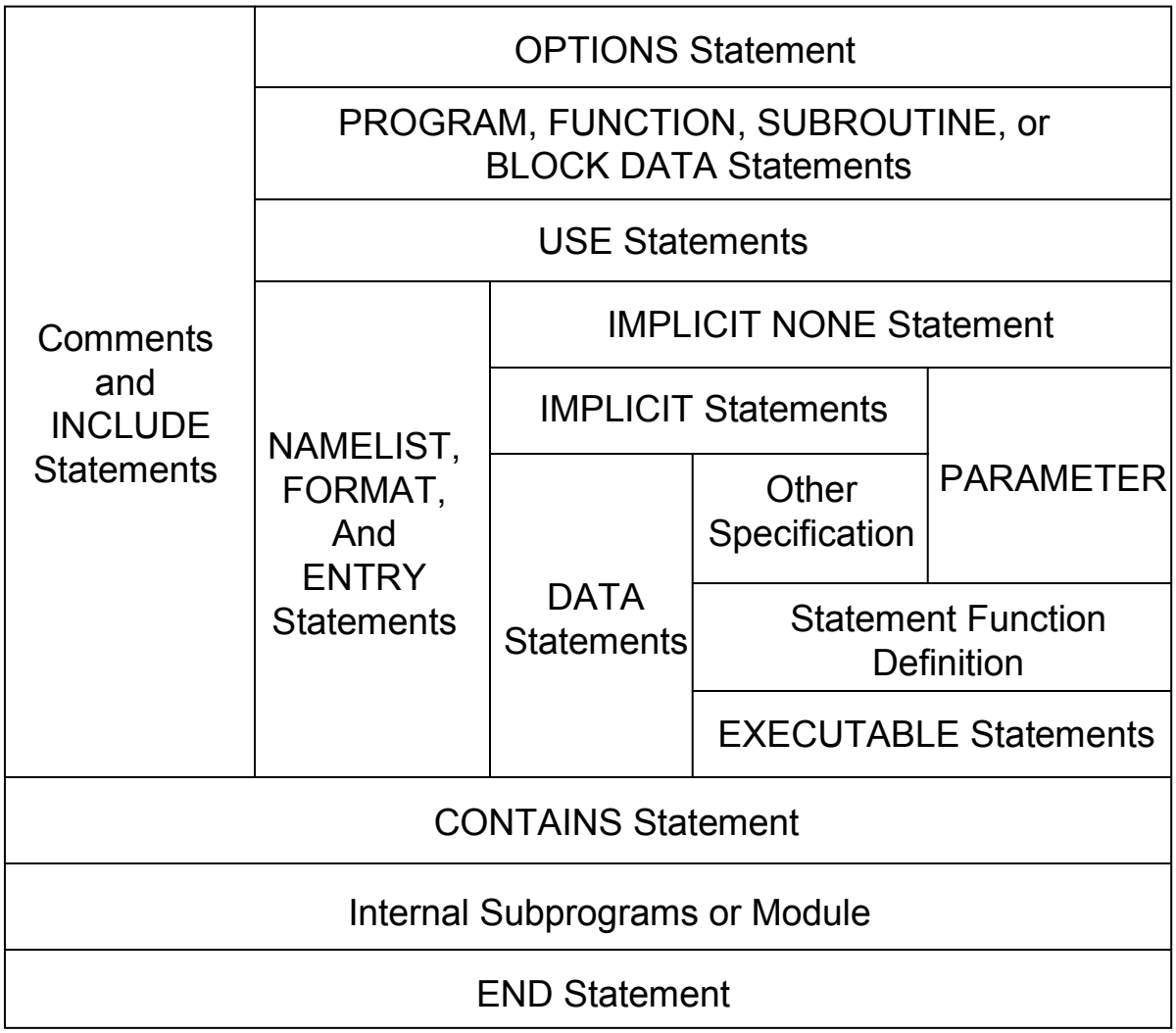
Statement Ordering

Fortran statements and constructs must conform to ordering requirements imposed by the language definition. The figure "Order of Statements" illustrates these requirements. Vertical lines separate statements and constructs that can be interspersed. Horizontal lines separate statements that must not be interspersed.

These rules are less strict than those in the ANSI standard. The differences are as follows:

- DATA statements can be freely interspersed with PARAMETER statements and other specification statements.
- NAMELIST statements are supported and have the same order requirements as FORMAT and ENTRY statements.
- The IMPLICIT NONE statement can precede other IMPLICIT statements.

Figure 1-1: Order of Statements



The Fortran Character Set

Table 1-1 , "Fortran Characters", shows the set of Fortran characters. Character variables and constants can use any ASCII character. The value of the command-line option –Mupcase determines if the compiler distinguishes between case (upper and lower) in identifiers. By default, without the –Mupcase option selected, the compiler does not distinguish between upper and lower case characters in identifiers

(upper and lower case are always significant in character constants).

Table 1-1: Fortran Characters

Character	Description	Character	Description
,	Comma	A-Z, a-z	Alphabetic
:	Colon	<space>	Space character
;	Semicolon	=	Equals
_	Underscore character	+	Plus
<	Less than	-	Minus
>	Greater than	*	Asterisk
?	Question mark	/	Slash
%	Percent	(	Left parenthesis
"	Quotation mark	)	Right parenthesis
$	Currency symbol	[	Left bracket
.	Decimal point	]	Right bracket
!	Exclamation mark	<CR>	Carriage return
0-9	Numeric	<TAB>	Tabulation character

Table 1-2 , "C Language Character Escape Sequences", shows C language character escape sequences that the PGI Fortran compilers recognize in character string constants. These values depend on the command-line option –Mbackslash.

Table 1-2: C Language Character Escape Sequences

Character	Description
\v	vertical tab
\a	alert (bell)
\n	newline
\t	tab
\b	backspace
\f	formfeed
\r	carriage return
\0	null
\'	apostrophe (does not terminate a string)
\"	double quotes (does not terminate a string)
\\	\
\x	x, where x is any other character
\ddd	character with the given octal representation.

Free Form Formatting

Using free form formatting, columns are not significant for the elements of a Fortran line, and a blank or series of blanks or tabs and the context of a token specify the token type. 132 characters are valid per line, and the compiler option –Mextend does not apply. Comments are indicated by a blank line, or by following a Fortran line with the ! character. All characters after the ! are stripped out of the Fortran text.

Using free form formatting, the & character at the end of a line means the following line represents a continuation line. If a continuation line starts with the & character, then the characters following the & are the start of the continuation line. Without a leading & at the start of the continuation line, all characters on the line are part of the continuation line, including any initial blanks or tabs.

A single Fortran line may contain multiple statements. The ; (semicolon) separates multiple statements on a single line. Free format labels are valid at the start of a line, as long as the label is separated from the remaining statements on the line by at least one blank or a <TAB>. Labels consist of a numeric field drawn from digits 0 to 9. The label cannot be more than 5 characters.

Fixed Formatting

This section describes the two types of fixed formatting that PGI Fortran compilers support, column formatting and tab formatting.

Column Formatting

Using column formatting a Fortran record consists of a sequence of up to 73 ASCII characters, the last being <CR>. There is a fixed layout as shown in the table below.

Table 1-3: Fixed Format Record Positions and Fields

Position	Field
1-5	Label field
6	Continuation field
7-72	Statement field

Characters beyond position 72 on a line are ignored unless the –Mextend option is specified. In addition, any characters following a ! character are comments and are disregarded during compilation.

Fixed Format Label Field

The label field holds up to five characters. The characters C or * in the first character position of a label field indicate a comment line.

In addition to the characters C or *, either of the characters D or ! in the first position of a label field also indicate a comment line.

When a numeric field drawn from digits 0 to 9 is placed in the label field, the field is a label. A line with no label, and with five space characters or a <TAB> in the label field, is an unlabeled statement. Each label must be unique in its program unit. Continuation lines must not be labeled. Labels can only be jumped to when they are on executable statements.

Fixed Format Continuation Field

The sixth character position, or the position after the tab, is the continuation field. This field is ignored in comment lines. It is invalid if the label field is not five spaces. A value of 0, <space> or <TAB> indicates the first line of a statement. Any other value indicates a subsequent, continuation line to the preceding statement.

Fixed Format Statement Field

The statement field consists of valid identifiers and symbols, possibly separated by <space> or <TAB> and terminated by <CR>.

Within the statement field tabs and spaces are ignored as are comments, characters following a !, or any characters found beyond the 72nd character (unless the option –Mextend is enabled).

Fixed Format Debug Statements

The letter D in column 1 using fixed formatting designates the statement on the specified line is a debugging statement. The compiler will treat the debugging statement as a comment, that is ignoring it, unless the command line option –Mdlines is set during compilation. In that case, the compiler acts as if the line starting with D were a Fortran statement and compiles the line according to the standard rules.

Tab Formatting

The PGI Fortran compilers support an alternate form of fixed source from called tab source form. A tab formatted source file is made up of a label field, an optional continuation indicator and a statement field. The label field is terminated by a tab character. The label cannot be more than 5 characters.

A continuation line is indicated by a tab character followed immediately by a digit. The statement field starts after a continuation indicator, when one is present. The 73rd and subsequent characters are ignored.

Fixed Input File Format – Summary

Tab-Format lines are supported. A tab in columns 1-6 ends the statement label field and begins an optional continuation indicator field. If a non-zero digit follows the tab character, the continuation field exists and indicates a continuation field. If anything other than a non-zero digit follows the tab character, the statement body begins with that character and extends to the end of the source statement. Note that this does not override Fortran's free source form handling since no valid Fortran statement can begin with a non-zero digit. The tab character is ignored if it occurs in a line except in Hollerith or character constants.

Input lines may be of varying lengths. If there are fewer than 72 characters, the line is padded with blanks; characters after the 72nd are ignored unless the –Mextend option is used on the command line.

If the –Mextend option is used on the command line then the input line can extend to 132 characters. The line is padded with blanks if it is fewer than 132 characters; characters after the 132nd are ignored. Note that use of this option extends the statement field to position 132.

Blank lines are allowed at the end of a program unit.

The number of continuation lines allowed is extended to 1000 lines.

Including Fortran Source Files

The sequence of consecutive compilation of source statements may be interrupted so that an extra source file can be included. This is carried out using the INCLUDE statement which takes the form:

```
INCLUDE
"filename"
```

where filename is the name of the file to be included. Pairs of either single or double quotes are acceptable enclosing filename.

The INCLUDE file is compiled to replace the INCLUDE statement, and on completion of that source the file is closed and compilation continues with the statement following the INCLUDE.

INCLUDE files are especially recommended when the same COMMON blocks and the same COMMON block data mappings are used in several program units. For example the following statement includes the file MYFILE.DEF.

```
INCLUDE "MYFILE.DEF"
```

Recursive includes are not allowed. That is, if a file includes a file, that file may not also include the same file.

Nested includes are allowed, up to a PGI Fortran defined limit of 20.

The Components of Fortran Statements

Fortran program units are made up of statements which consist of expressions and elements. An expression can be broken down to simpler expressions and eventually to its elements combined with operators. Hence the basic building block of a statement is an element. An element takes one of the following forms:

A constant represents a fixed value.

A variable represents a value which may change during program execution.

An array is a group of values that can be referred to as a whole, as a section, or separately. The separate values are known as the elements of the array. The array has a symbolic name.

A function reference or subroutine reference is the name of a function or subroutine followed by an argument list. The reference causes the code specified at function/subroutine definition to be executed and if a function, the result is substituted for the function reference.

Symbolic Names

Symbolic names identify different entities in Fortran source code. A symbolic name is a string of letters and digits, which must start with a letter and be terminated by a character not in the symbolic names set (for example a <space> or a <TAB> character). Underscore (_) characters may appear within symbolic names. Only the first thirty-one characters identify the symbolic name. Below are several examples of symbolic names:

```
NUM
CRA9
numericabcdefghijklmnopqrstuvwxyz
```

The last example is identified by its first 31 characters and is equivalent to:

```
numericabcdefghijklmnopqrstuvwx
```

The following examples are invalid symbolic names.

```
8Q
```

This is invalid because it begins with a number.

```
FIVE.4
```

This is invalid because it contains a period which is an invalid character for a symbolic name.

Expressions

Each data item, such as a variable or a constant, represents a particular value at any point during program execution. These elements may be combined together to form expressions, using binary or unary operators, so that the expression itself yields a value. A Fortran expression may be any of the following:

- A scalar expression
- An array expression
- A constant expression
- A specification expression
- An initialization expression
- Mixed array and scalar expressions

Expression Precedence Rules

Arithmetic, relational and logical expressions may be identified to the compiler by the use of parentheses, as described in "Arithmetic Expressions" on page 12. When no guidance is given to the compiler it will impose a set of precedence rules to identify each expression uniquely. Table 1-4 , "Fortran Operator Precedence", shows the operator precedence rules for expressions.

Table 1-4: Fortran Operator Precedence

Operator	Evaluated
Unary defined	Highest
**	N/A
* or /	N/A
Unary + or -	N/A
Binary + or –	N/A
Relational operators: GT., .GE., .LE.	N/A
Relational operators ==, /=	Same precedence
Relational operators <, <=, >, >=	Same precedence
Relational operators .EQ., .NE., .LT.	Same precedence
.NOT.	N/A
.AND.	N/A
.OR.	N/A
.NEQV. and .EQV.	N/A
Binary defined	Lowest

An expression is formed as:

```
expr binary-operator expr
```

or

```
unary-operator expr
```

where an expr is formed as

```
expression or element
```

For example,

```
A+B
-C
+D
```

These are simple expressions whose components are elements. Expressions fall into one of four classes: arithmetic, relational, logical or character.

Operators of equal rank are evaluated left to right. Thus:

```
A*B+B**C .EQ. X+Y/Z
.AND. .NOT. K-3.0 .GT. T
```

is equivalent to:

```
((((A*B)+(B**C)) .EQ.
(X+(Y/Z))) .AND. (.NOT. ((K-3.0) .GT. T)))
```

Arithmetic Expressions

Arithmetic expressions are formed from arithmetic elements and arithmetic operators. An arithmetic element may be:

- an arithmetic expression
- a variable
- a constant
- an array element
- a function reference
- a field of a structure

The arithmetic operators specify a computation to be performed on the elements. The result is a numeric result. Table 1-5 , “Arithmetic Operators”, shows the arithmetic operators.

Note that a value should be associated with a variable or array element before it is used in an expression. Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. The precedence of each arithmetic operator is shown in Table 1-5 , “Arithmetic Operators”.

This following example is resolved into the arithmetic expressions `(A) + (B * C)` rather than `(A + B) * (C)`.

```
A + B * C
```

Normal ranked precedence may be overcome using parentheses which force the item(s) enclosed to be evaluated first.

```
(A + B) * C
```

The compiler resolves this into the expressions `(A + B) * (C)`.

Table 1-5: Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
-	Subtraction or unary minus

Table 1-6: Arithmetic Operator Precedence

Operator	Precedence
**	First
* and /	Second
+ and -	Third

The type of an arithmetic expression is:

INTEGER if it contains only integer elements.

REAL if it contains only real and integer elements.

DOUBLE PRECISION	if it contains only double precision, real and integer elements.
COMPLEX	if any element is complex. Any element which needs conversion to complex will be converted by taking the real part from the original value and setting the imaginary part to zero.
DOUBLE COMPLEX	if any element is double complex.

Relational Expressions

A relational expression is composed of two arithmetic expressions separated by a relational operator. The value of the expression is true or false (`.TRUE.` or `.FALSE.`) depending on the value of the expressions and the nature of the operator. The table below shows the relational operators.

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

In relational expressions the arithmetic elements are evaluated to obtain their values. The relationship is then evaluated to obtain the true or false result. Thus the relational expression:

```
TIME + MEAN .LT. LAST
```

means if the sum of `TIME` and `MEAN` is less than the value of `LAST`, then the result is true, otherwise it is false.

Logical Expressions

A logical expression is composed of two relational or logical expressions separated by a logical operator. Each logical expression yields the value true or false (`.TRUE.` or `.FALSE.`) The following table shows the logical operators.

Operator	Relationship
.AND.	True if both expressions are true.
.OR.	True if either expression or both is true.
.NOT.	This is a unary operator; it is true if the expression is false, otherwise it is false.
.NEQV.	False if both expressions have the same logical value
.XOR.	Same as .NEQV.
.EQV.	True if both expressions have the same logical value

In the following example, TEST will be .TRUE. if A is greater than B or I is not equal to J+17.

```
TEST = A .GT. B .OR. I .NE. J+17
```

Character Expressions

An expression of type CHARACTER can consist of one or more printable characters. Its length is the number of characters in the string. Each character is numbered consecutively from left to right beginning with 1. For example:

```
'ab_&'
'A@HJi2'
'var[1,12]'
```

Character Concatenation

A character expression can be formed by concatenating two (or more) valid character expressions using the concatenation operator //. The following table shows several examples of concatenation.

Expression	Value
'ABC'//'YZ'	"ABCYZ"

Expression	Value
`'JOHN '//'SMITH'`	`"JOHN SMITH"`
`'J '//'JAMES '// 'JOY'`	`"J JAMES JOY"`

Symbolic Name Scope

Fortran 90/95 scoping is expanded from the traditional FORTRAN 77 capabilities which provide a scoping mechanism using subroutines, main programs, and COMMONs. Fortran 90/95 adds the MODULE statement. Modules provide an expanded alternative to the use of both COMMONs and INCLUDE statements. Modules allow data and functions to be packaged and defined as a unit, incorporating data hiding and using a scope that is determined with the USE statement.

Fortran 90/95 and HPF scoping is expanded from the traditional FORTRAN 77 capabilities which provide a scoping mechanism using subroutines, main programs, and COMMONs. Fortran 90/95 and HPF add the MODULE statement. Modules provide an expanded alternative to the use of both COMMONs and INCLUDE statements. Modules allow data and functions to be packaged and defined as a unit, incorporating data hiding and using a scope that is determined with the USE statement.

Names of COMMON blocks, SUBROUTINEs and FUNCTIONs are global to those modules that reference them. They must refer to unique objects, not only during compilation, but also in the link stage.

The scope of names other than these is local to the module in which they occur, and any reference to the name in a different module will imply a new local declaration. This includes the arithmetic function statement.

Assignment Statements

A Fortran assignment statement can be any of the following:

- An intrinsic assignment statement
- A statement label assignment
- An array assignment
- A masked array assignment

- A pointer assignment
- A defined assignment

Arithmetic Assignment

The arithmetic assignment statement has the following form:

```
object = arithmetic-expression
```

where *object* is one of the following:

- Variable
- Function name (within a function body)
- Subroutine argument
- Array element
- Field of a structure

The type of *object* determines the type of the assignment (INTEGER, REAL, DOUBLE PRECISION or COMPLEX) and the arithmetic-expression is coerced into the correct type if necessary.

In the case of:

```
complex = real expression
```

the implication is that the real part of the complex number becomes the result of the expression and the imaginary part becomes zero. The same applies if the expression is double precision, except that the expression will be coerced to real.

The following are examples of arithmetic assignment statements.

```
A=(P+Q)*(T/V)
B=R**T**2
```

Logical Assignment Statement

The logical assignment statement has the following form:

```
object = logical-expression
```

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- A field of a structure

The type of *objec*t must be logical.

In the following example, `FLAG` takes the logical value `.TRUE.` if `P+Q` is greater than `R`; otherwise `FLAG` has the logical value `.FALSE.`

```
FLAG=(P+Q) .GT. R
```

Character Assignment

The form of a character assignment is:

```
object = character
expression
```

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- Character substring
- A field of a structure

Above, *object* must be of type character.

None of the character positions being defined in object can be referenced in the character expression and only such characters as are necessary for the assignment to object need to be defined in the character expression. The character expression and `object` can have different lengths. When object is longer than the character expression trailing blanks are added to the object; and if object is shorter than the character expression the right-hand characters of the character expression are truncated as necessary.

In the following example, note that all the variables and arrays are assumed to be of type character.

```
FILE = 'BOOKS'
PLOT(3:8) = 'PLANTS'
TEXT(I,K+1)(2:B-1) = TITLE//X
```

Listing Controls

The PGI Fortran compilers recognize three compiler directives that affect the program listing process:

%LIST	Turns on the listing process beginning at the following source code line.
%NOLIST	Turns off the listing process (including the %NOLIST line itself).
%EJECT	Causes a new listing page to be started.

These directives have an effect only when the –Mlist option is used. All of the directives must begin in column one.

OpenMP Directives

OpenMP directives in a Fortran program provide information that allows the PGF77 and PGF95 compilers to generate executable programs that use multiple threads and processors on a shared-memory parallel (SMP) computer system. An OpenMP directive may have any of the following forms:

```
!$OMPdirective
C$OMPdirective
*$OMPdirective
```

A complete list and specifications of OpenMP directives supported by the PGF77 and PGF95 compilers, along with descriptions of the related OpenMP runtime library routines, can be found in Chapter 8, “OpenMP Directives for Fortran”.

HPF Directives

HPF directives in a Fortran program provide information that allows the PGHPF compiler to explicitly create data distributions from which parallelism can be derived. An HPF directive may have any of the following forms:

```
CHPF$directive
!HPF$directive
*HPF$directive
```

Since HPF supports two source forms, fixed source form and free source form, there are a variety of methods to enter a directive. The C, !, or * must be in column 1 for fixed source form directives. In free source form, Fortran limits the comment character to !. If you use the !HPF$ form for the directive origin, your code will be universally valid. The body of the directive may immediately follow the directive origin. Alternatively, any number of blanks may precede the HPF directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equals sign.

The directive name, including the directive origin, may contain upper or lower case letters (case is not significant). A complete list and specifications of HPF directives supported by the PGHPF compiler can be found in Chapter 9, "HPF Directives".

2 Fortran Data Types

Every Fortran element and expression has a data type. The data type of an element may be implicit in its definition or explicitly attached to the element in a declaration statement. This chapter describes the Fortran data types and constants that are supported by the PGI Fortran compilers.

Fortran provides two kinds of data types, intrinsic data types and derived data types. Types provided by the language are intrinsic types. Types specified by the programmer and built from the intrinsic data types are called derived types.

Intrinsic Data Types

Fortran provides six different intrinsic data types as shown in Table 2-1 , "Fortran Intrinsic Data Types". Table 2-2 , "Data Types Kind Parameters" and Table 2-3 , "Data Type Extensions" show variations and different "kinds" of the intrinsic data types supported by the PGI Fortran compilers.

Kind Parameter

The Fortran 95 KIND parameter specifies a precision for intrinsic data types. The KIND parameter follows a data type specifier and specifies size or type of the supported data type. A KIND specification overrides the length attribute that the statement implies and assigns a specific length to the item, regardless of the compiler's command-line options. A KIND is defined for a data type by a PARAMETER statement, using sizes supported on the particular system.

The following are some examples using a KIND specification:

```
INTEGER (SHORT) :: L
REAL (HIGH) B
REAL (KIND=HIGH) XVAR, YVAR
```

These examples require that the programmer use a PARAMETER statement to define kinds:

```
INTEGER, PARAMETER :: SHORT=1
INTEGER HIGH
PARAMETER (HIGH=8)
```

The following table shows several examples of KINDs that a system could support.

Table 2-1: Fortran Intrinsic Data Types

Data Type	Value
INTEGER	An integer number.
REAL	A real number.
DOUBLE PRECISION	A double precision floating point number, real number, taking up two numeric storage units and whose precision is greater than REAL.
LOGICAL	A value which can be either TRUE or FALSE.
COMPLEX	A pair of real numbers used in complex arithmetic. Fortran provides two precisions for COMPLEX numbers.
CHARACTER	A string consisting of one or more printable characters.

Table 2-2: Data Types Kind Parameters

Type	Kind	Size
INTEGER	SHORT	1 byte
INTEGER	LONG	4 bytes
REAL	HIGH	8 bytes

Number of Bytes Specification

The PGI Fortran compilers support a length specifier for some data types. The data type can be followed by a data type length specifier of the form *s, where s is one of the supported lengths for the data type. Such a specification overrides the length attribute that the statement implies and assigns a specific length to the specified item, regardless of the compiler options. For example, REAL*8 is equivalent to DOUBLE PRECISION. The following table shows the lengths of data types, their meanings, and their sizes.

Table 2-3: Data Type Extensions

Type	Meaning	Size
LOGICAL*1	Small LOGICAL	1 byte
LOGICAL*2	Short LOGICAL	2 bytes
LOGICAL*4	LOGICAL	4 bytes
LOGICAL*8	LOGICAL	8 bytes
BYTE	Small INTEGER	1 byte
INTEGER*1	Same as BYTE	1 byte
INTEGER*2	Short INTEGER	2 bytes
INTEGER*4	INTEGER	4 bytes
INTEGER*8	INTEGER	8 bytes
REAL*4	REAL	4 bytes
REAL*8	DOUBLE PRECISION	8 bytes
COMPLEX*8	COMPLEX	8 bytes
COMPLEX*16	DOUBLE COMPLEX	16 bytes

The BYTE type is treated as a signed one-byte integer and is equivalent to LOGICAL*1.

Assignment of a value too big for the data type to which it is assigned is an undefined operation.

A specifier is allowed after a CHARACTER function name even if the CHARACTER type word has a specifier.

For example:

```
CHARACTER*4 FUNCTION C*8 (VAR1)
```

The function size specification C*8 overrides the CHARACTER*4 specification. Logical data items can be used with any operation where a similar sized integer data item is permissible and vice versa. The logical data item is treated as an integer or the integer data item is treated as a logical of the same size and no type conversion is performed.

Floating point data items of type REAL or DOUBLE PRECISION may be used as array subscripts, in computed GOTOs, in array bounds and in alternate returns. The floating point data item is converted to an integer.

The data type of the result of an arithmetic expression corresponds to the type of its data. The type of an expression is determined by the rank of its elements. The following table shows the ranks of the various data types, from lowest to highest.

Table 2-4: Data Type Ranks

Data Type	Rank
LOGICAL	1 (lowest)
LOGICAL*8	2
INTEGER*2	3
INTEGER*4	4
INTEGER*8	5
REAL*4	6
REAL*8 (Double precision)	7
COMPLEX*8 (Complex)	8
COMPLEX*16 (Double complex)	9 (highest)

The data type of a value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. The exception to this rule is that an operation involving a COMPLEX*8 element and a REAL*8 element produces a COMPLEX*16 result. In this operation, the COMPLEX*8 element is converted to a COMPLEX*16 element, which consists of two REAL*8 elements, before the operation is performed.

In most cases, a logical expression will have a LOGICAL*4 result. In cases where the hardware supports LOGICAL*8 and if the expression is LOGICAL*8, the result may be LOGICAL*8.

Constants

A constant is an unchanging value that can be determined at compile time. It takes a form corresponding to one of the data types. The PGI Fortran compilers support decimal (INTEGER and REAL), unsigned binary, octal, hexadecimal, character and Hollerith constants.

The use of character constants in a numeric context, for example, in the right-hand side of an arithmetic assignment statement, is supported. These constants assume a data type that conforms to the context in which they appear.

Integer Constants

The form of a decimal integer constant is:

```
[s]d1d2...dn [ _ kind-parameter ]
```

where s is an optional sign and di is a digit in the range 0 to 9. The optional _kind-parameter is a Fortran 90/95 feature supported only by PGF95 and PGHPF, and specifies a supported kind. The value of an integer constant must be within the range for the specified kind.

The value of an integer constant must be within the range -2147483648 to 2147483647 inclusive (-231 to (231 - 1)). Integer constants assume a data type of INTEGER*4 and have a 32-bit storage requirement.

The –i8 compilation option causes all data of type INTEGER to be promoted to an 8 byte INTEGER. The –i8 option does not override an explicit data type extension size specifier (for example INTEGER*4). The range, data type and storage requirement change if the –i8 flag is specified (this flag is not supported on all targets). With the –i8 flag, the range for integer constants is -263 to (263 - 1)), and in this case the value of an integer constant must be within the range -9223372036854775808 to 9223372036854775807. If the constant does not fit in an INTEGER*4 range, the data type is INTEGER*8 and the storage requirement is 64 bits.

Below are several examples of integer constants.

```
+2
-36
437
-36_SHORT
369_I2
```

Binary, Octal and Hexadecimal Constants

The PGI compilers and Fortran 90/95 support various types of constants besides decimal constants. Fortran allows unsigned binary, octal, or hexadecimal constants in DATA statements. PGI compilers support these constants in DATA statements, and additionally, support some of these constants outside of DATA statements. For more information on support of these constants, refer to "Fortran Binary, Octal and Hexadecimal Constants" on page 33.

Real Constants

Real constants have two forms, scaled and unscaled. An unscaled real constant consists of a signed or unsigned decimal number (a number with a decimal point). A scaled real constant takes the same form as an unscaled constant, but is followed by an exponent scaling factor of the form:

```
E+digits [_ kind-parameter ]
Edigit [_ kind-parameter ]
E-digits [_ kind-parameter ]
```

where digits is the scaling factor, the power of ten, to be applied to the unscaled constant. The first two forms above are equivalent, that is, a scaling factor without a sign is assumed to be positive. The following table shows several real constants.

Table 2-5: Example of Real Constants

Constant	Value
1.0	unscaled single precision constant
1.	unscaled single precision constant
-.003	signed unscaled single precision constant
-.003_LOW	signed unscaled constant with kind LOW
-1.0	signed unscaled single precision constant
6.1E2_LOW	is equivalent to 610.0 with kind LOW
+2.3E3_HIGH	is equivalent to 2300.0 with kind HIGH
6.1E2	is equivalent to 610.0
+2.3E3	is equivalent to 2300.0
-3.5E-1	is equivalent to -0.35

Double Precision Constants

A double precision constant has the same form as a scaled REAL constant except that the E is replaced by D and the kind parameter is not permitted. For example:

```
D+digits
Ddigit
D-digits
```

The following table shows several double precision constants.

Table 2-6: Double Precision Constants

Constant	Value
6.1D2	is equivalent to 610.0
+2.3D3	is equivalent to 2300.0
-3.5D-1	is equivalent to -0.35
+4D4	is equivalent to 40000

Complex Constants

A complex constant is held as two real or integer constants separated by a comma and surrounded by parentheses. The first real number is the real part and the second real number is the imaginary part. Together these values represent a complex number. Integer values supplied as parameters for a COMPLEX constant are converted to REAL numbers. Below are several examples:

```
(18,-4)
(3.5,-3.5)
(6.1E2,+2.3E3)
```

Double Complex Constants

A complex constant is held as two double constants separated by a comma and surrounded by parentheses. The first double is the real part and the second double is the imaginary part. Together these values represent a complex number. Below is an example:

```
(6.1D2,+2.3D3)
```

Logical Constants

A logical constant is one of:

```
.TRUE. [ _ kind-parameter ]
.FALSE.[ _ kind-parameter ]
```

The logical constants .TRUE. and .FALSE. are by default defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise[1].

Below are several examples:

```
.TRUE.
.FALSE.
.TRUE._BIT
```

The abbreviations T and F can be used in place of .TRUE. and .FALSE. in data initialization statements and in NAMELIST input.

Character Constants

Character string constants may be delimited using either an apostrophe (') or a double quote ("). The apostrophe or double quote acts as a delimiter and is not part of the character constant. Use two apostrophes together to include an apostrophe as part of the expression. If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escape. Within character constants, blanks are significant. For further information on the use of the backslash character, refer to –Mbackslash in the PGI User's Guide.

A character constant is one of:

```
[ kind-parameter_ ] "[characters]"
[ kind-parameter_ ] '[characters]'
```

Below are several examples of character constants.

```
'abc'
'abc '
'ab''c'
"Test Word"
GREEK_"μ"
```

A zero length character constant is written as '' or "".

If a character constant is used in a numeric context, for example as the expression on the right side of an arithmetic assignment statement, it is treated as a Hollerith constant. The rules for typing and sizing character constants used in a numeric context are described in "Hollerith Constants" on page 35.

---

1. The option –Munixlogical defines a logical expression to be TRUE if its value is non-zero, and FALSE otherwise; also, the internal value of .TRUE. is set to one. This option is not available on all target systems.

PARAMETER Constants

The PARAMETER statement permits named constants to be defined. Refer to the description of the PARAMETER statement found in Chapter 3, "Fortran Statements", for more details on defining constants.

Derived Types

A derived type is a type made up of components whose type is either intrinsic or another derived type. The TYPE and END TYPE keywords define a derived type. For example, the following derived type declaration defines the type PERSON and the array CUSTOMER of type PERSON:

```
! Declare a structure to define a person derived type
TYPE PERSON
 INTEGER ID
 LOGICAL LIVING
 CHARACTER(LEN=20) FIRST, LAST, MIDDLE
 INTEGER AGE
END TYPE PERSON
TYPE (PERSON) CUSTOMER(10)
```

A derived type statement definition is called a derived-type statement (the statements between TYPE PERSON and END TYPE PERSON in the previous example. The definition of a variable of the new type is called a TYPE statement (CUSTOMER in the previous example); note the use of parentheses in the TYPE statement.

The % character accesses the components of a derived type. For example:

```
CUSTOMER(1)%ID = 11308
```

Arrays

Arrays in Fortran are not data types, but are data objects of intrinsic or derived type with special characteristics. A dimension statement provides a data type with one or more dimensions. There are several differences between Fortran 90/95 and traditional FORTRAN 77 arrays.

Note: Fortran 90/95 supports all FORTRAN 77 array semantics.

An array is a group of consecutive, contiguous storage locations associated with a symbolic name which is the array name. Each individual element of storage, called the array element, is referenced by the array name modified by a list of subscripts. Arrays are declared with type declaration statements, DIMENSION statements and COMMON statements; they are not defined by implicit reference. These declarations will introduce an array name and establish the number of dimensions and the bounds and size of each dimension. If a symbol, modified by a list of subscripts is not defined as an array, then it will be assumed to be a FUNCTION reference with an argument list.

Fortran 90/95 arrays are "objects" and operations and expressions involving arrays may apply to every element of the array in an unspecified order. For example, in the following code, where A and B are arrays of the same shape (conformable arrays), the following expression adds six to every element of B and assigns the results to the corresponding elements of A:

```
A = B + 6
```

Fortran arrays may be passed with unspecified shapes to subroutines and functions, and sections of arrays may be used and passed as well. Arrays of derived type are also valid. In addition, allocatable arrays may be created with deferred shapes (number of dimensions is specified at declaration, but the actual bounds and size of each dimension are determined when the array is allocated while the program is running).

An Array Declaration Element

An array declaration has the following form:

```
name([lb:]ub[,[lb:]ub]...)
```

where name is the symbolic name of the array, lb is the specification of the lower bound of the dimension and ub is the specification of the upper bound. The upper bound, ub must be greater than the lower bound lb. The values lb and ub may be negative. The bound lb is taken to be 1 if it is not specified. The difference (ub-lb+1) specifies the number of elements in that dimension. The number of lb,ub pairs specifies the rank of the array. Assuming the array is of a data type that requires N bytes per element, the total amount of storage of the array is:

```
N*(ub-lb+1)*(ub-lb+1)*...
```

The dimension specifiers of an array subroutine argument may themselves be subroutine arguments or members of COMMON.

Deferred Shape Arrays

Deferred-shape arrays are those arrays whose shape can be changed by an executable statement. Deferred-shape arrays are declared with a rank, but with no bounds information. They assume their shape when either an ALLOCATE statement or a REDIMENSION statement is encountered.

For example, the following statement declares a deferred shape REAL array A of rank two:

```
REAL A(:, :)
```

Subscripts

A subscript is used to specify an array element for access. An array name qualified by a subscript list has the following form:

```
name(sub[,sub]...)
```

where there must be one sub entry for each dimension in array name.

Each sub must be an integer expression yielding a value which is within the range of the lower and upper bounds. Arrays are stored as a linear sequence of values in memory and are held such that the first element is in the first store location and the last element is in the last store location. In a multi-dimensional array the first subscript varies more rapidly than the second, the second more rapidly than the third, and so on (column major order).

Character Substring

A character substring is a contiguous portion of a character variable and is of type character. A character substring can be referenced, assigned values and named. It can take either of the following forms:

```
character_variable_name(x1:x2)
character_array_name(subscripts)(x1:x2)
```

where x1 and x2 are integers and x1 denotes the left-hand character position and x2 the right-hand one. These are known as substring expressions. In substring expressions x1 must be both greater than or equal to 1 and less than x2 and x2 must be less than or equal to the length of the character variable or array element.

For example:

```
J(2:4)
```

the characters in positions 2 to 4 of character variable `J`.

```
K(3,5)(1:4)
```

the characters in positions 1 to 4 of array element `K(3,5)`.

A substring expression can be any valid integer expression and may contain array elements or function references.

Fortran Pointers and Targets

Fortran pointers are similar to allocatable arrays. Pointers are declared with a type and a rank; they do not actually represent a value, however, but represent a value's address. Fortran 90/95 has a specific assignment operator, =>, for use in pointer assignments.

Fortran Binary, Octal and Hexadecimal Constants

The PGI Fortran compilers support two representations for binary, octal, and hexadecimal numbers: the standard Fortran 90/95 representation and the PGI extension representation. Refer to the next section for details on the alternate representation.

Fortran supports binary, octal and hexadecimal constants in DATA statements. The form of a binary constant is:

```
B'b1b2...bn'
B"b1b2...bn"
```

where b i is either 0 or 1.

The form of an octal constant is:

```
O'c1c2...cn'
O"c1c2...cn"
```

where c i is in the range 0 through 7.

The form of a hexadecimal constant is:

```
Z'a1a2...an'
Z"a1a2...an"
```

or

```
'a1a2...an'X
"a1a2...an"X
```

where a is in the range 0 through 9 or a letter in the range A through F or a through f (case mixing is allowed).

Octal and Hexadecimal Constants - Alternate Form §

The PGF95 compiler supports additional extensions.The PGF95 and PGHPF compilers support additional extensions. This is an alternate form for octal constants, outside of DATA statements. The form for an octal constant is:

```
'c1c2...cn'O
```

The form of a hexadecimal constant is:

```
'a1a2...an'X
```

where ci is a digit in the range 0 to 7 and ai is a digit in the range 0 to 9 or a letter in the range A to F or a to f (case mixing is allowed). Up to 64 bits (22 octal digits or 16 hexadecimal digits) can be specified.

Octal and hexadecimal constants are stored as either 32-bit or 64-bit quantities. They are padded on the left with zeroes if needed and assume data types based on how they are used.

The following are the rules for converting these data types:

- A constant is always either 32 or 64 bits in size and is typeless. Sign-extension and type-conversion are never performed. All binary operations are performed on 32-bit or 64-bit quantities. This implies that the rules to follow are only concerned with mixing 32-bit and 64-bit data.
- When a constant is used with an arithmetic binary operator (including the assignment operator) and the other operand is typed, the constant assumes the type and size of the other operand.
- When a constant is used in a relational expression such as .EQ., its size is chosen from the operand having the largest size. This implies that 64-bit comparisons are possible.
- When a constant is used as an argument to the generic AND, OR, EQV, NEQV, SHIFT, or COMPL function, a 32-bit operation is performed if no argument is more than 32 bits in size; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument in any other context, no data type is assumed; however, a length of four bytes is always used. If necessary, truncation on the left occurs.

- When a specific 32-bit or 64-bit data type is required, that type is assumed for the constant. Array subscripting is an example.
- When a constant is used in a context other than those mentioned above, an INTEGER*4 data type is assumed. Logical expressions and binary arithmetic operations with other untyped constants are examples.
- When the required data type for a constant implies that the length needed is more than the number of digits specified, the leftmost digits have a value of zero. When the required data type for a constant implies that the length needed is less than the number of digits specified, the constant is truncated on the left. Truncation of nonzero digits is allowed.

In the example below, the constant I (of type INTEGER*4) and the constant J (of type INTEGER*2) will have hex values 1234 and 4567, respectively. The variable D (of type REAL*8) will have the hex value x4000012345678954 after its second assignment:

```
I = '1234'X ! Leftmost Pad with zero
J = '1234567'X ! Truncate Leftmost 3 hex digits
D = '40000123456789ab'X
D = NEQV(D,'ff'X) ! 64-bit Exclusive Or
```

Hollerith Constants

The form of a Hollerith constant is:

```
nHc1c2...cn
```

where *n* specifies the positive number of characters in the constant and cannot exceed 2000 characters. A Hollerith constant is stored as a byte string with four characters per 32-bit word. Hollerith constants are untyped arrays of INTEGER*4. The last word of the array is padded on the right with blanks if necessary. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected. The data type of a Hollerith constant used in a numeric expression is determined by the following rules:

- Sign-extension is never performed.
- The byte size of the Hollerith constant is determined by its context and is not strictly limited to 32 or 64 bits like hexadecimal and octal constants.
- When the constant is used with a binary operator (including the assignment operator), the data type of the constant assumes the data type of the other operand.

- When a specific data type is required, that type is assumed for the constant. When an integer or logical is required, INTEGER*4 and LOGICAL*4 are assumed. When a float is required, REAL*4 is assumed (array subscripting is an example of the use of a required data type).
- When a constant is used as an argument to certain generic functions (AND, OR, EQV, NEQV, SHIFT, and COMPL), a 32-bit operation is performed if no argument is larger than 32 bits; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument, no data type is assumed and the argument is passed as an INTEGER*4 array. Character constants are passed by descriptor only.
- When a constant is used in any other context, a 32-bit INTEGER*4 array type is assumed.

When the length of the Hollerith constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right.

Structures

A structure, a DEC extension to FORTRAN 77, is a user-defined aggregate data type having the following form:

```
STRUCTURE [/structure_name/][field_namelist]
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END STRUCTURE
```

Where:

structure_name	is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.
field_namelist	is a list of fields having the structure of the associated structure declaration. A field_namelist is allowed only in nested structure declarations.
field_declaration	can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware; therefore a structure's storage requirements are machine-dependent. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

Data initialization can occur for the individual fields.

Records

A record, a DEC extension to FORTRAN 77, is a user-defined aggregate data item having the following form:

```
RECORD /structure_name/record_namelist
 [,/structure_name/record_namelist]
 ...
 [,/structure_name/record_namelist]
```

where:

structure_name	is the name of a previously declared structure.
record_namelist	is a list of one or more variable or array names separated by commas.

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, recordname.fieldname). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

The following is an example of RECORD and STRUCTURE usage.

```
STRUCTURE /person/ ! Declare a structure
defining a person
 INTEGER id
 LOGICAL living
 CHARACTER*5 first, last, middle
 INTEGER age
END STRUCTURE
 ! Define population to be an array where each element is
 ! of type person. Also define a variable, me, of type
 ! person.
RECORD /person/ population(2), me
 ...
me.age = 34 ! Assign values for the variable me
to
me.living = .TRUE. ! some of the fields.
me.first = 'Steve'
me.id = 542124822
 ...
population(1).last = 'Jones' ! Assign the "last" field
of
 ! element 1 of array population.
population(2) = me ! Assign all values of record
 ! "me" to the record
 ! population(2)
```

UNION and MAP Declarations

A UNION declaration, a DEC extension to FORTRAN 77, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a MAP declaration, with one or more fields per MAP declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration.

The format of a UNION statement is described in the following example:

```
UNION
 map_declaration
 [map_declaration]
 ...
 [map_declaration]
END UNION
```

The format of the map_declaration is as follows:

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

where field_declaration is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

Field alignment within multiple map declarations is performed as previously defined in structure declarations.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the C language requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following is an example of RECORD, STRUCTURE, MAP and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP, the employee map (24 bytes).

```
STRUCTURE /account/
 INTEGER typetag ! Tag to determine defined map.
 UNION
 MAP ! Structure for an employee
 CHARACTER*12 ssn ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12) ! Items supplied
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

Data Initialization

Data initialization is allowed within data type declaration statements. This is an extension to the Fortran language. Data is initialized by placing values bounded by slashes immediately following the symbolic name (variable or array) to be initialized. Initialization of fields within structure declarations is allowed, but initialization of unnamed fields and records is not.

Hollerith, octal and hexadecimal constants can be used to initialize data in both data type declarations and in DATA statements. Truncation and padding occur for constants that differ in size from the declared data item (as specified in the discussion of constants).

Pointer Variables

The POINTER statement, a CRAY extension to FORTRAN 77 which is distinct from the Fortran 90/95 POINTER specification statement or attribute, declares a scalar variable to be a pointer variable (of data type INTEGER), and another variable to be its pointer-based variable.

The syntax of the POINTER statement is:

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and v2	are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.
p1 and p2	are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined.

The pointer is defined in one of the following ways:

- By assigning the value of the LOC function.
- By assigning a value defined in terms of another pointer variable.
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

The following code illustrates the use of pointers:

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0 ! IC gets 0
P = LOC(XC)
Q = P + 20 ! same as LOC(XC(6))
X(1) = 0 ! XC(6) gets 0
ALLOCATE (X) ! Q locates an allocated memory area
```

Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarators are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is assumed not to overlap with another pointer-based variable.

3 Fortran Statements

This chapter describes each of the Fortran statements supported by the PGI Fortran compilers. Each description includes a brief summary of the statement, a syntax description, a complete description and an example. The statements are listed in alphabetical order. The first section lists terms that are used throughout the chapter.

Definition of Terms

character scalar memory reference
: is a character variable, a character array element, or a character member of a structure.

integer scalar memory reference
: is an integer variable, an integer array element, or an integer member of a structure.

logical scalar memory reference
: is a logical variable, a logical array element, or a logical member of a structure.

obsolescent
: The statement is unchanged from the FORTRAN 77 definition but has a better replacement in Fortran 95.

Origin of Statement

At the top of each reference page is an indication of the origin of the statement.

HeadingExplanation

77
: FORTRAN 77 statements that are essentially unchanged from the original FORTRAN 77 standard and are supported by the PGF77 compiler.

90/95
: This statement is either new for Fortran 90/95 or significantly changed in Fortran 95 from its original FORTRAN 77 definition and is supported by the PGF95 and PGHPF compilers.

HPF
: The statement has its origin in the HPF standard.

§
: The statement is an extension to the Fortran language.

CMF
: Indicates a CM Fortran feature (CM Fortran is a version of Fortran that was produced by Thinking Machines Corporation for parallel computers).

Statements

ACCEPT § 77

The ACCEPT statement has the same syntax as the PRINT statement and causes formatted input to be read on standard input. ACCEPT is identical to the READ statement with a unit specifier of asterisk (*).

Syntax

```
ACCEPT f [,iolist]
ACCEPT namelist
```

f	format-specifier, a * indicates list directed input.
iolist	is a list of variables to be input.
namelist	is the name of a namelist specified with the NAMELIST statement.

Examples

```
 ACCEPT *, IA, ZA
 ACCEPT 99, I, J, K
 ACCEPT SUM
99 FORMAT(I2, I4, I3)
```

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
accept ch
```

See Also

READ, PRINT

ALLOCATABLE 90

The ALLOCATABLE specification statement (attribute) specifies that an array with fixed rank but deferred shape is available for a future ALLOCATE statement. An ALLOCATE statement allocates space for the allocatable array.

Syntax

```
ALLOCATABLE [ :: ] array-name [(deferred-array-spec)]
 [, array-name [(deferred-array-spec)]]...
```

array-name	is the name of the allocatable array.
deferred-array-spec	is a : character.

Example

```
REAL SCORE(:), NAMES(:,:)
REAL, ALLOCATABLE, DIMENSION(:,:,:) :: TEST
ALLOCATABLE SCORE, NAMES
INTEGER, ALLOCATABLE :: REC1(: ,: , :)
```

See Also

ALLOCATE, DEALLOCATE

ALLOCATE 90

The ALLOCATE statement is an extension to FORTRAN 77 but is part of the Fortran 90/95 standard. It allocates storage for each pointer-based variable and allocatable array which appears in the statement. ALLOCATE also declares storage for deferred-shape arrays.

Syntax

```
ALLOCATE ( allocation-list [ , STAT= var ] )
```

allocation-list is:

```
allocate-object [ allocate-shape-spec-list ]
```

allocate-object is:

```
variable-name
structure-component
```

allocate-shape-spec-list is:

```
[ allocate-lower-bound : ] allocate-upper-bound
```

var is an integer variable, integer array element or an integer member of a STRUCTURE (that is, an integer scalar memory reference). This variable is assigned a value depending on the success of the ALLOCATE statement.

name is a pointer-based variable or name of an allocatable COMMON enclosed in slashes.

Description

For a pointer-based variable, its associated pointer variable is defined with the address of the allocated memory area. If the specifier STAT= is present, successful execution of the ALLOCATE statement causes the status variable to be defined with a value of zero. If an error occurs during execution of the statement and the specifier STAT= is present, the status variable is defined to have the integer value one. If an error occurs and the specifier STAT= is not present, program execution is terminated.

A dynamic, or allocatable COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. Note: Allocatable COMMON blocks are an extension to FORTRAN 77 supported only by PGF77 compiler, and not by the PGF95 or PGHPF compilers.

For an ALLOCATABLE array, the array is allocated with the executable ALLOCATE statement.

Examples

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

For a deferred shape array, the allocate must include the bounds of the array.

```
REAL, ALLOCATABLE :: A(:), B(:)
ALLOCATE (A(10), B(SIZE(A)))
REAL A(:,:)
N=3
M=1
```

```
ALLOCATE (A(1:11, M:N))
INTEGER FLAG, N
REAL, ALLOCATABLE:: B(:,:)
ALLOCATE (B(N,N),STAT=FLAG)
```

ARRAY CMF

The ARRAY attribute defines the number of dimensions in an array that may be defined and the number of elements and bounds in each dimension.

Syntax

```
ARRAY [::] array-name (array-spec)
 [, array-name (array-spec) ] ...
```

array-name	is the symbolic name of an array.
array-spec	is a valid array specification, either explicit-shape, assumed-shape, deferred-shape, or assumed size (refer to Chapter 4, "Fortran Arrays", for details on array specifications).

Description

ARRAY can be used in a subroutine as a synonym for DIMENSION to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays). Note however that these integer variables must be either arguments or declared in COMMON; they cannot be local. Note that in this case the function of ARRAY is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

The typing of the array in an ARRAY statement is defined by the initial letter of the array name in the same way as variable names, unless overridden by an IMPLICIT or type declaration statement. Arrays may appear in type declaration and COMMON statements but the array name can appear in only one array declaration.

Example

```
REAL, ARRAY(3:10):: ARRAY_ONE
INTEGER, ARRAY(3,-2:2):: ARRAY_TWO
```

This specifies ARRAY_ONE as a vector having eight elements with the lower bound of 3 and the upper bound of 10.

ARRAY_TWO as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

ASSIGN 77

(Obsolescent) The assign statement assigns a statement label to a variable. Internal procedures can be used in place of the assign statement. Other cases where the assign statement is used can be replaced by using character strings (for different format statements that were formally assigned labels by using an integer variable as a format specifier).

Syntax

```
ASSIGN a TO b
```

a is the statement label.

b is an integer variable.

Description

Executing an ASSIGN statement assigns a statement label to an integer variable. This is the only way that a variable may be defined with a statement label value. The statement label must be:

- A statement label in the same program unit as the ASSIGN statement.
- The label of an executable statement or a FORMAT statement.

A variable must be defined with a statement label when it is referenced:

- In an assigned GOTO statement.
- As a format identifier in an input/output statement and while so defined must not be referenced in any other way.

An integer variable defined with a statement label can be redefined with a different statement label, the same statement label or with an integer value.

Example

```
 ASSIGN 40 TO K
 GO TO K
40 L = P + I + 56
```

BACKSPACE 77

When a BACKSPACE statement is executed the file connected to the specified unit is positioned before the preceding record.

Syntax

```
BACKSPACE unit
BACKSPACE ([UNIT=]unit [,ERR=errs] [,
IOSTAT=ios])
```

UNIT=unit	unit is the unit specifier.
ERR=s	s is an executable statement label for the statement used for processing an error condition.
IOSTAT=ios	ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

Description

If there is no preceding record, the position of the file is not changed. A BACKSPACE statement cannot be executed on a file that does not exist. Do not issue a BACKSPACE statement for a file that is open for direct or append access.

Examples

```
BACKSPACE 4
BACKSPACE ( UNIT=3 )
BACKSPACE ( 7, IOSTAT=IOCHEK, ERR=50 )
```

BLOCK DATA 77

The BLOCK DATA statement introduces a number of statements that initialize data values in COMMON blocks. No executable statements are allowed in a BLOCK DATA segment.

Syntax

```
BLOCK DATA [name]
 [specification]
END [BLOCK DATA [name]]
```

name	is a symbol identifying the name of the block data and must be unique among all global names (COMMON block names, program name, module names). If missing, the block data is given a default name.

Example

```
 BLOCK DATA
 COMMON /SIDE/ BASE, ANGLE, HEIGHT, WIDTH
 INTEGER SIZE
 PARAMETER (SIZE=100)
 INTEGER BASE(0:SIZE)
 REAL WIDTH(0:SIZE), ANGLE(0:SIZE)
 DATA (BASE(I),I=0,SIZE)/SIZE*-1,-1/,
+ (WIDTH(I),I=0,SIZE)/SIZE*0.0,0.0/
 END
```

BYTE § 77

The BYTE statement establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

```
BYTE name [/clist/],
...
```

name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Byte statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. BYTE declaration statements must not be labeled.

Example

```
BYTE TB3, SEC, STORE (5,5)
```

CALL 77

The CALL statement transfers control to a subroutine.

Syntax

```
CALL subroutine [([ actual-arg-list]...])]
```

subroutine	is the name of the subroutine.
argument	is the actual argument being passed to the subroutine. The first argument corresponds to the first dummy argument in the SUBROUTINE statement and so on.
actual-arg-list	has the form:

[keyword =]	subroutine-argument.
keyword	is a dummy argument name in the subroutine interface.
subroutine-argument	is an actual argument.

Description

Actual arguments can be expressions including: constants, scalar variables, function references and arrays.

Actual arguments can also be alternate return specifiers. Alternate return specifiers are labels prefixed by asterisks (*) or ampersands (&). The ampersand is an extension to FORTRAN 77.

Recursive calls are allowed using the –Mrecursive command-line option.

Examples

```
CALL CRASH ! no arguments
CALL BANG(1.0) ! one argument
CALL WALLOP(V, INT) ! two arguments
CALL ALTRET(I, *10, *20)
SUBROUTINE ONE
DIMENSION ARR ( 10, 10 )
REAL WORK
```

```
INTEGER ROW, COL
PI=3.142857
CALL EXPENS(ARR,ROW,COL,WORK,SIN(PI/2)+3.4)
RETURN
END
```

CASE 90

The CASE statement begins a case-statement-block portion of a SELECT CASE construct.

Syntax

```
[ case-name : ] SELECT CASE (case-expr)
[ CASE (selector) [name]
 block ] ...
[ CASE DEFAULT [case-name]
 block ]
END SELECT [case-name]
```

Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
 TYPE=1
CASE ( 4:6 )
 TYPE=2
CASE DEFAULT
 TYPE=0
END SELECT
```

Type

Executable

See Also

SELECT CASE

CHARACTER 90

The CHARACTER statement establishes the data type of a variable by explicitly attaching the name of a variable to a character data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for CHARACTER has two forms, the standard Fortran form and the PGI extended form. This section describes both syntax forms.

```
CHARACTER [character-selector] [,
attribute-list ::] entity-list
```

character-selector	the character selector specifies the length of the character string. This has one of several forms:

```
([LEN=] type-param-value)
* character-length [,]
```

Character-selector also permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list	is the list of attributes for the character variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
CHARACTER [*len][,] name [dimension] [*len] [/clist/],
...
```

len	is a constant or *. A * is only valid if the corresponding name is a dummy argument.
name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Character type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled.

Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Examples

```
CHARACTER A*4, B*6, C
CHARACTER (LEN=10):: NAME
```

A is 4 and B is 6 characters long and C is 1 character long. NAME is 10 characters long.

CLOSE 77

The CLOSE statement terminates the connection of the specified file to a unit.

Syntax

```
CLOSE ([UNIT=] u [,ERR= errs ] [,DISP[OSE]= sta]
 [,IOSTAT=ios] [,STATUS= sta] )
```

u is the external unit specifier where u is an integer.

errs is an error specifier in the form of a statement label of an executable statement in the same program unit. If an error condition occurs, execution continues with the statement specified by errs.

ios is an integer scalar; if this is included ios becomes defined with 0 (zero) if no error condition exists or a positive integer when there is an error condition. A value of -1 indicates an end-of-file condition with no error. A value of -2 indicates an end-of-record condition with no error when using non-advancing I/O.

sta is a character expression, where case is insignificant, specifying the file status and the same keywords are used for the dispose status. Status can be 'KEEP' or 'DELETE' (the quotes are required). KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement; conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

Description

A unit may be the subject of a CLOSE statement from within any program unit. If the unit specified does not exist or has no file connected to it the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

Example

In the following example, the file on UNIT 6 is closed and deleted.

```
CLOSE(UNIT=6,STATUS='DELETE')
```

COMMON 77

The COMMON statement defines global blocks of storage that are either sequential or non sequential. There are two forms of the COMMON statement, a static form and a dynamic form. Each common block is identified by the symbolic name defined in the COMMON block.

Syntax

```
COMMON /[name ] /nlist [, /name/nlist]...
```

name	is the name of each common block and is declared between the /.../ delimiters for a named common and with no name for a blank common.
nlist	is a list of variable names where arrays may be defined in DIMENSION statements or formally declared by their inclusion in the COMMON block.

Common Block Rules and Behaviors

A common block is a global entity. Any common block name (or blank common) can appear more than once in one or more COMMON statements in a program unit. The following is a list of rules associated with common blocks:

Blank Common	The name of the COMMON block need not be supplied; without a name, the common is a BLANK COMMON. In this case the compiler uses a default name.
Same Names	There can be several COMMON block statements of the same name in a program segment; these are effectively treated as one statement, with variables concatenated from one COMMON statement of the same name to the next. This is an alternative to the use of continuation lines when declaring a common block with many symbols. Common blocks with the same name that are declared in different program units share the same storage area when combined into one executable program and they are defined using the SEQUENCE attribute.

HPF	In HPF, a common block is non-sequential by default, unless there is an explicit SEQUENCE directive that specifies the array as sequential. Note this may require that older FORTRAN 77 programs assuming sequence association in COMMON statements have SEQUENCE statements for COMMON variables.

Example

```
DIMENSION R(10)
COMMON /HOST/ A, R, Q(3), U
```

This declares a common block called HOST

Note

The different types of declaration used for R (declared in a DIMENSION statement) and Q (declared in the COMMON statement).

The declaration of HOST in a SUBROUTINE in the same executable program, with a different shape for its elements would require that the array be declared using the SEQUENCE attribute.

```
 SUBROUTINE DEMO
!HPF$ SEQUENCE HOST
 COMMON/HOST/STORE(15)
 .
 .
 .
 RETURN
 END
```

Common Blocks and Subroutines

If the main program has the common block declaration as in the previous example, the COMMON statement in the subroutine causes STORE(1) to correspond to A, STORE(2) to correspond to R(1), STORE(3) to correspond to R(2), and so on through to STORE(15) corresponding to the variable U.

Common Block Records and Characters

You can name records within a COMMON block. Because the storage requirements of records are machine-dependent, the size of a COMMON block containing records may vary between machines. Note that this may also affect subsequent equivalence associations to variables within COMMON blocks that contain records.

Both character and non-character data may reside in one COMMON block. Data is aligned within the COMMON block in order to conform to machine-dependent alignment requirements.

Blank COMMON is always saved. Blank COMMON may be data initialized.

See Also

The SEQUENCE directive.

Syntax Extension – dynamic COMMON§

A dynamic, or allocatable, COMMON block is a common block whose storage is not allocated until an explicit ALLOCATE statement is executed. PGF77 supports dynamic COMMON blocks, while PGF95 does not.and PGHPF do not.

If the ALLOCATABLE attribute is present, all named COMMON blocks appearing in the COMMON statement are marked as allocatable. Like a normal COMMON statement, the name of an allocatable COMMON block may appear in more than one COMMON statement. Note that the ALLOCATABLE attribute need not appear in every COMMON statement.

The following restrictions apply to the dynamic COMMON statement:

- Before members of an allocatable COMMON block can be referenced, the common block must have been explicitly allocated using the ALLOCATE statement.
- The data in an allocatable common block cannot be initialized.
- The memory used for an allocatable common block may be freed using the DEALLOCATE statement.
- If a SUBPROGRAM declares a COMMON block to be allocatable, all other subprograms containing COMMON statements of the same COMMON block must also declare the COMMON to be allocatable.

Example (dynamic COMMON)

```
COMMON, ALLOCATABLE /ALL1/ A, B, /ALL2/ AA,
BB
COMMON /STAT/ D, /ALL1/ C
```

This declares the following variables:

ALL1 is an allocatable COMMON block whose members are A, B, and C.

ALL2 is an allocatable COMMON block whose members are AA, and BB.

STAT is a statically-allocated COMMON block whose only member is D.

A reference to a member of an allocatable COMMON block appears in a Fortran statement just like a member of a normal (static) COMMON block. No special syntax is required to access members of allocatable common blocks. For example, using the above declarations, the following is a valid pgf77 statement:

```
AA = B * D
```

COMPLEX 90

The COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for COMPLEX has two forms, the standard Fortran form and the PGI extended form. This section describes both syntax forms.

```
COMPLEX [ ( [ KIND =] kind-value
) ] [, attribute-list ::] entity-list
```

COMPLEX permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list is the list of attributes for the character variable.

entity-list is the list of defined entities.

Syntax Extension§

```
COMPLEX name [*n] [dimensions] [/clist/] [,
name] [/clist/] ...
```

name is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement below for an explanation of array declarators).

clist is a list of constants that initialize the data, as in a DATA statement.

Description

COMPLEX statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. COMPLEX statements must not be labeled.

Note

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
COMPLEX CURRENT
COMPLEX DIMENSION(8) :: CONV1, FLUX1
```

The default size of a COMPLEX variable is 8 bytes. With the -r8 option, the default size of a COMPLEX variable is 16 bytes.

CONTAINS 90

The CONTAINS statement precedes a subprogram, a function or subroutine, that is defined inside a main program, external subprogram, or module subprogram (internal subprogram). The CONTAINS statement is a flag indicating the presence of a subroutine or function definition. An internal subprogram defines a scope for the internal subprogram's labels and names. Scoping is defined by use and host scoping rules within scoping units. Scoping units have the following precedence for names:

- A derived-type definition.
- A procedure interface body.
- A program unit or a subprogram, excluding contained subprograms.

Syntax

```
SUBROUTINE X
 INTEGER H, I
 .
 .
 .
 CONTAINS
 SUBROUTINE Y
```

```
  INTEGER I
  I = I + H
  .
  .
  END SUBROUTINE Y
END SUBROUTINE X
```

Type

Non-executable

See Also

MODULE

CONTINUE 77

The CONTINUE statement passes control to the next statement. It is supplied mainly to overcome the problem that transfer of control statements are not allowed to terminate a DO loop.

Syntax

```
CONTINUE
```

Example

```
  DO 100 I = 1,10
  SUM = SUM + ARRAY (I)
  IF(SUM .GE. 1000.0) GOTO 200
100 CONTINUE
200 ...
```

CYCLE 90

The CYCLE statement interrupts a DO construct execution and continues with the next iteration of the loop.

Syntax

```
CYCLE [do-construct-name]
```

Example

```
DO
 IF (A(I).EQ.0) CYCLE
 B=100/A(I)
 IF (B.EQ.5) EXIT
END DO
```

See Also

EXIT, DO

DATA 77

The DATA statement assigns initial values to variables before execution.

Syntax

```
DATA vlist/dlist/[[, ]vlist/dlist/]...
```

vlist is a list of variable names, array element names or array names separated by commas.

dlist is a list of constants or PARAMETER constants, separated by commas, corresponding to elements in the vlist. An array name in the vlist demands that dlist constants be supplied to fill every element of the array.

Repetition of a constant is provided by using the form:

```
n*constant-value
```

n a positive integer, is the repetition count.

Example

```
REAL A, B, C(3), D(2)
DATA A, B, C(1), D /1.0, 2.0, 3.0, 2*4.0/
```

This performs the following initialization:

```
A = 1.0
B = 2.0
C(1) = 3.0
D(1) = 4.0
D(2) = 4.0
```

DEALLOCATE 77

The DEALLOCATE statement causes the memory allocated for each pointer-based variable or allocatable array that appears in the statement to be deallocated (freed). Deallocate also deallocates storage for deferred-shape arrays.

Syntax

```
DEALLOCATE ( allocate-object-list [ , STAT= var ] )
```

Where:

allocate-object-list	is a variable name or a structure component.
al	is a pointer-based variable or the name of an allocatable COMMON block enclosed in slashes.
var	is the status indicator, an integer variable, integer array element or an integer member of a structure.

Description

An attempt to deallocate a pointer-based variable or an allocatable COMMON block which was not created by an ALLOCATE statement results in an error condition.

If the specifier STAT= is present, successful execution of the statement causes var to be defined with the value of zero. If an error occurs during the execution of the statement and the specifier STAT= is present, the status variable is defined to have the integer value one. If an error occurs and the specifier STAT= is not present, program execution is terminated.

Examples

```
REAL, ALLOCATABLE :: X(:,:)
ALLOCATE (X(10,2))
X=0
DEALLOCATE (X)
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
```

```
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

DECODE § 77

The DECODE statement transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential READ statements.

Syntax

```
DECODE (c, f, b [ ,IOSTAT= ios ] [,
ERR= errs]) [ list ]
```

c	is an integer expression specifying the number of bytes involved in translation.
f	is the format-specifier.
b	is a scalar or array reference for the buffer area containing formatted data (characters).
ios	is an integer scalar memory reference which is the input/output status specifier: if this is specified ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.
errs	an error specifier which takes the form of a statement label of an executable statement in the same program unit. If an error condition occurs execution continues with the statement specified by errs.
list	is a list of input items.

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
accept ch
```

See Also

READ, PRINT

DIMENSION 90

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

Syntax

```
DIMENSION [::] array-name (array-spec)
 [, array-name (array-spec) ] ...
DIMENSION array-name ([lb:]ub[,[lb:]ub]...)
 [,name([lb:]ub[,[lb:]ub]...)]
```

array-name	is the symbolic name of an array.
array-spec	is a valid array specification, either explicit-shape, assumed-shape, deferred-shape, or assumed size (refer for details on array specifications).
lb:ub	is a dimension declarator specifying the bounds for a dimension (the lower bound lb and the upper bound ub). lb and ub must be integers with ub greater than lb. The lower bound lb is optional; if it is not specified, it is taken to be 1.

Description

DIMENSION can be used in a subroutine to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays). Note however that these integer variables must be either arguments or declared in COMMON; they cannot be local. In this case the function of DIMENSION is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

The typing of the array in a DIMENSION statement is defined by the initial letter of the array name in the same way as variable names. The letters I,J,K,L,M and N imply that the array is of INTEGER type and an array with a name starting with any of the letters A to H and O to Z will be of type REAL, unless overridden by an IMPLICIT or type declaration statement. Arrays may appear in type declaration and COMMON statements but the array name can appear in only one array declaration.

DIMENSION statements must not be labeled.

Examples

```
DIMENSION ARRAY1(3:10), ARRAY2(3,-2:2)
```

This specifies ARRAY1 as a vector having eight elements with the lower bound of 3 and the upper bound of 10.

ARRAY2 as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

```
CHARACTER B(0:20)*4
```

This example sets up an array B with 21 character elements each having a length of four characters. Note that the character array has been dimensioned in a type declaration statement and therefore cannot subsequently appear in a DIMENSION statement.

DO (Iterative) 90

The DO statement introduces an iterative loop and specifies the loop control index and parameters. There are two forms of DO statement, block and non-block (FORTRAN 77 style). There are two forms of block do statements, DO iterative and DO WHILE. Refer to the description of DO WHILE for more details on this form of DO statement.

Syntax

DO (block)

```
[do-construct-name : ] DO [label ] [loop-control]
 [execution-part-construct]
[label] END DO
```

loop-control	is an increment index expression of the form:[index = e1 e2 [, e3]]
label	labels the last executable statement in the loop (this must not be a transfer of control).

index	is the name of a variable called the DO variable.
e1	is an expression which yields an initial value for i.
e2	is an expression which yields a final value for i.
e3	is an optional expression yielding a value specifying the increment value for i. The default for e3 is 1.

DO (non-block)

```
DO label [,] index = e1, e2 [,
e3]
```

label	labels the last executable statement in the loop (this must not be a transfer of control).
index	is the name of a variable called the DO variable.
e1	is an expression which yields an initial value for i.
e2	is an expression which yields a final value for i.
e3	is an optional expression yielding a value specifying the increment value for i. The default for e3 is 1.

Description

The DO loop consists of all the executable statements after the specifying DO statement up to and including the labeled statement, called the terminal statement. The label is optional. If omitted, the terminal statement of the loop is an END DO statement.

Before execution of a DO loop, an iteration count is initialized for the loop. This value is the number of times the DO loop is executed, and is:

```
INT((e2-e1+e3)/e3)
```

If the value obtained is negative or zero the loop is not executed.

The DO loop is executed first with i taking the value e1, then the value (e1+e3), then the value (e1+e3+e3), etc.

It is possible to jump out of a DO loop and jump back in, as long as the do index variable has not been adjusted. In a nested DO loop, it is legal to transfer control from an inner loop to an outer loop. It is illegal, however, to transfer into a nested loop from outside the loop.

Syntax Extension§

Nested DO loops may share the same labeled terminal statement if required. They may not share an END DO statement.

Examples

```
 DO 100 J = -10,10
 DO 100 I = -5,5
100 SUM = SUM + ARRAY (I,J)
 DO
 A(I)=A(I)+1
 IF (A(I).EQ.4) EXIT
 END DO
 DO I=1,N
 A(I)=A(I)+1
 END DO
```

DO WHILE 77

The DO WHILE statement introduces a logical do loop and specifies the loop control expression.

The DO WHILE statement executes for as long as the logical expression continues to be true when tested at the beginning of each iteration. If expression is false, control transfers to the statement following the loop.

Syntax

```
DO [label[,]] WHILE
expression
```

The end of the loop is specified in the same way as for an iterative loop, either with a labeled statement or an END DO.

label	labels the last executable statement in the loop (this must not be a transfer of control).
expression	is a logical expression and label.

Description

The logical expression is evaluated. If it is .FALSE., the loop is not entered. If it is .TRUE., the loop is executed once. Then logical expression is evaluated again, and the cycle is repeated until the expression evaluates .FALSE.

Example

```
DO WHILE (K == 0)
 SUM = SUM + X
END DO
```

DOUBLE COMPLEX § 77

The DOUBLE COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for DOUBLE COMPLEX has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE COMPLEX [, attribute-list ::] entity-list
```

attribute-list	is the list of attributes for the double complex variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
DOUBLE COMPLEX name [/clist/] [,
name] [/clist/]...
```

name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a DOUBLE COMPLEX variable is 16 bytes. With the -r8 option, the default size of a DOUBLE COMPLEX variable is also 16 bytes.

Examples

```
DOUBLE COMPLEX CURRENT, NEXT
```

DOUBLE PRECISION 90

The DOUBLE PRECISION statement establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for DOUBLE PRECISION has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE PRECISION [, attribute-list ::] entity-list
```

attribute-list	is the list of attributes for the double precision variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
DOUBLE PRECISION name [/clist/] [,
name] [/clist/]...
```

name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a DOUBLE PRECISION variable is 8 bytes.

Example

```
DOUBLE PRECISION PLONG
```

ELSE 77

The ELSE statement begins an ELSE block of an IF block and encloses a series of statements that are conditionally executed.

Syntax

```
IF logical expression THEN
 statements
ELSE IF logical expression THEN
 statements
ELSE
 statements
END IF
```

The ELSE section is optional and may occur only once. Other IF blocks may be nested within the statements section of an ELSE block.

Example

```
IF (I.LT.15) THEN
 M = 4
ELSE
 M=5
END IF
```

ELSE IF 77

The ELSE IF statement begins an ELSE IF block of an IF block series and encloses statements that are conditionally executed.

Syntax

```
IF logical expression THEN
 statements
ELSE IF logical expression THEN
 statements
ELSE
 statements
END IF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an ELSE IF block.

Example

```
IF (I.GT.70) THEN
 M=1
ELSE IF (I.LT.5) THEN
 M=2
ELSE IF (I.LT.16) THEN
 M=3
END IF
```

ELSE WHERE 90

The WHERE statement and the WHERE ELSE WHERE construct permit masked assignments to the elements of an array (or to a scalar, zero dimensional array).

Syntax

WHERE Statement

```
WHERE (logical-array-expr) array-variable = array-expr
```

WHERE Construct

```
WHERE (logical-array-expr)
 array-assignments
[ELSE WHERE
 array-assignments ]
END WHERE
```

Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 ) GRADE = 'P'
WHERE ( SCORE > 60 )
 GRADE = 'P'
ELSE WHERE
 GRADE = 'F'
END WHERE
```

ENCODE § 77

The ENCODE statement transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE statements.

Syntax

```
ENCODE (c,f,b[,IOSTAT=ios] [,ERR=errs])[list]
```

c	is an integer expression specifying the number of bytes involved in translation.
f	is the format-specifier.
b	is a scalar or array reference for the buffer area receiving formatted data (characters).
ios	is an integer scalar memory reference which is the input/output status specifier: if this is included, ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.
errs	an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by errs.
list	a list of output items.

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4
ch
ch = '()'
accept ch
```

See Also

READ, PRINT

END 77

The END statement terminates a segment of a Fortran program. There are several varieties of the END statement. Each is described below.

END Syntax

```
END
```

Description

The END statement terminates a module. The END statement has the same effect as a RETURN statement in a SUBROUTINE or FUNCTION, or the effect of a STOP statement in a PROGRAM program unit. END may be the last statement in a compilation or it may be followed by a new program unit or module.

END DO Syntax

The END DO statement terminates a DO or DO WHILE loop.

```
END DO
```

Description

The END DO statement terminates an indexed DO or DO WHILE statement which does not contain a terminal-statement label.

The END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement contains a terminal-statement label.

END FILE Syntax

```
END FILE u
END FILE ([UNIT=]u, [,IOSTAT =ios] [,ERR=errs] )
```

u	is the external unit specifier where u is an integer.
IOSTAT=ios	an integer scalar memory reference which is the input/output specifier: if this is included in list, ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.
ERR=errs	an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by errs.

Description

When an END FILE statement is executed an endfile record is written to the file as the next record. The file is then positioned after the endfile record. Note that only records written prior to the endfile record can be read later.

A BACKSPACE or REWIND statement must be used to reposition the file after an END FILE statement prior to the execution of any data transfer statement. A file is created if there is an END FILE statement for a file connected but not in existence.

For example:

```
END FILE(20)
END FILE(UNIT=34, IOSTAT=IOERR, ERR=140)
```

END IF Syntax

The END IF statement terminates an IF ELSE or ELSE IF block.

Syntax

```
END IF
```

Description

The END IF statement terminates an IF block. There must be a matching block IF statement (at the same IF level) earlier in the same subprogram.

Syntax Extension – END MAP§

END MAP Syntax

The END MAP statement terminates a MAP declaration.

Syntax

```
END MAP
```

Description

See the MAP statement for details.

END SELECT Syntax

The END SELECT statement terminates a SELECT declaration.

Syntax

```
END SELECT
```

Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
 TYPE=1
CASE ( 4:6 )
 TYPE=2
CASE DEFAULT
 TYPE=0
END SELECT
```

Syntax Extension – END STRUCTURE§

END STRUCTURE Syntax

The END STRUCTURE statement terminates a STRUCTURE declaration.

Syntax

```
END STRUCTURE
```

Description

See the STRUCTURE statement for details.

Syntax Extension – END UNION§

END UNION

The END UNION statement terminates a UNION declaration.

Syntax

```
END UNION
```

Description

See the UNION statement for details.

ENTRY 77

The ENTRY statement allows a subroutine or function to have more than one entry point.

Syntax

```
ENTRY name [(variable, variable...)]
```

name	is the symbolic name, or entry name, by which the subroutine or function may be referenced.
variable	is a dummy argument. A dummy argument may be a variable name, array name, dummy procedure or, if the ENTRY is in a subroutine, an alternate return argument indicated by an asterisk. If there are no dummy arguments name may optionally be followed by (). There may be more than one ENTRY statement within a subroutine or function, but they must not appear within a block IF or DO loop.

Description

The name of an ENTRY must not be used as a dummy argument in a FUNCTION, SUBROUTINE or ENTRY statement, nor may it appear in an EXTERNAL statement.

Within a function a variable name which is the same as the entry name may not appear in any statement that precedes the ENTRY statement, except in a type statement.

If name is of type character the names of each entry in the function and the function name must be of type character. If the function name or any entry name has a length of (*) all such names must have a length of (*); otherwise they must all have a length specification of the same integer value.

A name which is used as a dummy argument must not appear in an executable statement preceding the ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE or ENTRY statement that precedes the executable statement. Neither must it appear in the expression of a statement function unless the name is also a dummy argument of the statement function, or appears in a FUNCTION or SUBROUTINE statement, or in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, execution of that statement is only permitted during the execution of a reference to the function or subroutine if the dummy argument appears in the dummy argument list of the procedure name referenced.

When a subroutine or function is called using the entry name, execution begins with the statement immediately following the ENTRY statement. If a function entry has no dummy arguments the function must be referenced by name() but a subroutine entry without dummy arguments may be called with or without the parentheses after the entry name.

An entry may be referenced from any program unit except the one in which it is defined.

The order, type, number and names of dummy arguments in an ENTRY statement can be different from those used in the FUNCTION, SUBROUTINE or other ENTRY statements in the same program unit but each reference must use an actual argument list which agrees in order, number and type with the dummy argument list of the corresponding FUNCTION, SUBROUTINE or ENTRY statement. When a subroutine name or an alternate return specifier is used as an actual argument there is no need to match the type.

Entry names within a FUNCTION subprogram need not be of the same data type as the function name, but they all must be consistent within one of the following groups of data types:

```
BYTE, INTEGER*2, INTEGER*4, LOGICAL*1,
LOGICAL*2, LOGICAL*4, REAL*4, REAL*8,
COMPLEX*8
COMPLEX*16
CHARACTER
```

If the function is of character data type, all entry names must also have the same length specification as that of the function.

Example

```
FUNCTION SUM(TALL,SHORT,TINY)
 .
SUM=TALL-(SHORT+TINY)
RETURN
ENTRY SUM1(X,LONG,TALL,WIDE,NARROW)
 .
 .
SUM1=(X*LONG)+(TALL*WIDE)+NARROW
RETURN
ENTRY SUM2(SHORT,SMALL,TALL,WIDE)
 .
 .
SUM2=(TALL-SMALL)+(WIDE-SHORT)
RETURN
END
```

When the calling program calls the function SUM it can do so in one of three ways depending on which ENTRY point is desired.

For example if the call is:

```
Z=SUM2(LITTLE,SMALL,BIG,HUGE)
```

the ENTRY point is SUM2.

If the call is:

```
Z=SUM(T,X,Y)
```

the ENTRY point is SUM and so on.

EQUIVALENCE 77

The EQUIVALENCE statement allows two or more named regions of data memory to share the same start address. Arrays that are subject to the EQUIVALENCE statement in HPF are treated as sequential and any attempt at non-replicated data distribution or mapping is ignored for such arrays.

Syntax

```
EQUIVALENCE (list)[,(list)...]
```

list	is a set of identifiers (variables, arrays or array elements) which are to be associated with the same address in data memory. The items in a list are separated by commas, and there must be at least two items in each list. When an array element is chosen, the subscripts must be integer constants or integer PARAMETER constants.

Description

The statement can be used to make a single region of data memory have different types, so that for instance the imaginary part of a complex number can be treated as a real value. It can also be used to make arrays overlap, so that the same region of store can be dimensioned in several different ways. Records and record fields cannot be specified in EQUIVALENCE statements.

Syntax Extension§

An array element may be identified with a single subscript in an EQUIVALENCE statement even though the array is defined to be a multidimensional array. Also, EQUIVALENCE of character and non-character data is allowed as long as misalignment of non-character data does not occur.

Example

```
COMPLEX NUM
REAL QWER(2)
EQUIVALENCE (NUM,QWER(1))
```

In the above example, QWER(1) is the real part of NUM and QWER(2) is the imaginary part. EQUIVALENCE statements are illegal if there is any attempt to make a mapping of data memory inconsistent with its linear layout.

EXIT 90

The EXIT statement interrupts a DO construct execution and continues with the next statement after the loop.

Syntax

```
EXIT [do-construct-name]
```

Example

```
DO
 IF (A(I).EQ.0) CYCLE
 B=100/A(I)
 IF (B.EQ.5) EXIT
END DO
```

See Also

CYCLE, DO

EXTERNAL 77

The EXTERNAL statement identifies a symbolic name as an external or dummy procedure. This procedure can then be used as an actual argument.

Syntax

```
EXTERNAL proc [,proc]..
```

proc	is the name of an external procedure, dummy procedure or block data program unit. When an external or dummy procedure name is used as an actual argument in a program unit it must appear in an EXTERNAL statement in that program unit.

Description

If an intrinsic function appears in an EXTERNAL statement an intrinsic function of the same name cannot then be referenced in the program unit. A symbolic name can appear only once in all the EXTERNAL statements of a program unit.

EXTRINSIC HPF

The EXTRINSIC statement identifies a symbolic name as an external or dummy procedure that is written in some language other than HPF.

Syntax

```
EXTRINSIC ( extrinsic-kind-keyword ) procedure name
```

extrinsic-kind-keyword	is the name of an extrinsic interface supported. The currently supported value is F77_LOCAL.
procedure name	is either a subroutine-statement or a function-statement defining a name for an external and extrinsic procedure.

Description

The EXTRINSIC procedure can then be used as an actual argument once it is defined. The call to an EXTRINSIC procedure should be semantically equivalent to the execution of an HPF procedure in that on return from the procedure, all processors are still available, and all data and templates will have the same distribution and alignment as when the procedure was called.

See Also

For a complete description of the PGHPF extrinsic facility, along with examples, refer to the PGHPF User's Guide.

FORALL F95

The FORALL statement and the FORALL construct provide a parallel mechanism to assign values to the elements of an array.

Syntax

```
FORALL (forall-triplet-spec-list [, scalar-mask-expr] )
forall-assignment
```

or

```
FORALL (forall-triplet-spec-list [, scalar-mask-expr] )
forall-body
[forall-body ]...
END FORALL
```

where forall-body is one of:

```
forall-assignment
where-statement
where-construct
forall-statement
forall-construct
```

Description

The FORALL statement is computed in four stages:

First, compute the valid set of index values. Second, compute the active set of index values, taking into consideration the scalar-mask-expr. If no scalar-mask-expr is present, the valid set is the same as the active set of index values. Third, for each index value, the right-hand side of the body of the FORALL is computed. Finally, the right-hand side is assigned to the left-hand side, for each index value.

Examples

```
FORALL (I = 1:3) A(I) = B(I)
FORALL(I = 1:L, A(I) == 0.0) A(I) = R(I)
FORALL (I = 1:3)
 A(I) = D(I)
 B(I) = C(I) * 2
END FORALLFORALL (I = 1:5)
 WHERE (A(I,:) /= 0.0)
 A(I,:) = A(I-1,:) + A(I+1,:)
 ELSEWHERE
 B(I,:) = A(6-I,:)
 END WHERE
END FORALL
```

FORMAT 77

The FORMAT statement specifies format requirements for input or output.

Syntax

```
label FORMAT (list-items)
```

list-items can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.

- Nonrepeatable editor commands.
- A format specification list optionally preceded by an integer constant which defines the number of repeats.

Each action of format control depends on the next edit code and the next item in the input/output list where one is used. If an input/output list contains at least one item there must be at least one repeatable edit code in the format specification. An empty format specification () can only be used if no list items are specified; in such a case one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. Where a repeat specification is used the appropriate item is repeated the required number of times.

Description

Refer to for more details on using the FORMAT statement.

Examples

```
  WRITE (6,90) NPAGE
90 FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

produces:

```
  PAGE NUMBER 10 SALES REPORT, Cont.
```

The following example shows use of the tabulation specifier T:

```
  PRINT 25
25 FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

produces:

```
  COLUMN
1 COLUMN 2
```

```
  DIMENSION A(6)
  DO 10 I = 1,6
10 A(I) = 25.
  TYPE 100,A
100 FORMAT(' ',F8.2,2PF8.2,F8.2) ! ' '
C ! gives single spacing
```

produces:

```
25.00
2500.00 2500.00
2500.00 2500.00 2500.00
```

Note that the effect of the scale factor continues until another scale factor is used.

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4
ch
ch = '()'
accept ch
```

See Also

READ, PRINT

FUNCTION 77

The FUNCTION statement introduces a program unit; the statements that follow all apply to the function itself and are laid out in the same order as those in a PROGRAM program unit.

Syntax

```
[function-prefix] FUNCTION name [*n] ([argument [,argument]...])
.
.
.
END [ FUNCTION [function-name]]
```

function-prefix is one of:

[type-spec] RECURSIVE

[RECURSIVE] type-spec

	where type-spec is a valid type specification. Type will explicitly apply a type to the function. If the function is not explicitly typed then the function type is taken from the initial letter and is dictated by the usual default.
name	is the name of the function and must be unique among all the program unit names in the program. name must not clash with any local, COMMON or PARAMETER names.
*n	is the optional length of the data type.
argument	is a symbolic name, starting with a letter and containing only letters and digits. An argument can be of type REAL, INTEGER, DOUBLE PRECISION, CHARACTER, LOGICAL, COMPLEX, or BYTE, etc.

Description

The statements and names apply only to the function, except for subroutine or function references and the names of COMMON blocks. The function must be terminated by an END statement.

A function produces a result; this allows a function reference to appear in an expression, where the result is assumed to replace the actual reference. The symbolic name of the function must appear as a variable in the function, unless the RESULT keyword is used. The value of this variable, on exit from the function, is the result of the function. The function result is undefined if the variable has not been defined.

The type of a FUNCTION refers to the type of its result.

Recursion is allowed if the –Mrecursive option is used on the command-line and the RECURSIVE prefix is included in the function definition.

Examples

```
FUNCTION FRED(A,B,C)
REAL X
.
END
FUNCTION EMPTY() ! Note parentheses
END
PROGRAM FUNCALL
.
SIDE=TOTAL(A,B,C)
.
```

```
 END
 FUNCTION TOTAL(X,Y,Z)
 .
 END
 FUNCTION AORB(A,B)
 IF(A-B)1,2,3
1 AORB = A
 RETURN
2 AORB = B
 RETURN
3 AORB = A + B
 RETURN
 END
```

See Also

PURE, RECURSIVE, RESULT

GOTO (Assigned)

(Obsolescent) The assigned GOTO statement transfers control so that the statement identified by the statement label is executed next. Internal procedures can be used in place of the assign statement used with an assigned GO TO.

Syntax

```
GOTO integer-variable-name[[,] (list)]
```

integer-variable-name	must be defined with the value of a statement label of an executable statement within the same program unit. This type of definition can only be done by the ASSIGN statement.
list	consists of one or more statement labels attached to executable statements in the same program unit. If a list of statement labels is present, the statement label assigned to the integer variable must be in that list.

Examples

```
 ASSIGN 50 TO K
 GO TO K(50,90)
90 G=D**5
 .
 .
50 F=R/T
```

GOTO (Computed) 77

The computed GOTO statement allows transfer of control to one of a list of labels according to the value of an expression.

Syntax

```
GOTO (list) [,] expression
```

list	is a list of labels separated by commas.
expression	selects the label from the list to which to transfer control. Thus a value of 1 implies the first label in the list, a value of 2 implies the second label and so on. An expression value outside the range will result in transfer of control to the statement following the computed GOTO statement.

Example

```
 READ *, A, B
 GO TO (50,60,70)A
 WRITE (*, 10) A, B
10 FORMAT (' ', I3, F10.4, 5X, 'A must be 1, 2
 + or 3')
 STOP
50 X=A**B ! Come here if A has the
value 1
 GO TO 100
60 X=(A*56)*(B/3) ! Come
here if A is 2
 GO TO 100
70 X=A*B ! Come here if A has the value 3
100 WRITE (*, 20) A, B, X
20 FORMAT (' ', I3, F10.4, 5X, F10.4)
```

GOTO (Unconditional) 77

The GOTO statement unconditionally transfers control to the statement with the label label. The statement label label must be declared within the code of the program unit containing the GOTO statement and must be unique within that program unit.

Syntax

```
GOTO label
```

label is a statement label

Example

```
 TOTAL=0.0
30 READ *, X
 IF (X.GE.0) THEN
 TOTAL=TOTAL+X
 GOTO 30
 END IF
```

IF (Arithmetic) 77

(Obsolescent) The arithmetic IF statement transfers control to one of three labeled statements. The statement chosen depends upon the value of an arithmetic expression.

Syntax

```
IF (arithmetic-expression) label-1, label-2, label-3
```

Control transfers to label-1, label-2 or label-3 if the result of the evaluation of the arithmetic-expression is less than zero, equal to zero or greater than zero respectively.

Example

```
IF X 10, 20, 30
```

If X is less than zero then control is transferred to label 10. If X equals zero then control is transferred to label 20. If X is greater than zero then control is transferred to label 30.

IF (Block) 77

The block IF statement consists of a series of statements that are conditionally executed.

Syntax

```
IF logical expression THEN
 statements
ELSE IF logical expression THEN
 statements
ELSE
 statements
END IF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an ELSE IF block.

The ELSE section is optional and may occur only once. Other IF blocks may be nested within the statements section of an ELSE block.

Example

```
IF (I.GT.70) THEN
 M=1
ELSE IF (I.LT.5) THEN
 M=2
ELSE IF (I.LT.16) THEN
 M=3
END IF
IF (I.LT.15) THEN
 M = 4
ELSE
 M=5
END IF
```

IF (Logical) 77

The logical IF statement executes or does not execute a statement based on the value of a logical expression.

Syntax

```
IF (logical-expression) statement
```

logical-expression	is evaluated and if it is true the statement is executed. If it is false, the statement is not executed and control is passed to the next executable statement.

statement	can be an assignment statement, a CALL statement or a GOTO statement.

Examples

```
IF(N .LE. 2) GOTO 27
IF(HIGH .GT. 1000.0 .OR. HIGH .LT. 0.0) HIGH=1000.0
```

IMPLICIT 77

The IMPLICIT statement redefines the implied data type of symbolic names from their initial letter. Without the use of the IMPLICIT statement all names that begin with the letters I,J,K,L,M or N are assumed to be of type integer and all names beginning with any other letters are assumed to be real.

Syntax

```
IMPLICIT spec (a[,a]...) [,spec
(a[,a]...)]
IMPLICIT NONE
```

spec	is a data type specifier.
a	is an alphabetic specification expressed either as a or a1-a2, specifying an alphabetically ordered range of letters.

Description

IMPLICIT statements must not be labeled.

Symbol names may begin with a dollar sign ($) or underscore (_) character, both of which are of type REAL by default. In an IMPLICIT statement, these characters may be used in the same manner as other characters, but they cannot be used in a range specification.

The IMPLICIT NONE statement specifies that all symbolic names must be explicitly declared, otherwise an error is reported. If IMPLICT NONE is used, no other IMPLICIT can be present.

Examples

```
IMPLICIT REAL (L,N)
IMPLICIT INTEGER (S,W-Z)
IMPLICIT INTEGER (A-D,$,_)
```

INCLUDE § 77

The INCLUDE statement directs the compiler to start reading from another file.

Note

The INCLUDE statement is used for FORTRAN 77. There is no support for VAX/VMS text libraries or the module_name pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

Syntax

```
INCLUDE 'filename [/[NO]LIST]'
INCLUDE "filename [/[NO]LIST]"
```

The INCLUDE statement may be nested to a depth of 20 and can appear anywhere within a program unit as long as Fortran's statement-ordering restrictions are not violated.

§The qualifiers /LIST and /NOLIST can be used to control whether the include file is expanded in the listing file (if generated).

Either single or double quotes may be used.

If the final component of the file pathname is /LIST or /NOLIST, the compiler will assume it is a qualifier, unless an additional qualifier is supplied.

The filename and the /LIST or /NOLIST qualifier may be separated by blanks.

The include file is searched for in the following directories:

- Each –I directory specified on the command-line.
- The directory containing the file that contains the INCLUDE statement (the current working directory.)
- The standard include area.

Example

```
INCLUDE '/mypath/list /list'
```

This line includes a file named /mypath/list and expands it in the listing file, if a listing file is used.

INQUIRE 77

An INQUIRE statement has two forms and is used to inquire about the current properties of a particular file or the current connections of a particular unit. INQUIRE may be executed before, during or after a file is connected to a unit.

Syntax

```
INQUIRE (FILE=filename, list)
INQUIRE ([UNIT=]unit,list)
```

In addition list may contain one of each of the following specifiers in any order, following the unit number if the optional UNIT specifier keyword is not supplied.

ACCESS=acc	acc returns a character expression specifying the access method for the file as either DIRECT or SEQUENTIAL. The default is SEQUENTIAL.
ACTION=acc	acc is a character expression specifying the access types for the connection. Either READ, WRITE, or READWRITE.
BLANK= blnk	blnk is a character expression which returns the value NULL or ZERO or UNDEFINED. NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. This specifier must only be used when a file is connected for formatted input/output.
DELIM= del_char del_char	is a character expression which returns the value APOSTROPHE, QUOTE, NONE or UNDEFINED. These values specify the character delimiter for list-directed or namelist formatted data transfer statements.
DIRECT= dir_char	dir_char is a character reference which returns the value YES if DIRECT is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if DIRECT is included.
ERR= errs	errs is an error specifier which returns the value of a statement label of an executable statement within the same program. If an error condition occurs execution continues with the statement specified by errs.

EXIST= value	value is a logical variable or logical array element which becomes .TRUE. if there is a file/unit with the name or .FALSE. otherwise.
FILE= fin	fin is a character expression whose value is the file name expression, the name of the file connected to the specified unit.
FORM= fm	fm is a character expression specifying whether the file is being connected for FORMATTED or UNFORMATTED input/output. The default is UNFORMATTED.
FORMATTED= fmt	fmt is a character memory reference which takes the value YES if FORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if FORMATTED is included.
IOSTAT= ios	ios input/output status specifier where ios is an integer reference: if this is included in list, ios is defined as 0 if no error condition occurred and a positive integer when there is an error condition.
NAME= fn	fn is a character scalar memory reference which is assigned the name of the file when the file has a name, otherwise it is undefined
NAMED= nmd	nmd is a logical scalar memory reference which becomes .TRUE. if the file has a name, otherwise it becomes .FALSE.
NEXTREC= nr	nr is an integer scalar memory reference which is assigned the value n+1, where n is the number of the record read or written. It takes the value 1 if no records have been read or written. If the file is not connected or its position is indeterminate nr is undefined.
NUMBER= num	num is an integer scalar memory reference or integer array element assigned the value of the external unit number of the currently connected unit. It becomes undefined if no unit is connected.
OPENED= od	od is a logical scalar memory reference which becomes .TRUE. if the file/unit specified is connected (open) and .FALSE. if the file is not connected (.FALSE.).
PAD= pad_char	pad_char is a character expression specifying whether to use blank padding. Values for pad_char are YES or NO: yes specifies blank padding is used, no requires that input records contain all requested data.

POSITION= pos_char	pos_char is a character expression specifying the file position. Values are ASIS, REWIND or APPEND. For a connected file, on OPEN ASIS leaves the position in the current position, REWIND rewinds the file and APPEND places the current position at the end of the file, immediately before the end-of-file record.
READ= rl	rl is a character reference which takes the value YES if UNFORMATTED is one of the allowed access methods for file, NO if not, or UNKNOWN if it is not known if UNFORMATTED is included.
READWRITE= rl	rl is a character scalar memory reference which takes the value YES if UNFORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if UNFORMATTED is included.
RECL= rcll	rcl is an integer expression defining the record length in a file connected for direct access. When sequential input/output is specified this is the maximum record length. This specifier must only be given when a file is connected for direct access.
SEQUENTIAL= seq	seq a character scalar memory reference which takes the value YES if UNFORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if UNFORMATTED is included.
UNFORMATTED= unf	unf a character scalar memory reference which takes the value YES if UNFORMATTED is one of the allowed access methods for the file, NO if not and UNKNOWN if it is not known if UNFORMATTED is included.
WRITE= rl	rl a character scalar memory reference which takes the value YES, NO, or UNKNOWN. These values indicate that WRITE is allowed, not allowed, or indeterminate for the specified file, respectively.

Description

When an INQUIRE by file statement is executed the following specifiers will only be assigned values if the file name is acceptable: nmd, fn, seq, dir, fmt and unf. num is defined, and acc, fm, rcl, nr and blnk may become defined only if od is defined as .TRUE..

When an INQUIRE by unit statement is executed the specifiers num, nmd, fn, acc, seq, dir, fm, fmt, unf, rcl, nr and blnk are assigned values provided that the unit exists and a file is connected to that unit. Should an error condition occur during the execution of an INQUIRE statement all the specifiers except ios become undefined.

INTEGER 77

The INTEGER statement establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for INTEGER has two forms, a standard FORTRAN 77 or 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
INTEGER [([ KIND = kind-value
) ][, attribute-list ::] entity-list
```

INTEGER permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list	is the list of attributes for the character variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
INTEGER [*n] [,] name [*n] [dimensions] [/clist/]...
```

n	is an optional size specification.
name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Integer type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. INTEGER statements must not be labeled. The default size of an INTEGER variable is 4 bytes. With the -Mnoi4 option, the default size of an INTEGER variable is 2 bytes.

Note

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

Example

```
INTEGER TIME, SECOND, STORE (5,5)
```

INTENT 90

The INTENT specification statement (attribute) specifies intended use of a dummy argument. This statement (attribute) may not be used in a main program's specification statement.

Syntax

```
INTENT (intent-spec) [ :: ] dummy-arg-list
```

intent-spec — is one of:

- IN
- OUT
- INOUT

dummy-arg-list — is the list of dummy arguments with the specified intent.

Description

With intent specified as IN, the subprogram argument must not be redefined by the subprogram.

With intent specified as OUT, the subprogram should use the argument to pass information to the calling program.

With intent specified as INOUT, the subprogram may use the value passed through the argument, but should also redefine the argument to pass information to the calling program.

See Also

OPTIONAL

Example

```
SUBROUTINE IN_OUT(R1,I1)
 REAL, INTENT (IN)::R1
 INTEGER, INTENT(OUT)::I1
 I1=R1
END SUBROUTINE IN_OUT
```

INTERFACE 90

The INTERFACE statement block makes an implicit procedure an explicit procedure where the dummy parameters and procedure type are known to the calling module. This statement is also used to overload a procedure name.

Syntax

```
INTERFACE [generic-spec]
 [interface-body]...
 [MODULE PROCEDURE procedure-name-list]...
END INTERFACE
```

where generic-spec is one of the following:

```
generic-name
OPERATOR (defined operator)
ASSIGNMENT (=)
```

and the interface body specifies the interface for a function or a subroutine:

```
function-statement
 [specification-part]
END FUNCTION [function name]
subroutine-statement
 [specification-part]
END FUNCTION [subroutine name]
```

See Also

END INTERFACE

Example

```
INTERFACE
 SUBROUTINE IN_OUT(R1,I1)
 REAL, INTENT (IN)::R1
 INTEGER, INTENT(OUT)::I1
 END SUBROUTINE IN_OUT
END INTEFACE
```

INTRINSIC 77

An INTRINSIC statement identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.

Syntax

```
INTRINSIC func [,func]
```

func is the name of an intrinsic function such as SIN, COS, etc.

Description

Do not use any of the following functions in INTRINSIC statements:

- type conversions:

  ```
  INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR,
  CHAR
  ```

- lexical relationships:

  ```
  LGE, LGT, LLE, LLT
  ```

- values:

  ```
  MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1,
  DMIN1, AMIN0, MIN1
  ```

When a specific name of an intrinsic function is used as an actual argument in a program unit it must appear in an INTRINSIC statement in that program unit. If the name used in an INTRINSIC statement is also the name of a generic intrinsic function, it retains its generic properties. A symbolic name can appear only once in all the INTRINSIC statements of a program unit and cannot be used in both an EXTERNAL and INTRINSIC statement in a program unit.

The following example illustrates the use of INTRINSIC and EXTERNAL:

```
EXTERNAL MYOWN
INTRINSIC SIN, COS
.
.
CALL TRIG (ANGLE,SIN,SINE)
.
CALL TRIG (ANGLE,MYOWN,COTANGENT)
.
CALL TRIG (ANGLE,COS,SINE)
SUBROUTINE TRIG (X,F,Y)
Y=F(X)
RETURN
END
```

```
FUNCTION MYOWN
MYOWN=COS(X)/SIN(X)
RETURN
END
```

In this example, when TRIG is called with a second argument of SIN or COS the function reference F(X) references the intrinsic functions SIN and COS; however when TRIG is called with MYOWN as the second argument F(X) references the user function MYOWN.

LOGICAL 77

The LOGICAL statement establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for LOGICAL has two forms, a standard FORTRAN 77 and 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
LOGICAL [ ( [ KIND = kind-value
) ] [, attribute-list ::] entity-list
```

LOGICAL permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list	is the list of attributes for the character variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
LOGICAL [*n] [,] name [*n] [dimensions] [/clist/]
 [, name] [*n][dimensions] [/clist/]...
```

n	is an optional size specification.
name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

Logical type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a LOGICAL variable is 4 bytes. With the -Mnoi4 option, the default size of a LOGICAL variable is 2 bytes.

Example

```
LOGICAL TIME, SECOND, STORE(5,5)
```

MAP § 77

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. Field alignment within multiple map declarations is performed as previously defined in structure declarations.

Syntax

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

field_declaration	is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

Data can be initialized in field declaration statements in union declarations. However, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

Example

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
 INTEGER typetag ! Tag to determine defined map
 UNION
 MAP ! Structure for an employee
 CHARACTER*12 ssn ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12) ! Items supplied
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

MODULE 90

(PGF95 and PGHPF only) The MODULE statement specifies the entry point for a Fortran 90/95 module program unit. A module defines a host environment of scope of the module, and may contain subprograms that are in the same scoping unit.

Syntax

```
MODULE [name]
 [specification-part]
 [ CONTAINS [module-subprogram-part]]
END [MODULE [ module-name ]]
```

name	is optional; if supplied it becomes the name of the program module and must not clash with any other names used in the program. If it is not supplied, a default name is used.
specification-part	contains specification statements. See the Fortran 95 Handbook for a complete description of the valid statements.
module-subprogram-part	contains function and subroutine definitions for the module, preceded by a single CONTAINS keyword.

Modules can be independently compiled and used within programs using the USE statement. Use of Fortran 90/95 modules causes the compiler to create a filename.mod file in the current directory (a .mod file). This file contains all the information the compiler needs concerning interface specifications and the data types for the routines defined in the module. When a program, routine, or another module encounters the USE statement, the .mod file is read and "included" in the program, using the scope rules defined in Fortran 90/95 for USE association. If you are using separate modules, this creates another step in the program development process. When a module is compiled, both a .mod and an object file are created. The .mod file is used when a USE statement is encountered, and the object file is used when the program is linked.

For example, if module1.f contains a module with several procedures, and test1.f contains a USE statement that uses module1, the compilation would involve the steps.

```
% pgf95 -c module1.f
% pgf95 -o test1 test1.f module1.o
```

The search for a .mod file includes the following directories:

Each –I directory specified on the command-line.

The directory containing the file that contains the USE statement (the current working directory.)

The standard include area.

Using the –I command-line option directories can be added to the search path for .mod files.

Example

```
MODULE MYOWN
 REAL MEAN, TOTAL
 INTEGER, ALLOCATABLE, DIMENSION(:):: A
 CONTAINS
 RECURSIVE INTEGER FUNCTION X(Y)
 .
 .
 .
 END FUNCTION X
END MODULE MYOWN
```

NAMELIST 90

The NAMELIST statement allows for the definition of namelist groups for namelist-directed I/O.

Syntax

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

group-name	is the name of the namelist group.
namelist	is the list of variables in the namelist group.

Example

In the following example a named group PERS consists of a name, an account, and a value.

```
CHARACTER*12 NAME
INTEGER*$ ACCOUNT
REAL*4 VALUE
NAMELIST /PERS/ NAME, ACCOUNT, VALUE
```

NULLIFY 90

The NULLIFY statement disassociates a pointer from its target.

Syntax

```
NULLIFY (pointer-object-list)
```

Example

```
NULLIFY (PTR1)
```

See Also

ALLOCATE, DEALLOCATE

OPEN 77

The OPEN statement connects an existing file to a unit, creates and connects a file to a unit, creates a file that is preconnected or changes certain specifiers of a connection between a file and a unit.

Syntax

```
OPEN ( list )
```

list	must contain exactly one unit specifier of the form:

```
[UNIT=] u
```

where the UNIT= is optional and the external unit specifier u is an integer.

In addition list may contain one of each of the following specifiers in any order, following the unit number if the optional UNIT specifier keyword is not supplied.

ACCESS= acc	acc is a character expression specifying the access method for file connection as SEQUENTIAL, DIRECT or STREAM; the default is SEQUENTIAL.
ACTION= acc	acc is a character expression specifying the permitted access types for connection. One of READ, WRITE, UNKNOWN or READWRITE is allowed. The default is UNKNOWN .
ASYNCHRONOUS=async	async is a character expression specifying whether to allow asynchronous data transfer on this file connection. One of ‘YES’ or ‘NO’ is allowed.
BLANK=blnk	blnk is a character expression which takes the value 'NULL' or 'ZERO'. 'NULL' causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of

	zero. 'ZERO' causes all blanks other than leading blanks to be treated as zeros. The default is 'NULL.' This specifier must only be used when a file is connected for formatted input/output.
DELIM= del_char	del_char is a character expression which takes the value 'APOSTROPHE', 'QUOTE' or 'NONE'. These values specify the character delimiter for list-directed or namelist formatted data transfer statements.
ERR=errs	errs is an error specifier; it takes the form of a statement label of an executable statement within the program. If an error condition occurs execution continues with the statement specified by errs.
FILE= fin	fin is a character expression whose value is the file name expression, the name of a file to be connected to the specified unit.
FORM=fm	fm is a character expression specifying whether the file is being connected for 'FORMATTED' or 'UNFORMATTED' input/output.
IOSTAT= ios	ios is an integer scalar; if this is included ios becomes defined with 0 (zero) if no error condition exists or a positive integer when there is an error condition. A value of -1 indicates an end-of-file condition with no error. A value of -2 indicates an end-of-record condition with no error when using non-advancing I/O.
PAD= pad_char	pad_char is a character expression specifying whether to use blank padding. Acceptable values are YES or NO; yes specifies that blank padding is used and no requires that input records contain all requested data.
POSITION= pos_char	pos_char is a character expression specifying the file position. Values are ASIS, REWIND or APPEND. For a connected file, on OPEN ASIS leaves the position in the current position, REWIND rewinds the file and APPEND places the current position at the end of the file, immediately before the end-of-file record.
RECL= rl	rl is an integer expression defining the record length in a file connected for direct access. When sequential input/output is specified this is the maximum record length.

STATUS= sta	sta is a character expression whose value can be: NEW, OLD, SCRATCH, UNKNOWN or REPLACE. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default status is UNKNOWN which specifies that the file's existence is unknown, which limits the error checking when opening the file. With status OLD, the file must exist or an error is reported. With status NEW, the file is created; if the file exists, an error is reported. Status SCRATCH specifies that the file is removed when closed.

Description

The record length, RECL=, must be specified if a file is connected for direct access and optionally one of each of the other specifiers may be used. RECL is ignored if the access method is sequential.

The unit specified must exist and once connected by an OPEN statement can be referenced in any program unit of the executable program. If a file is connected to a unit it cannot be connected to a different unit by the OPEN statement.

If a unit is connected to an existing file, execution of an OPEN statement for that file is allowed. Where FILE= is not specified the file to be connected is the same as the file currently connected. If the file specified for connection to the unit does not exist but is the same as a preconnected file, the properties specified by the OPEN statement become part of the connection. However, if the file specified is not the same as the preconnected file this has the same effect as the execution of a CLOSE statement without a STATUS= specifier immediately before the execution of the OPEN statement. When the file to be connected is the same as the file already connected only the BLANK= specifier may be different from the one currently defined.

The sequential and direct access methods access files that contain fixed-length records. The stream access method, a Fortran 2003 language extension, allows access to files that do not contain fixed-length records. Stream I/O is enabled by specifying access='STREAM'. Stream I/O may be formatted or unformatted.

Asynchronous i/o, the ability to return control before the statement has completed, is supported in certain situations. If ASYNCHRONOUS='yes' is specified on the OPEN statement and a READ or WRITE statement for a particular file connection, a processor may perform an asynchronous data transfer asynchronously, but is not required to do so. In practice, the underlying operating system controls much of what can be performed. A file must be seekable to support aysynchronous I/O; i.e. you cannot perform asynchronous I/O on a non-seekable file such as a fifo. Asynchronous I/O is only supported for the stream access method.

Examples

In the following example a new file, BOOK, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters. Numeric values will have blanks ignored and E1 will be assigned some positive value if an error condition exists when the OPEN statement is executed; execution will then continue with the statement labeled 20. If no error condition pertains, E1 is assigned the value zero (0) and execution continues with the next statement.

```
OPEN(12, IOSTAT=E1, ERR=20, FILE='BOOK',
+ BLANK='NULL', ACCESS='DIRECT', RECL=98,
+ FORM='FORMATTED',STATUS='NEW')
```

The next example shows how to use asynchronous I/O.

```
program test
character*13 b
b = "hello, world\n"
open(unit=10,file='u.dat',access='stream',form='unformatted',
&
asynchronous='yes')
write (unit=10,asynchronous='yes') b
! Do something useful
wait(10)
close(10)
end
```

Environment Variables

For an OPEN statement which does not contain the FILE= specifier, an environment variable may be used to specify the file to be connected to the unit. If the environment variable FORddd exists, where ddd is a 3 digit string whose value is the unit, the environment variable's value is the name of the file to be opened.

PGI Fortran Extensions§

PGI has extended the OPEN statement as follows:

CONVERT=order	order is a character expression specifying the byte order of the file. One of 'BIG_ENDIAN', 'LITTLE_ENDIAN', or 'NATIVE' is allowed.

The CONVERT specifier allows byte-swapping I/O to be performed on specific logical units. The value 'BIG_ENDIAN' is used to convert big-endian format data files produced by most RISC workstations and high-end servers to the little-endian format used on Intel Architecture systems on-the-fly during

file reads/writes. This value assumes that the record layouts of unformatted sequential access and direct access files are the same on the systems. For the values 'LITTLE_ENDIAN' and 'NATIVE', byte-swapping is not performed during file reads/writes since the little-endian format is used by the x86 architecture.

VAX/VMS Fortran Extensions§

VAX/VMS introduces a number of extensions to the OPEN statement. Many of these relate only to the VMS file system and are not supported (e.g., KEYED access for indexed files). The following keywords for the OPEN statement have been added or augmented as shown below. Refer to Programming in VAX FORTRAN for additional details on these keywords.

ACCESS	The value of 'APPEND' will be recognized and implies sequential access and positioning after the last record of the file. Opening a file with append access means that each appended record is written at the end of the file.
ASSOCIATEVARIABLE	This keyword specifies an INTEGER*4 integer scalar memory reference which is updated to the next sequential record number after each direct access I/O operation. Applies only to direct access mode.
DISPOSE and DISP	These keywords specify the disposition for the file after it is closed. 'KEEP' or 'SAVE' is the default on anything other than STATUS='SCRATCH' files. 'DELETE' indicates that the file is to be removed after it is closed. The PRINT and SUBMIT values are not supported.
NAME	This keyword is a synonym for FILE.
READONLY	This keyword specifies that an existing file can be read but prohibits writing to that file. The default is read/write.
RECL=len	The record length given is interpreted as the number of words in a record if the runtime environment parameter FTNOPT is set to "vaxio". This simplifies the porting of VAX/VMS programs. The default is that len is given in number of bytes in a record.
TYPE	This keyword is a synonym for STATUS.

OPTIONAL 90

The OPTIONAL specification statement (attribute) specifies dummy arguments that may be omitted or that are optional.

Syntax

```
OPTIONAL [::] dummy-arg-list
```

Examples

```
OPTIONAL :: VAR4, VAR5
OPTIONAL VAR6, VAR7
INTEGER, OPTIONAL :: VAR8, VAR9
```

See Also

INTENT

OPTIONS § 77

The OPTIONS statement confirms or overrides certain compiler command-line options.

Syntax

```
OPTIONS /option [/option ...]
```

The following table shows what options are available for the OPTIONS statement:

Table 3-1: OPTIONS Statement

Option	Action Taken
CHECK=ALL	Enable array bounds checking
CHECK=[NO]OVERFLOW	None (recognized but ignored)
CHECK=[NO]BOUNDS	(Disable) Enable array bounds checking
CHECK=[NO]UNDERFLOW	None
CHECK=NONE	Disable array bounds checking
NOCHECK	Disable array bounds checking
[NO]EXTEND_SOURCE	(Disable) Enable the –Mextend option
[NO]G_FLOATING	None
[NO]F77	(Disable) Enable the –Mstandard option
[NO]I4	(Disable) Enable the –Mi4 option
[NO]RECURSIVE	(Disable) Enable the –Mrecursive option
[NO]REENTRANT	(Enable) Disable optimizations that may result in code that is not reentrant.
[NO]STANDARD	(Disable) Enable the –Mstandard option

The following restrictions apply to the OPTIONS statement:

- The OPTIONS statement must be the first statement in a program unit; it must precede the PROGRAM, SUBROUTINE, FUNCTION, and BLOCKDATA statements.
- The options listed in the OPTIONS statement override values from the driver command-line for the program unit (subprogram) immediately following the OPTIONS statement.
- Any abbreviated version of an option that is long enough to identify the option uniquely is a legal abbreviation for the option.

- Case is not significant, unless the –Mupcase is present on the command line. If it is, each option must be in lowercase.

PARAMETER 77

The PARAMETER statement gives a symbolic name to a constant.

Syntax

```
PARAMETER (name = expression[,name = expression...] )
```

expression	is an arithmetic expression formed from constant or PARAMETER elements using the arithmetic operators +, -, *, />. The usual precedence order can be changed by using parentheses. expression may include a previously defined PARAMETER.

Examples

```
PARAMETER ( PI = 3.142 )
PARAMETER ( INDEX = 1024 )
PARAMETER ( INDEX3 = INDEX * 3 )
```

The following VAX/VMS extensions to the PARAMETER statement are fully supported:

Its list is not bounded with parentheses.

The form of the constant (rather than the implicit or explicit typing of the symbolic name) determines the data type of the variable.

The form of the alternative PARAMETER statement is:

```
PARAMETER p=c [,p=c]...
```

where p is a symbolic name and c is a constant, symbolic constant, or a compile time constant expression.

PAUSE

(Obsolescent) The PAUSE statement stops the program's execution. The PAUSE statement is obsolescent because a WRITE statement may send a message to any device, and a READ statement may be used to wait for a message from the same device.

Syntax

```
PAUSE [character-expression | digits ]
```

The PAUSE statement stops the program's execution. The program may be restarted later and execution will then continue with the statement following the PAUSE statement.

POINTER 90

The POINTER specification statement or attribute declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its target pointer-based variable. The target may be a scalar or an array of any type.

Syntax

```
POINTER [::] object-name [ (deferred-shape-spec-list) ]
 [, object-name [ ( deferred-shape-spec-list ) ]]
```

Example

```
REAL, DIMENSION(:,:), POINTER :: X
```

POINTER (Cray) § 77

The POINTER statement is an extension to FORTRAN 77. It declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its pointer-based variable.

Syntax

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and v2	are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.
p1 and p2	are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in

Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined. The pointer is defined in one of the following ways:

By assigning the value of the LOC function.

By assigning a value defined in terms of another pointer variable.

By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

Example

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0 ! IC gets 0
P = LOC(XC)
Q = P + 20 ! same as LOC(XC(6))
X(1) = 0 ! XC(6) gets 0
ALLOCATE (X) ! Q locates a dynamically
 ! allocated memory area
```

Restrictions

- The following restrictions apply to the POINTER statement:
- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.

- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is assumed not to overlap with another pointer-based variable.

PRINT 77

The PRINT statement is a data transfer output statement.

Syntax

```
PRINT format-specifier [, iolist]
```

or

```
PRINT namelist-group
```

formatspecifier	a label of a format statement or a variable containing a format string.
iolist	is an input/output list that must either be one of the items in an input list or any other expression. A character expression involving concatenation of an operand of variable length cannot be included in an output list, however, unless the operand is the symbolic name of a constant.
namelist-group	the name of the namelist group.

Description

When a PRINT statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification.[1] The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

1. If an asterisk (*) is used instead of a format identifier, the list-directed formatting rules apply.

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
print sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
print ch
```

See Also

READ, PRINT

PRIVATE 90

The PRIVATE statement specifies entities defined in a module are not accessible outside of the module. This statement is only valid in a module. The default specification for a module is PUBLIC.

Syntax

```
PRIVATE [:: [ access-id-list ]]
```

Description

Example

```
MODULE FORMULA
PRIVATE
PUBLIC :: VARA
.
.
.
END MODULE
```

Type

Non-executable

See Also

PUBLIC, MODULE

PROGRAM 77

The PROGRAM statement specifies the entry point for the linked Fortran program.

Syntax

```
PROGRAM [name]
.
.
.
END [ PROGRAM [program-name]]
```

name	is optional; if supplied it becomes the name of the program module and must not clash with any other names used in the program. If it is not supplied, a default name is used.

Description

The program statement specifies the entry point for the linked Fortran program. An END statement terminates the program.

The END PROGRAM statement terminates a main program unit that begins with the optional PROGRAM statement. The program name found in the END PROGRAM must match that in the PROGRAM statement.

Example

```
PROGRAM MYOWN
REAL MEAN, TOTAL
.
CALL TRIG(A,B,C,MEAN)
.
END
```

PUBLIC 90

The PUBLIC statement specifies entities defined in a module are accessible outside of the module. This statement is only valid in a module. The default specification for a module is PUBLIC.

Syntax

```
PUBLIC [ :: [ access-id-list ]]
```

Example

```
MODULE FORMULA
PRIVATE
PUBLIC :: VARA
.
.
.
END MODULE
```

Type

Non-executable

See Also

PRIVATE, MODULE

PURE 95

The PURE attribute indicates that a function or subroutine has no side effects. Use of PURE can enable additional opportunities for optimization, and for the PGHPF compiler indicates that a subroutine or function can be used in a FORALL statement or construct or within an INDEPENDENT DO loop.

Syntax

```
PURE [type-specification] FUNCTION
```

or

```
type-specification PURE FUNCTION
```

or

```
PURE SUBROUTINE
```

Type

Non-executable

See Also

FUNCTION, SUBROUTINE

READ 90

The READ statement is the data transfer input statement.

Syntax

```
READ ([unit=] u, format-specifier [,control-information) [iolist]
READ format-specifier [,iolist]
READ ([unit=] u, [NML=] namelist-group [,control-information])
```

where the UNIT= is optional and the external unit specifier u is an integer.

In addition control-information is an optional control specification which can be any of the following: may contain one of each of the following specifiers in any order, following the unit number if the optional UNIT specifier keyword is not supplied.

ASYNCHRONOUS= async	async is a character expression specifying whether to allow the data transfer to be done asynchronously. The value specified may be 'YES' or 'NO'.
FMT= format	format is a label of a format statement or a variable containing a format string.
NML= namelist	namelist is a namelist group
ADVANCE= spec	spec is a character expression specifying the access method for file connection as either YES or NO.
END=s	s is an executable statement label for the statement used for processing an end of file condition.
EOR=s	s is an executable statement label for the statement used for processing an end of record condition.
ERR=s	s is an executable statement label for the statement used for processing an error condition.
IOSTAT=ios	ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

REC=rn	rn is a record number to read and must be a positive integer. This is only used for direct access files.
SIZE=n	n is the number of characters read.
iolist	(input list) must either be one of the items in an input list or any other expression.

Description

When a READ statement is executed, the following operations are carried out:

data is transferred from the standard input device to the items specified in the input and format specification.[1]

The data are transferred between the specified destinations in the order specified by the input/output list.

Every item whose value is to be transferred must be defined.

Example

```
 READ(2,110) I,J,K
110 FORMAT(I2, I4, I3)
```

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
accept ch
```

1. If an asterisk (*) is used instead of a format identifier, the list-directed formatting rules apply.

See Also

OPEN, PRINT, WRITE

REAL 90

The REAL statement establishes the data type of a variable by explicitly attaching the name of a variable to a data type. This overrides the implication of data typing by the initial letter of a symbolic name.

Syntax

The syntax for REAL has two forms, a standard Fortran 90/95 attributed form, and the PGI extended form. This section describes both syntax forms.

```
REAL [ ( [ KIND = kind-value
) ] [, attribute-list ::] entity-list
```

REAL permits a KIND specification. Refer to the Fortran 95 Handbook for more syntax details.

attribute-list	is the list of attributes for the character variable.
entity-list	is the list of defined entities.

Syntax Extension§

```
REAL [*n] name [*n] [dimensions] [/clist/] [,
name] [*n] [dimensions][/clist/]...
```

n	is an optional size specification.
name	is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement below for an explanation of array declarators).
clist	is a list of constants that initialize the data, as in a DATA statement.

Description

The REAL type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled.

Note

The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a REAL variable is 4 bytes. With the -Mr8 option, the default size of an REAL variable is 8 bytes.

Example

```
REAL KNOTS
```

RECORD § 77

The RECORD statement defines a user-defined aggregate data item.

Syntax

```
RECORD /structure_name/record_namelist
 [,/structure_name/record_namelist]
 ...
 [,/structure_name/record_namelist]
END RECORD
```

structure_name	is the name of a previously declared structure.
record_namelist	is a list of one or more variable or array names separated by commas.

Description

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, `recordname.fieldname`). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of the COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

Records are allowed in COMMON and DIMENSION statements.

Example

```
STRUCTURE /PERSON/ ! Declare a structure
defining a person
 INTEGER ID
 LOGICAL LIVING
 CHARACTER*5 FIRST, LAST, MIDDLE
 INTEGER AGE
END STRUCTURE
 ! Define population to be an array where each element is of
 ! type person. Also define a variable, me, of type person.
RECORD /PERSON/ POPULATION(2), ME
 ...
ME.AGE = 34 ! Assign values for the variable me
to
ME.LIVING = .TRUE. ! some of the fields.
ME.FIRST = 'Steve'
ME.ID = 542124822
 ...
POPULATION(1).LAST = 'Jones' ! Assign the "LAST" field
of
 ! element 1 of array population.
POPULATION(2) = ME ! Assign all the values
of record
 ! "ME" to the record population(2)
```

RECURSIVE 90

The RECURSIVE statement indicates whether a function or subroutine may call itself recursively.

Syntax

```
RECURSIVE [type-specification] FUNCTION
```

or

```
type-specification RECURSIVE FUNCTION
```

or

```
RECURSIVE SUBROUTINE
```

Type

Non-executable

See Also

FUNCTION, SUBROUTINE

REDIMENSION § 77

The REDIMENSION statement, a CRAY extension to FORTRAN 77, dynamically defines the bounds of a deferred-shape array. After a REDIMENSION statement, the bounds of the array become those supplied in the statement, until another such statement is encountered.

Syntax

```
REDIMENSION name ([lb:]ub[,[lb:]ub]...)
 [,name([lb:]ub[,[lb:]ub]...)]...
```

Where:

name	is the symbolic name of an array.
[lb:]ub	is a dimension declarator specifying the bounds for a dimension (the lower bound lb and the upper bound ub). lb and ub must be integers with ub greater than lb. The lower bound lb is optional; if it is not specified, it is assumed to be 1. The number of dimension declarations must be the same as the number of dimensions in the array.

Example

```
REAL A(:, :)
POINTER (P, A)
P = malloc(12 * 10 * 4)
REDIMENSION A(12, 10)
A(3, 4) = 33.
```

RETURN 77

The RETURN statement causes a return to the statement following a CALL when used in a subroutine, and returns to the relevant arithmetic expression when used in a function.

Syntax

```
RETURN
```

Alternate RETURN

(Obsolescent) The alternate RETURN statement is obsolescent for HPF and Fortran 90/95. Use the CASE statement where possible in new or updated code. The alternate RETURN statement takes the following form:

```
RETURN expression
```

expression	expression is converted to integer if necessary (expression may be of type integer or real). If the value of expression is greater than or equal to 1 and less than or equal to the number of asterisks in the SUBROUTINE or subroutine ENTRY statement then the value of expression identifies the nth asterisk in the actual argument list and control is returned to that statement.

Example

```
 SUBROUTINE FIX (A,B,*,*,C)
40 IF (T) 50, 60, 70
50 RETURN
60 RETURN 1
70 RETURN 2
 END
 PROGRAM FIXIT
 CALL FIX(X, Y, *100, *200, S)
 WRITE(*,5) X, S ! Come here if (T) < 0
 STOP
100 WRITE(*, 10) X, Y ! Come here if (T) = 0
 STOP
200 WRITE(*,20) Y, S ! Come here if (T) > 0
```

REWIND 77

The REWIND statement positions the file at its beginning. The statement has no effect if the file is already positioned at the start or if the file is connected but does not exist.

Syntax

```
REWIND unit
REWIND (unit,list)
```

unit	is an integer value which is the external unit.
list	contains the optional specifiers as follows:
UNIT=unit	unit is the unit specifier.
ERR=errs	errs is an executable statement label for the statement used for processing an error condition. If an error condition occurs execution continues with the statement specified by s.
IOSTAT=ios	ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

Examples

```
REWIND 5
REWIND(2, ERR=30)
REWIND(3, IOSTAT=IOERR)
```

SAVE 77

The SAVE statement retains the definition status of an entity after a RETURN or END statement in a subroutine or function has been executed.

Syntax

```
SAVE [v [, v ]...]
```

v name of array, variable, or common block (enclosed in slashes)

Description

Using a common-block name, preceded and followed by a slash, ensures that all entities within that COMMON block are saved. SAVE may be used without a list, in which case all the allowable entities within the program unit are saved (this has the same effect as using the –Msave command-line option). Dummy arguments, names of procedures and names of entities within a common block may not be specified in a SAVE statement. Use of the SAVE statement with local variables ensures the values of the local variables are retained for the next invocation of the SUBROUTINE or FUNCTION. Within a main program the SAVE statement is optional and has no effect.

When a RETURN or END is executed within a subroutine or function, all entities become undefined with the exception of:

- Entities specified by a SAVE statement

- Entities in blank common or named common
- Entities initially defined which have not been changed in any way

Example

```
PROGRAM SAFE
 .
 CALL KEEP
 .
 SUBROUTINE KEEP
 COMMON /LIST/ TOP, MIDDLE
 INTEGER LOCAL1
 .
 SAVE /LIST/, LOCAL1
```

SELECT CASE 90

The SELECT CASE statement begins a CASE construct.

Syntax

```
[case-name:]SELECT CASE (case-expr)
[ CASE selector [name]
 block] ...
[ CASE DEFAULT [case-name]
 block
END SELECT [case-name]
```

Example

```
SELECT CASE (FLAG)
CASE ( 1, 2, 3 )
 TYPE=1
CASE ( 4:6 )
 TYPE=2
CASE DEFAULT
 TYPE=0
END SELECT
```

SEQUENCE 90

The SEQUENCE statement is a derived type qualifier that specifies the ordering of the storage associated with the derived type. This statement specifies storage for use with COMMON and EQUIVALENCE statements (the preferred method for derived type data sharing is using MODULES).

Note

There is also an HPF SEQUENCE directive that specifies whether an array, common block, or equivalence is sequential or non-sequential. Refer to the PGHPF User's Guide for more information.

Syntax

```
TYPE
 [SEQUENCE]
 type-specification...
END TYPE
```

Example

```
TYPE RECORD
 SEQUENCE
 CHARACTER NAME(25)
 INTEGER CUST_NUM
 REAL COST
END TYPE
```

STOP 77

The STOP statement stops the program's execution and precludes any further execution of the program.

Syntax

```
STOP [character-expression | digits ]
```

STRUCTURE § 77

The STRUCTURE statement, a DEC extension to FORTRAN 77, defines an aggregate data type.

Syntax

```
STRUCTURE [/structure_name/][field_namelist]
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END STRUCTURE
```

structure_name	is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.
field_namelist	is a list of fields having the structure of the associated structure declaration. A field_namelist is allowed only in nested structure declarations.
field_declaration	can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Description

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications. Note that aligning of structure fields is not supported by VAX/VMS Fortran.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware and a structure's storage requirements are therefore machine-dependent. Note that VAX/VMS Fortran does no padding. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

Data initialization can occur for the individual fields.

The UNION and MAP statements are supported.

The following is an example of record and structure usage.

```
STRUCTURE /account/
 INTEGER typetag ! Tag to determine defined map
 UNION
 MAP ! Structure for an employee
 CHARACTER*12 ssn ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP ! Structure for a customer
```

```
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12) ! Items supplied
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

SUBROUTINE 77

The SUBROUTINE statement introduces a subprogram unit. The statements that follow should be laid out in the same order as a PROGRAM module.

Syntax

```
[RECURSIVE] SUBROUTINE name &
 [(argument[,argument...])] &
 [specification-part]
 [execution-part]
 [internal-subspart]
END [SUBROUTINE [name]]
```

name	is the name of the subroutine being declared and must be unique among all the subroutine and function names in the program. name should not clash with any local, COMMON, PARAMETER or ENTRY names.
argument	is a symbolic name, starting with a letter and containing only letters and digits. The type of argument can be REAL, INTEGER, DOUBLE PRECISION, CHARACTER, COMPLEX, or BYTE, etc.
specification-part	is the specification of data types for the subroutine.
execution-part	contains the subprogram's executable statements.
internal-subs-part	contains subprograms defined within the subroutine.

Description

A SUBROUTINE must be terminated by an END statement. The statements and names in the subprogram only apply to the subroutine except for subroutine or function references and the names of COMMON blocks. Dummy arguments may be specified as * which indicates that the SUBROUTINE contains alternate returns.

Recursion is allowed if the –Mrecursive option is used on the command-line and the RECURSIVE prefix is included in the function definition.

Example

```
SUBROUTINE DAXTIM (A, X, Y, N, M, ITER, FP, TOH)
 INTEGER*4 N, M, ITER
 REAL*8 A, X(N,M), Y(N,M), FP, TOH
.
.
.
END SUBROUTINE DAXTIM
```

See Also

PURE, RECURSIVE

TARGET 90

The TARGET specification statement (attribute) specifies that a data type may be the object of a pointer variable (e.g., pointed to by a pointer variable). Likewise, types that do not have the TARGET attribute cannot be the target of a pointer variable.

Syntax

```
TARGET [ :: ] object-name [(array-spec)]
 [, object-name [(array-spec)]]...
```

See Also

ALLOCATABLE, POINTER

THEN 77

The THEN statement is part of a block IF statement and surrounds a series of statements that are conditionally executed.

Syntax

```
IF logical expression THEN
 statements
ELSE IF logical expression THEN
 statements
ELSE
 statements
ENDIF
```

The ELSE IF section is optional and may be repeated any number of times. Other IF blocks may be nested within the statements section of an IF block.

Example

```
IF (I.GT.70) THEN
 M=1
ELSE IF (I.LT.5) THEN
 M=2
ELSE IF (I.LT.16) THEN
 M=3
ENDIF
IF (I.LT.15) THEN
 M = 4
ELSE
 M=5
ENDIF
```

TYPE 77

The TYPE statement begins a derived type data specification or declares variables of a specified user-defined type.

Syntax Type Declaration

```
TYPE (type-name) [ , attribute-list :: ] entity-list
```

Syntax Derived Type Definition

```
TYPE [[ access-spec ] :: ] type-name
 [ private-sequence-statement ] ...
component-definition-statement
 [ component-definition-statement ]...
END TYPE [type-name]
```

FORTRAN 77 Type Statement

```
TYPE
```

The TYPE statement has the same syntax and effect as the PRINT statement. Refer to the PRINT statement for a full description.

UNION § 77

A UNION declaration, a DEC extension to FORTRAN 77, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a UNION statement is as follows:

Syntax

```
UNION
 map_declaration
 [map_declaration]
 ...
 [map_declaration]
END UNION
```

The format of the map_declaration is as follows:

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

field_declaration	where field declaration is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
 INTEGER typetag ! Tag to determine defined map.
 UNION
 MAP ! Structure for an employee
 CHARACTER*12 ssn ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12) ! Items supplied
```

```
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

USE 90

The USE statement gives a program unit access to the public entities or to the named entities in the specified module.

Syntax

```
USE module-name [, rename-list ]
USE module-name, ONLY: [ only-list ]
```

Description

A module-name file has an associated compiled .mod file that is included when the module is used. The .mod file is searched for in the following directories:

Each –I directory specified on the command-line.

The directory containing the file that contains the USE statement (the current working directory.)

The standard include area.

Examples

```
USE MOD1
USE MOD2, TEMP => VAR
USE MOD3, ONLY: RESULTS, SCORES => VAR2
```

Type

Non-executable

See Also

MODULE

VOLATILE § 77

The VOLATILE statement inhibits all optimizations on the variables, arrays and common blocks that it identifies.

Syntax

```
VOLATILE nitem [, nitem ...]
```

nitem	is the name of a variable, an array, or a common block enclosed in slashes.

Description

If nitem names a common block, all members of the block are volatile. The volatile attribute of a variable is inherited by any direct or indirect equivalences, as shown in the example.

Example

```
COMMON /COM/ C1, C2
VOLATILE /COM/, DIR ! /COM/ and
DIR are volatile
EQUIVALENCE (DIR, X) ! X is volatile
EQUIVALENCE (X, Y) ! Y is volatile
```

WHERE 90

The WHERE statement and the WHERE END WHERE construct permit masked assignments to the elements of an array (or to a scalar, zero dimensional array).

Syntax

WHERE Statement

```
WHERE (logical-array-expr) array-variable = array-expr
```

WHERE Construct

```
WHERE (logical-array-expr)
 array-assignments
[ELSE WHERE
 array-assignments ]
END WHERE
```

Description

This construct allows for conditional assignment to an array based on the result of a logical array expression. The logical array expression and the array assignments must involve arrays of the same shape.

Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 ) GRADE = 'P'
WHERE ( SCORE > 60 )
 GRADE = 'P'
ELSE WHERE
 GRADE = 'F'
END WHERE
```

WRITE 90

The WRITE statement is a data transfer output statement.

Syntax

```
WRITE ([unit=] u, [,control-information) [iolist]
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

where the UNIT= is optional and the external unit specifier u is an integer. This may also be a * indicating list-directed output.

In addition to the unit specification, control-information are optional control specifications, and may be any of those listed in the following (there are some limits on the allowed specifications depending on the type of output, for example, non-advancing, direct and sequential):

ADVANCE=spec	spec is a character expression specifying the access method for the write. YES indicates advancing formatted sequential data transfer. NO indicates non-advancing formatted sequential data transfer.
ASYNCHRONOUS= async	async is a character expression specifying whether to allow the data transfer to be done asynchronously. The value specified may be 'YES' or 'NO'.
ERR=s	s is an executable statement label for the statement used for processing an error condition.
[FMT=]format	format is a label of a format statement or a variable containing a format string.
IOSTAT=ios	ios is an integer variable or array element. ios becomes defined with 0 if no error occurs, and a positive integer when there is an error.

[NML=] namelist	namelist is a namelist group
REC=rn	rn is a record number to read and must be a positive integer. This is only used for direct access files.
iolist	iolist must either be one of the items in an input list or any other expression. However a character expression involving concatenation of an operand of variable length cannot be included in an output list unless the operand is the symbolic name of a constant.

Description

When a WRITE statement is executed the following operations are carried out: data is transferred to the standard output device from the items specified in the output list and format specification.[1] The data are transferred between the specified destinations in the order specified by the input/output list. Every item whose value is to be transferred must be defined.

Example

```
 WRITE (6,90) NPAGE
90 FORMAT('1PAGE NUMBER ',I2,16X,'SALES REPORT, Cont.')
```

Non-character Format-specifier§

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4 ch
ch = '()'
accept ch
```

1. If an asterisk (*) is used instead of a format identifier, the list-directed formatting rules apply.

See Also

READ, PRINT

4 Fortran Arrays

Fortran arrays are any object with the dimension attribute. In Fortran 90/95, arrays may be very different from arrays in older versions of Fortran. Arrays can have values assigned as a whole without specifying operations on individual array elements, and array sections can be accessed. Also, allocatable arrays that are created dynamically are available as part of the Fortran 90/95 standard. This chapter describes some of the features of Fortran 90/95 arrays.

Fortran arrays are any object with the dimension attribute. In Fortran 90/95, and in HPF, arrays may be very different from arrays in older versions of Fortran. Arrays can have values assigned as a whole without specifying operations on individual array elements, and array sections can be accessed. Also, allocatable arrays that are created dynamically are available as part of the Fortran 90/95 and HPF standards. Arrays in HPF play a central role in data distribution and data alignment (refer to this chapter and The High Performance Fortran Handbook for details on working with arrays in HPF). This chapter describes some of the features of Fortran 90/95 and HPF arrays.

The following example illustrates valid array operations.

```
REAL(10,10) A,B,C
A=12!Assign 12 to all elements of A
B=3!Assign 3 to all elements of B
C=A+B!Add each element of A to each of B
```

Array Types

Fortran supports four types of arrays: explicit-shape arrays, assumed-shape arrays, deferred-shape arrays and assumed-size arrays. Both explicit-shape arrays and deferred shape arrays are valid in a main program. Assumed shape arrays and assumed size arrays are only valid for arrays used as dummy arguments. Deferred shape arrays, where the storage for the array is allocated during execution, must be declared with either the ALLOCATABLE or POINTER attributes.

Every array has properties of type rank, shape and size. The extent of an array's dimension is the number of elements in the dimension. The array rank is the number of dimensions in the array, up to a maximum of seven. The shape is the vector representing the extents for all dimensions. The size is the product of the extents. For some types of arrays, all of these properties are determined when the array is declared. For other types of arrays, some of these properties are determined when the array is allocated or when a procedure using the array is entered. For arrays that are dummy arguments, there are several special cases.

Allocatable arrays are arrays that are declared but for which no storage is allocated until an allocate statement is executed when the program is running. Allocatable arrays provide Fortran 90/95 and HPF programs with dynamic storage. Allocatable arrays are declared with a rank specified with the ":" character rather than with explicit extents, and they are given the ALLOCATABLE attribute.

Explicit Shape Arrays

Explicit shape arrays are those arrays familiar to FORTRAN 77 programmers. Each dimension is declared with an explicit value. There are two special cases of explicit arrays. In a procedure, an explicit array whose bounds are passed in from the calling program is called an automatic-array. The second special case, also found in a procedure, is that of an adjustable-array which is a dummy array where the bounds are passed from the calling program.

Assumed Shape Arrays

An assumed shape array is a dummy array whose bounds are determined from the actual array. Intrinsics called from the called program can determine sizes of the extents in the called program's dummy array.

Deferred Shape Arrays

A deferred shape array is an array that is declared, but not with an explicit shape. Upon declaration, the array's type, its kind, and its rank (number of dimensions) are determined. Deferred shape arrays are of two varieties, allocatable arrays and array pointers.

Assumed Size Arrays

An assumed size array is a dummy array whose size is determined from the corresponding array in the calling program. The array's rank and extents may not be declared the same as the original array, but its total size (number of elements) is the same as the actual array. This form of array should not need to be used in new Fortran programs.

Array Specification

Arrays may be specified in either of two types of data type specification statements, attribute-oriented specifications or entity-oriented specifications. Arrays may also optionally have data assigned to them when they are declared. This section covers the basic form of entity-based declarations for the various types of arrays. Note that all the details of array passing for procedures are not covered here; refer to The Fortran 95 Handbook for complete details on the use of arrays as dummy arguments.

Explicit Shape Arrays

Explicit shape arrays are defined with a specified rank, each dimension must have an upper bound specified, and a lower bound may be specified. Each bound is explicitly defined with a specification of the form:

```
[lower-bound:] upper-bound
```

An array has a maximum of seven dimensions. The following are valid explicit array declarations:

```
INTEGER NUM1(1,2,3)!Three dimensions
INTEGER NUM2(-12:6,100:1000)!Two dimensions with !lower and upper
bounds
INTEGER NUM3(0,12,12,12)!Array of size 0
INTEGER NUM3(M:N,P:Q,L,99)!Array with 4 dimensions
```

Assumed Shape Arrays

An assumed shape array is always a dummy argument. An assumed shape array has a specification of the form:

```
[lower-bound] :
```

The number of colons (:) determines the array's rank. An assumed shape array cannot be an ALLOCATABLE or POINTER array.

Deferred Shape Arrays

An deferred shape array is an array pointer or an allocatable array. An assumed shape array has a specification determines the array's rank and has the following form for each dimension:

```
:
```

For example:

```
INTEGER, POINTER::NUM1(:,:,:,:)
INTEGER, ALLOCATABLE::NUM2(:)
```

Assumed Size Arrays

An assumed size array is a dummy argument with an assumed size. The array's rank and bounds are specified with a declaration that has the following form:

```
[explicit-shape-spec-list ,][lower-bound
:]*
```

For example:

```
SUBROUTINE YSUM1(M,B,C)
 INTEGER M
 REAL, DIMENSION(M,4,5,*) :: B,C
```

Array Subscripts and Access

There are a variety of ways to access an array in whole or in part. Arrays can be accessed, used, and assigned to as whole arrays, as elements, or as sections. Array elements are the basic access method, for example:

```
INTEGER, DIMENSION(3,11) :: NUMB
NUMB(3,1)=5
```

This assigns the value 5 to element 3,1 of NUMB

The array NUMB may also be accessed as an entire array:

```
NUMB=5
```

This assigns the value 5 to all elements of NUMB.

Array Sections and Subscript Triplets

Another possibility for accessing array elements is the array section. An array section is an array accessed by a subscript that represents a subset of the entire array's elements and is not an array element. An array section resulting from applying a subscript list may have a different rank than the original array. An array section's subscript list consists of subscripts, subscript triplets, and/or vector subscripts. For example using a subscript triplet and a subscript:

```
NUMB(:,3)=6
```

assigns the value 6 to all elements of NUMB with the second dimension of value 3 (NUMB(1,3), NUMB(2,3), NUMB(3,3)). This array section uses the array subscript triplet and a subscript to access three elements of the original array. This array section could also be assigned to a rank one array with three elements, for example:

```
INTEGER(3,11) NUMB
INTEGER(3) NUMC
NUMB(:,3)=6
NUMC=NUMB(:,3)
```

Note that NUMC is rank 1 and NUMB is rank 2. This array section assignment illustrates how NUMC, an the array section of NUMB, has a shape that due to the use of the subscript 3, is of a different rank than the original array.

The general form for an array's dimension with a vector subscript triplet is:

```
[subscript] : [subscript] [:stride]
```

The first subscript is the lower bound for the array section, the second is the upper bound and the third is the stride. The stride is by default one. If all values except the : are omitted, then all the values for the specified dimensions are included in the array section. For example, using NUMB above:

```
NUMB(1:3:2,3)=7
```

assigns the value 7 to the elements NUMB(1,3) and NUMB(3,3).

Array Sections and Vector Subscripts

Vector-valued subscripts specify an array section by supplying a set of values defined in a one dimensional array (vector) for a dimension or several dimensions of an array section. For example:

```
INTEGER J(2), I(2)
INTEGER NUMB(3,6)
I=(/1,2/)
J=(/2,3/)
NUMB(J,I)=7
```

This array section uses the vectors I and J to assign the value 7 to the elements NUMB(2,1), NUMB(2,2), NUMB(3,1), NUMB(3,2).

Array Constructors

An array constructor can be used to assign values to an array. Array constructors form one-dimensional vectors to supply values to a one-dimensional array, or one dimensional vectors and the RESHAPE function to supply values to arrays with more than one dimension.

Array constructors can use a form of implied DO similar to that in a DATA statement. For example:

```
INTEGER DIMENSION(4):: K = (/1,2,7,11/)
INTEGER DIMENSION(20):: J = (/(I,I=1,40,2)/)
```

CM Fortran Extensions

The ARRAY Attribute §

The PGHPF compiler provides several extensions for handling arrays. The compiler handles the CM Fortran attribute ARRAY. The ARRAY attribute is similar to the DIMENSION attribute. Refer to for more details on the ARRAY statement.

Array Constructors Extensions §

The PGHPF compiler supports an extended form of the array constructor specification. In addition to the (/ ../) specification for array constructors, PGHPF supports the notation where [and] begin and end, respectively, an array constructor.

In addition, an array constructor item may be a 'subscript triplet' in the form of an array section where the values are assigned to the array:

```
lower-bound : upper-bound [ : <stride> ]
```

For the values i : j : k the array would be assigned values i, i+k, i+2k, ..., j. If k is not present, stride is assumed to be 1.

For example:

```
INTEGER, DIMENSION(20):: K = [1:40:2]
```

5 Input and Output Formatting

Input, output, and format statements provide the means for transferring data to or from files. Data is transferred as records to or from files. A record is a sequence of data which may be values or characters and a file is a sequence of such records. A file may be internal, that is, held in memory, or external such as those held on disk. To access an external file a formal connection must be made between a unit, for example a disk file, and the required file. An external unit must be identified either by a positive integer expression, the value of which indicates a unit, or by an asterisk (*) which identifies a standard input or output device.

This chapter describes the types of input and output available and provides examples of input, output and format statements. There are four types of input/output used to transfer data to or from files: unformatted, formatted, list directed, and namelist.

unformatted data is transferred between the item(s) in the input/output list (iolist) and the current record in the file. Exactly one record may be read or written.

formatted data is edited to conform to a format specification, and the edited data is transferred between the item or items in the iolist, and the file. One or more records may be read or written. Non-advancing formatted data transfers are a variety of formatted I/O where a portion of a data record is transferred with each input/output statement.

list directed input/output is an abbreviated form of formatted input/output that does not use a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to or from the file, using a default, and not necessarily accurate format specification.

namelist input/output is a special type of formatted data transfer; data is transferred between a named group (namelist group) of data items and one or more records in a file.

File Access Methods

You can access files using one of two methods, sequential access, or direct access (random access). The access method is determined by the specifiers supplied when the file is opened using the OPEN statement. Sequential access files are accessed one after the other, and are written in the same manner. Direct access files are accessed by specifying a record number for input, and by writing to the currently specified record on output.

Files may contain one of two types of records, fixed length records or variable length records. To specify the size of the fixed length records in a file, use the RECL specifier with the OPEN statement. RECL sets the record length in bytes.[1] RECL can only be used when access is direct.

A record in a variable length formatted file is terminated with \n. A record in a variable length unformatted file is preceded and followed by a word indicating the length of the record.

Standard Preconnected Units

Certain input and output units are predefined, depending on the value of compiler options. The PGI Fortran compilers –Mdefaultunit option tells the compiler to treat "*" as a synonym for standard input for reading and standard output for writing. When the option is –Mnodefaultunit, the compiler treats "*" as a synonym for unit 5 on input and unit 6 on output.

Opening and Closing Files

The OPEN statement establishes a connection to a file. OPEN allows you to do any of the following

- Connect an existing file to a unit.
- Create and connect a file to a unit.
- Create a file that is preconnected.
- Establish the access method and record format for a connection.

OPEN has the form:

```
OPEN (list)
```

where list contains a unit specifier of the form:

```
[UNIT=] u
```

where u, an integer, is the external unit specifier.

---

1. The units depend on the value of the FORTRANOPT environment variable. If the value is vaxio, then the record length is in units of 32-bit words. If FORTRANOPT is not defined, or its value is something other than vaxio, then the record length is always in units of bytes.

In addition list may contain one of each of the specifiers shown in Table 5-1 , “OPEN Specifiers”.

Direct Access Files

If a file is connected for direct access using OPEN with ACCESS='DIRECT', the record length must be specified using RECL=, and optionally one of each of the other specifiers may be used.

Any file opened for direct access must be via fixed length records.

In the following example a new file, book.dat, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters. Numeric values will have blanks ignored and the variable E1 will be assigned some positive value if an error condition exists when the OPEN statement is executed; execution will then continue with the statement labeled 20. If no error condition pertains, E1 is assigned the value 0 and execution continues with the statement following the OPEN statement.

```
OPEN(12,IOSTAT=E1,ERR=20,FILE='book.dat',BLANK='NULL',
+ACCESS='DIRECT',RECL=98,FORM='FORMATTED',STATUS='NEW')
```

Closing a File

Close a unit by specifying the CLOSE statement from within any program unit. If the unit specified does not exist or has no file connected to it, the CLOSE statement has no effect.

Provided the file is still in existence, it may be reconnected to the same or a different unit after the execution of a CLOSE statement. An implicit CLOSE is executed when a program stops.

The CLOSE statement terminates the connection of the specified file to a unit.

```
CLOSE ([UNIT=] u [,IOSTAT=ios] [,ERR= errs ]
[,STATUS= sta] [,DISPOSE= sta] [,DISP= sta])
```

CLOSE takes the status values IOSTAT, ERR, and STATUS, similar to those described in Table 5-1 , “OPEN Specifiers”. In addition, CLOSE allows the DISPOSE or DISP specifier which can take a status value sta which is a character string, where case is insignificant, specifying the file status (the same keywords are used for the DISP and DISPOSE status). Status can be KEEP or DELETE. KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement, conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

Table 5-1: OPEN Specifiers

Specifier	Description
ACCESS=acc	Where acc is a character string specifying the access method for file connection as DIRECT (random access) or SEQUENTIAL. The default is SEQUENTIAL.
ACTION=act	Where act is a character string specifying the allowed actions for the file and is one of READ, WRITE, or READWRITE.
BLANK=blnk	Where blnk is a character string which takes the value NULL or ZERO: NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. The default is NULL. This specifier must only be used when a file is connected for formatted input/output.
DELIM=del	Specify the delimiter for character constants written by a list-directed or namelist-formatted statement. The options are APOSTROPHE, QUOTE, and NONE.
ERR=errs	An error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs, execution continues with the statement specified by errs.*2*
FILE=fin	Where fin is a character string defining the file name to be connected to the specified unit.
FORM=fm	Where fm is a character string specifying whether the file is being connected for FORMATTED or UNFORMATTED output respectively. The default is FORMATTED.
IOSTAT=ios	Input/output status specifier where ios is an integer scalar memory reference. If this is included in list, ios becomes defined with 0 if no error exists or a positive integer when there is an error condition.[1]
PAD=padding	Specifies whether or not to use blank padding for input items. The padding values are YES and NO. The value NO requires that the input record and the input list format specification match.

Specifier	Description
POSITION=pos	Specifies the position of an opened file. ASIS indicates the file position remains unchanged. REWIND indicates the file is to be rewound, and APPEND indicates the file is to positioned just before an end-of-file record, or at its terminal point.
RECL=rl	Where rl is an integer which defines the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.
STATUS=sta	The file status where sta is a character expression: it can be NEW, OLD, SCRATCH, REPLACE or UNKNOWN. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default is UNKNOWN.

1. If IOSTAT and ERR are not present, the program terminates if an error occurs.

A unit may be the subject of a CLOSE statement from within any module. If the unit specified does not exist or has no file connected to it, the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

In the following example the file on UNIT 6 is closed and deleted.

```
CLOSE(UNIT=6,STATUS='DELETE')
```

Data Transfer Statements

Once a unit is connected, either using a preconnection, or by executing an OPEN statement, data transfer statements may be used. The available data transfer statements include: READ, WRITE, and PRINT. The general form for these data transfer statements is shown in Chapter 3, "Fortran Statements"; refer to that section for details on the READ, WRITE and PRINT statements and their valid I/O control specifiers.

Unformatted Data Transfer

Unformatted data transfer allows data to be transferred between the current record and the items specified in an input/output list. Use OPEN to open a file for unformatted output:

```
OPEN (2, FILE='new.dat', FORM='UNFORMATTED')
```

The unit specified must be an external unit.

After data is transferred, the file is positioned after the last record read or written, if there is no error condition or end-of-file condition set. Unformatted data transfer cannot be carried out if the file is connected for formatted input/output.

The following example shows an unformatted input statement:

```
READ (2, ERR=50) A, B
```

On output to a file connected for direct access, the output list must not specify more values than can fit into a record. If the values specified do not fill the record the rest of the record is undefined.

On input, the file must be positioned so that the record read is either:

An unformatted record or an endfile record.

The number of values required by the input list in the input statement must be less than or equal to the number of values in the record being read. The type of each value in the record must agree with that of the corresponding entity in the input list. However one complex value may correspond to two real list entities or vice versa. If the input list item is of type CHARACTER, its length must be the same as that of the character value

In the event of an error condition, the position of the file is indeterminate.

Formatted Data Transfer

During formatted data transfer, data is edited to conform to a format specification, and the edited data is transferred between the items specified in the input or output statement's iolist and the file; the current record is read or written and, possibly, so are additional records. On input, the file must be positioned so that the record read is either a formatted record or an endfile record. Formatted data transfer is prohibited if the file is connected for unformatted input/output.

For variable length record formatted input, each newline character is interpreted as a record separator. On output, the I/O system writes a newline at the end of each record. If a program writes a newline itself, the single record containing the newline will appear as two records when read or backspaced over. The maximum allowed length of a record in a variable length record formatted file is 2000 characters.

Implied DO List Input Output List

An implied DO list takes the form

```
(iolist,do-var=var1,var2,var3)
```

where the items in iolist are either items permissible in an input/output list or another implied DO list. The value do-var is an INTEGER, REAL or DOUBLE PRECISION variable and var1, var2 and var3 are arithmetic expressions of type INTEGER, REAL or DOUBLE PRECISION. Generally, do-var, var1, var2 and var3 are of type INTEGER. Should iolist occur in an input statement, the do-var cannot be used as an item in iolist. If var3 and the preceding comma are omitted, the increment takes the value 1. The list items are specified once for each iteration of the DO loop with the DO-variable being substituted as appropriate.

```
REAL C(6),D(6)
DATA OXO,(C(I),I=7,9),TEMP,(D(J),J=1,2)/4*0.0,3*10.0/
```

In the above example OXO, C(7), C(8) and C(9) are set to 0.0 with TEMP, D(1) and D(2) being set to 10.0. In the next example:

```
READ *,A,B,(R(I),I=1,4),S
```

has the same effect as

```
READ *,A,B,R(1),R(2),R(3),R(4),S
```

Format Specifications

Format requirements may be given either in an explicit FORMAT statement or alternatively, as fields within an input/output statement (as values in character variables, arrays or other character expressions within the input/output statement).

When a format identifier in a formatted input/output statement is a character array name or other character expression, the leftmost characters must be defined with character data that constitute a format specification when the statement is executed. A character format specification is enclosed in parentheses. Blanks may precede the left parenthesis. Character data may follow the right-hand

parenthesis and has no effect on the format specification. When a character array name is used as a format identifier, the length of the format specification can exceed the length of the first element of the array; a character array format specification is considered to be an ordered concatenation of all the array elements. When a character array element is used as a format identifier the length must not exceed that of the element used.

The FORMAT statement has the form:

```
FORMAT (list-of-format-requirements)
```

The list of format requirements can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.
- Non-repeatable editor commands.
- A format specification list enclosed in parentheses, optionally preceded by an integer constant which defines the number of repeats.

Each action of format control depends on a FORMAT specified edit code and the next item in the input/output list used. If an input/output list contains at least one item, there must be at least one repeatable edit code in the format specification. An empty format specification FORMAT( ) can only be used if no list items are specified. In such a case, one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. When a repeat specification is used, the appropriate item is repeated the required number of times.

Each repeatable edit code has a corresponding item in the iolist; however when a list item is of type complex two edit codes of F, E, D or G are required. The edit codes P, X, T, TL, TR, S, SP, SS, H, BN, BZ, /, : and apostrophe act directly on the record and have no corresponding item in the input/output list.

The file is positioned after the last character read or written when the edit codes I, F, E, D, G, L, A, H or apostrophe are processed. If the specified unit is a printer then the first character of the record is used to control the vertical spacing as shown in the following table:

Table 5-2: Format Character Controls for a Printer

Character	Vertical Spacing
Blank	One line
0	Two lines
1	To first line on next page
+	No advance

A Format Control – Character Data

The A specifier transfers characters. The A can optionally be followed by a field width w. When w is not specified, the width is determined by the size of the data item.

On output, if l is the length of the character item and w is the field width, then the following rules apply:

- If w > l,w – l blanks before the character.
- If w < l, leftmost w characters.

On input, if l is the length of the character I/O item and w is the field width, then the following rules apply:

- If w > l, rightmost l characters from the input filed.
- If w < l, leftmost w characters from the input filed and followed by l – w blanks.

You can also use the A format specifier to process data types other than CHARACTER. For types other than CHARACTER, the number of characters supplied for input/output will equal the size in bytes of the data allocated to the data type. For example, an INTEGER*4 value is represented with 4 characters and a LOGICAL*2 is represented with 2 characters.

The following shows a simple example that reads two CHARACTER arrays from the file data.src:

```
  CHARACTER STR1*8, STR2*12
  OPEN(2, FILE='data.src')
  READ(2, 10) STR1, STR2
10 FORMAT ( A8, A12 )
```

B Format Control – Binary Data

The B field descriptor transfers binary values and can be used with any integer data type. The edit descriptor has the form:

```
Bw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) binary characters only (0 or 1). An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the B field descriptor transfers the binary values of the corresponding I/O list element, right-justified, to an external field that is w characters long. If the value to be transmitted 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. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary. Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

D Format Control – Real Double Precision Data with Exponent

The D specifier transfers real values for double precision data with a representation for an exponent. The form of the D specifier is:

```
Dw.d
```

where w is the field width and d the number of digits in the fractional part.

For input, the same conditions apply as for the F specifier described later in this chapter.

For output, the scale factor k controls the decimal normalization. The scale factor k is the current scale factor specified by the most recent P format control; if one hasn't been specified, the default is zero (0). If `-d < k <= 0`, the output file contains leading zeros and `d-|k|` significant digits after the decimal point. If `0 < k < d+2`, there are exactly `|k|` significant digits to the left of the decimal point and `d-k+1` significant digits to the right of the decimal point. Other values of k are not allowed.

For example:

```
  DOUBLE PRECISION VAL1
  VAL1 = 141.8835
  WRITE( *, 20) VAL1
20 FORMAT ( D10.4 )
```

produces the following:

```
0.1418D+03
```

E Format Control – Real Single Precision Data with Exponent

The E specifier transfers real values for single precision data with an exponent. The E format specifier has two basic forms:

```
Ew.d
Ew.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For input the same conditions apply as for F editing. For output the scale factor controls the decimal normalization as in the D specifier.

EN Format Control

The EN specifier transfers real values using engineering notation.

```
ENw.d
ENw.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

On output, the number is in engineering notation where the exponent is divisible by 3 and the absolute value of the significand is 1000 > |significand | 1. This format is the same as the E format descriptor, except for restrictions on the size of the exponent and the significand.

ES Format Control

The ES specifier transfers real values in scientific notation. The ES format specifier has two basic forms:

```
ESw.d
ESw.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For output, the scale factor controls the decimal normalization as in the D specifier.

On output, the number is presented in scientific notation, where the absolute value of the significand is 10> | significand | 1.

F Format Control - Real Single Precision Data

The F specifier transfers real values. The form of the F specifier is:

```
Fw.d
```

where w is the field width and d is the number of digits in the fractional part.

On input, if the field does not contain a decimal digit or an exponent, right-hand d digits, with leading zeros, are interpreted as being the fractional part.

On output, a leading zero is only produced to the left of the decimal point if the value is less than one.

G Format Control

The G format specifier provides generalized editing of real data. The G format has two basic forms:

```
Gw.d
Gw.dEe
```

The specifier transfers real values; it acts like the F format control on input and depending on the value's magnitude, like E or F on output. The magnitude of the data determines the output format. For details on the actual format used, based on the magnitude, refer to the ANSI FORTRAN Standard (Section 13.5.9.2.3 G Editing).

I Format Control – Integer Data

The I format specifier transfers integer values. The I format specifier has two basic forms:

```
Iw
Iw.m
```

where w is the field width and m is the minimum filed width on output, including leading zeros. If present, m must not exceed width w.

On input, the external field to be input must contain (unsigned) decimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the I format descriptor transfers the decimal values of the corresponding I/O list element, right-justified, to an external field that is w characters long. If the value to be transmitted 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. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary. Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

L Format Control – Logical Data

The L format control transfers logical data of field width w:

```
Lw
```

On input, the list item will become defined with a logical value; the field consists of optional blanks, followed by an optional decimal point followed by T or F. The values .TRUE. or .FALSE. may also appear in the input field

The output field consists of w-1 blanks followed by T or F as appropriate.

Quote Format Control

Quote editing prints a character constant. The format specifier writes the characters enclosed between the quotes and cannot be used on input. The field width is that of the characters contained within quotes (you can also use apostrophes to enclose the character constant).

To write an apostrophe (or quote), use two consecutive apostrophes (or quotes).

For example:

```
 WRITE ( *, 101)
101 FORMAT ( 'Print an apostrophe '' and end.')
```

Produces:

```
Print an apostrophe ' and end.
```

Similarly, you can use quotes, for example:

```
 WRITE ( *, 102)
102 FORMAT ( "Print a line with a "" and end.")
```

Produces:

```
Print a line with a " and end.
```

BN Format Control – Blank Control

The BN and BZ formats control blank spacing. BN causes all embedded blanks except leading blanks in numeric input to be ignored, which has the effect of right-justifying the remainder of the field. Note that a field of all blanks has the value zero. Only input statements and I, F, E, D and G editing are affected.

BZ causes all blanks except leading blanks in numeric input to be replaced by zeros. Only input statements and I, F, E, D and G editing are affected.

H Format Control – Hollerith Control

The H format control writes the n characters following the H in the format specification and cannot be used on input.

The basic form of this format specification is:

```
nHc1cn...
```

where n is the number of characters to print and c1 through cn are the characters to print.

O Format Control Octal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with an integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long. If the value to be transmitted 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. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary. Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

P Format Specifier – Scale Control

The P format specifier

```
kP
```

is the scale factor format which is applied as follows.

- With F, E, D and G editing on input and F editing on output, the external number equals the internal number multiplied by 10**k . If there is an exponent in the field on input, editing with F, E, D and G the scale factor has no effect.
- On output with E and D editing, the basic real constant part of the number is multiplied by 10**k and the exponent reduced by k ; with G editing the effect of the scale factor is suspended unless the size of the datum to be edited is outside the range permitted for F editing. If E editing is required, the scale factor has the same effect as with E output editing.

The following is an example using a scale factor.

```
DIMENSION
A(6)
DO 10 I = 1,6
10A(I) = 25.
TYPE 100,A
100FORMAT(' ',F8.2,2PF8.2,F8.2)
```

produces:

```
25.00
2500.00 2500.00 2500.00 2500.00 2500.00
```

Note that the effect of the scale factor continues until another scale factor is used.

Q Format Control - Quantity

The Q edit descriptor calculates the number of characters remaining in the input record and stores that value in the next I/O list item. On output, the Q descriptor skips the next I/O item.

S Format Control – Sign Control

The S format specifier restores the default processing for writing a plus; the default is SS processing.

SP forces the processor to write a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

SS stops the processor from writing a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

T , TL and X Format Controls – Spaces and Tab Controls

The T specifier controls which portion of a record in an iolist value is read from or written to a file. The general form, which specifies that the nth value is to be written to or from a record, is as follows:

```
Tn
```

The TL form specifies the relative position to the left of the data to be read or written:

```
TLn
```

and specifies that the nth character to the left of the current position is to be written to or from the record. If the current position is less than or equal to n, the transmission will begin at position one of the record.

The TR form specifies the relative position to the right of the data to be read or written:

```
TRn
```

and specifies that the nth character to the right of the current position is to be written to or from the record.

The X control specifies a number of characters to skip forward and that the next character to be written to or from is n characters forward from the current position:

```
nX
```

The following example uses the X format specifier:

```
 NPAGE = 19
 WRITE ( 6, 90) NPAGE
90 FORMAT('1PAGE NUMBER ,I2, 16X, 'SALES REPORT, Cont.')
```

produces:

```
PAGE NUMBER 19 SALES REPORT, Cont.
```

The following example shows use of the T format specifier:

```
 PRINT 25
25 FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

produces:

```
 COLUMN 1 COLUMN 2
```

Z Format Control Hexadecimal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with any integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long. If the value to be transmitted 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. If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary. Note that if m is zero, and the internal representation is zero, the external field is blank-filled.

Slash Format Control / – End of Record

The slash (/) control indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the rest of the current record is skipped and the file positioned at the start of the next record.

On output a new record is created which becomes the last and current record. For an internal file connected for direct access, the record is filled with blank characters. For a direct access file, the record number is increased by one and the file is positioned at the start of the record.

Multiple slashes are permitted, thus multiple records are skipped.

The : Format Specifier – Format Termination

The (:) control terminates format control if there are no more items in the input/output list. It has no effect if there are any items left in the list.

$ Format Control

The $ field descriptor allows the programmer to control carriage control conventions on output. It is ignored on input. For example, on terminal output, it can be used for prompting.

The form of the $ field descriptor is:

```
$
```

Variable Format Expressions ,<expr>

Variable format expressions are supported. They provide a means for substituting run-time expressions for the field width, other parameters for the field and edit descriptors in a FORMAT statement (except for the H field descriptor and repeat counts).

Variable format expressions are enclosed in "<" and ">" and are evaluated each time they are encountered in the scan of a format. If the value of a variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Non-advancing Input and Output

Non-advancing input/output is character-oriented and applies to sequential access formatted external files. The file position is after the last character read or written and not automatically advanced to the next record.

For non-advancing input/output, use the ADVANCE='NO' specifier. Two other specifiers apply to non-advancing IO: EOR applies when end of record is detected and SIZE returns the number of characters read.

List-directed formatting

List-directed formatting is an abbreviated form of input/output that does not require the use of a format specification. The type of the data is used to determine how a value is read/written. On output, it will not always be accurate enough for certain ranges of values. The characters in a list-directed record constitute a sequence of values which cannot contain embedded blanks except those permitted within a character string. To use list-directed input/output formatting, specify a * for the list of format requirements. For example, the following example uses list-directed output:

```
READ( 1, * ) VAL1, VAL2
```

List-directed input

The form of the value being input must be acceptable for the type of item in the iolist. Blanks must not be used as zeros nor be embedded in constants except in a character constant or within a type complex form contained in parentheses.

Table 5-3: List Directed Input Values

Input List Type	Form
Integer	A numeric input field.
Real	A numeric input field suitable for F editing with no fractional part unless a decimal point is used.
Double precision	Same as for real.
Complex	An ordered pair of numbers contained within parentheses as shown (real part, imaginary part).
Logical	A logical field without any slashes or commas.
Character	A non-empty character string within apostrophes. A character constant can be continued on as many records as required. Blanks, slashes and commas can be used.

A null value has no effect on the definition status of the corresponding iolist item. A null value cannot represent just one part of a complex constant but may represent the entire complex constant. A slash encountered as a value separator stops the execution of that input statement after the assignment of the previous value. If there are further items in the list, they are treated as if they are null values.

Commas may be used to separate the input values. If there are consecutive commas, or if the first non-blank character of a record is a comma, the input value is a null value. Input values may also be repeated.

In the following example of list-directed formatting, assume that

```
A= -1.5
K= 125
```

and all other variables are undefined. When the statement below reads in the list from the input file:

```
READ * I, J, X, Y, Z, A, C, K
```

where the file contains the following record:

```
10,-14,25.2,-76,313,,29/
```

The variables are assigned the following values by the list-directed input/output mechanism:

```
I=10
J=-14
X=25.2
Y=-76.0
Z=313.0
A=-1.5
C=29
K=125.
```

Note that the value for A does not change because the input record is null (consecutive commas). No input is read for K, so it assumes null and K retains its previous value (the / terminates the input).

List-directed output

List directed input/output is an abbreviated form of formatted input/output that does not require the use of a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to or from the file, using a default, and not necessarily accurate format specification. The data type of each item appearing in the iolist is formatted according to the rules in the following table:

Table 5-4: Default List Directed Output Formatting

Data Type	Default Formatting
BYTE	I5
INTEGER*2	I7
INTEGER*4	I12
INTEGER*8	I24
LOGICAL*1	I5 (L2[1])
LOGICAL*2	L2
LOGICAL*4	L2
LOGICAL*8	L2
REAL*4	G15.7e2
REAL*8	G25.16e3
COMPLEX*8	(G15.7e2, G15.7e2)
COMPLEX*16	(G25.16e3, G25.16e3)
CHAR *n	An

1. This format is applied when the option –Munixlogical is selected when compiling.

The length of a record is less than 80 characters; if the output of an item would cause the length to exceed 80 characters, a new record is created.

Issues to note when using list-directed output:

- New records may begin as necessary.
- Logical output constants are T for true and F for false.

- Complex constants are contained within parentheses with the real and imaginary parts separated by a comma.
- Character constants are not delimited by apostrophes and have each internal apostrophe (if any are present) represented externally by one apostrophe.
- Each output record begins with a blank character to provide carriage control when the record is printed.
- A typeless value output with list-directed I/O is output in hexadecimal form by default. There is no other octal or hexadecimal capability with list-directed I/O.

Commas in External Field

Use of the comma in an external field eliminates the need to "count spaces" to have data match format edit descriptors. The use of a comma to terminate an input field and thus avoid padding the field is fully supported.

Namelist Groups

The NAMELIST statement allows for the definition of namelist groups. A namelist group allows for a special type of formatted input/output, where data is transferred between a named group of data items defined in a NAMELIST statement and one or more records in a file.

The general form of a namelist statement is:

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

where:

group-name	is the name of the namelist group.
namelist	is the list of variables in the namelist group.

Namelist Input

Namelist input is accomplished using a READ statement by specifying a namelist group as the input item. The following statement shows the format:

```
READ ([unit=] u, [NML=] namelist-group [,control-information])
```

One or more records are processed which define the input for items in the namelist group.

The records are logically viewed as follows:

```
$group-nameitem=value[,item=value].... $ [END]
```

The following rules describe these input records:

1. The start or end delimiter ($) may be an ampersand (&).
2. The start delimiter must begin in column 2 of a record.
3. The group-name begins immediately after the start delimiter.
4. The spaces or tabs may not appear within the group-name, within any item, or within any constants.
5. The value may be constants as are allowed for list directed input, or they may be a list of constants separated by commas (,). A list of items is used to assign consecutive values to consecutive elements of an array.
6. Spaces or tabs may precede the item, the = and the constants.
7. Array items may be subscripted.
8. Character items may be substringed.

Namelist Output

Namelist output is accomplished using a READ statement by specifying a namelist group as the output item. The following statement shows the format:

```
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

The records output are logically viewed as follows:

```
$group-name
item = value
$ [END]
```

The following rules describe these output records:

1. One record is output per value.
2. Multiple values are separated by a comma (,).

3. Values are formatted according to the rules of the list-directed write. Exception: character items are delimited by an apostrophe (').

4. An apostrophe (') or a quote (") in the value is represented by two consecutive apostrophes or quotes.

6 Fortran Intrinsics

This chapter lists the FORTRAN 77 and Fortran 90/95 intrinsics and subroutines, intrinsics defined in the HPF Language Specification, and CM Fortran intrinsics. Color-coding is used to highlight the different Fortran versions. The colors used are as follows:

- FORTRAN 77 is shown in black
- Fortran 90/95 is shown in blue
- HPF is shown in red
- CM Fortran is shown in green

FORTRAN 77 and Fortran 90/95 Intrinsics by Category

The tables in this section contain the FORTRAN 77 and Fortran 90/95 intrinsics supported by the PGF77 and PGF95 compilers. Intrinsics are categorized by functionality and alphabetized by generic name within each table. All FORTRAN 77 intrinsics are supported and are detailed in the ANSI FORTRAN Standard.

To simplify the tables in this section, two groups of intrinsic types have been given the following abbreviated group names:

NUMERIC	INTEGER, REAL, COMPLEX
NONCHAR	LOGICAL, INTEGER, REAL, COMPLEX

Table 6-1: Numeric Functions

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
ABS	Absolute Value	1		NUMERIC	NUMERIC
		1	IIABS	INTEGER*2	INTEGER*2
		1	JIABS	INTEGER*4	INTEGER*4
		1	KIABS	INTEGER*8	INTEGER*8
		1	ABS	REAL*4	REAL*4
		1	DABS	REAL*8	REAL*8
		1	CAB	COMPLEX*8	COMPLEX*8
		1	CDABS	COMPLEX*16	COMPLEX*16
AIMAG	Imaginary Part of Complex Number	1	AIMAG	COMPLEX*8	REAL*4
		1	DIMAG	COMPLEX*16	REAL*8
AINT	Truncation	1	AINT	REAL*4	REAL*4
		1	DINT	REAL*8	REAL*8
ANINT	Nearest Whole Number	1	ANINT	REAL*4	REAL*4
		1	DNINT	REAL*8	REAL*8
CEILING	Next Whole Number	1		REAL	INTEGER
		2		REAL, INTEGER	INTEGER
CMPLX	Convert to COMPLEX*8	1		NUMERIC	COMPLEX*8
		2		INTEGER, INTEGER	COMPLEX*8
		2		REAL, REAL	COMPLEX*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
CONJG	Complex Conjugate	1	CONJG	COMPLEX*8	COMPLEX*8
		1	DCONJG	COMPLEX*16	COMPLEX*16
DBLE	Convert to REAL*8	1		NUMERIC	REAL*8
		1	DFLOTI	INTEGER*2	REAL*8
		1	DFLOAT	INTEGER*4	REAL*8
		1	DFLOTJ	INTEGER*4	REAL*8
		1	DFLOTK	INTEGER*8	REAL*8
		1	DREAL	COMPLEX*16	REAL*8
DCMPLX	Convert to COMPLEX*16	1		NUMERIC	COMPLEX*16
		2		INTEGER,INTEGER	COMPLEX*16
		2		REAL, REAL	COMPLEX*16
DIM	Positive Difference	2	IIDIM	INTEGER*2, INTEGER*2	INTEGER*2
		2	JIDIM	INTEGER*4, INTEGER*4	INTEGER*4
		2	KIDIM	INTEGER*8, INTEGER*8	INTEGER*8
		2	DIM	REAL*4, REAL*4	REAL*4
		2	DDIM	REAL*8, REAL*8	REAL*8
FLOOR	Previous integer	1		REAL	INTEGER
		2		REAL, INTEGER	INTEGER

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
IINT	Truncation	1		NUMERIC	INTEGER*2
		1	IINT	REAL*4	INTEGER*2
		1	IIFIX	REAL*4	INTEGER*2
		1	IIDINT	REAL*8	INTEGER*2
ININT	Nearest Integer [a + .5 * sign(a)]	1	ININT	REAL*4	INTEGER*2
		1	IIDNNT	REAL*8	INTEGER*2
INT	Truncation	1		NUMERIC	INTEGER*4
		1	JIFIX	REAL*4	INTEGER*4
		1	IDINT	REAL*8	INTEGER*4
INT8	Truncation	1		REAL*4	INTEGER*8
		1	KIFIX	REAL*4	INTEGER*8
IZEXT	Zero-Extend Function (Conversion)	1		LOGICAL*1	INTEGER*2
		1		LOGICAL*2	INTEGER*2
		1		INTEGER*2	INTEGER*2
JINT	Truncation	1		NUMERIC	INTEGER*4
		1	JINT	REAL*4	INTEGER*4
		1	JIDINT	REAL*8	INTEGER*4
JNINT	Nearest Integer [a + .5 * sign(a)]	1		REAL	INTEGER*4
		1	JIDNNT	REAL*8	INTEGER*4
KNINT	Nearest Integer [a + .5 * sign(a)]	1		REAL	INTEGER*8
		1	KIDNNT	REAL*8	INTEGER*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
MAX	Maximum	n > 1	IMAX0	INTEGER*2	INTEGER*2
		n > 1	IMAX1	REAL*4	INTEGER*2
		n > 1	AIMAX0	INTEGER*2	REAL*4
		n > 1	JMAX0	INTEGER*4	INTEGER*4
		n > 1		INTEGER*8	INTEGER*8
		n > 1	JMAX1	REAL*4	INTEGER*4
		n > 1	KMAX1	REAL*4	INTEGER*8
		n > 1	AJMAX0	INTEGER*4	REAL*4
		n > 1	AKMAX0	INTEGER*8	REAL*4
		n > 1	MAX0	INTEGER*4	INTEGER*4
		n > 1	AMAX1	REAL*4	REAL*4
		n > 1	DMAX1	REAL*8	REAL*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
MIN	Minimum	n > 1	IMIN0	INTEGER*2	INTEGER*2
		n > 1	IMIN1	REAL*4	INTEGER*2
		n > 1	AIMIN0	INTEGER*2	REAL*4
		n > 1	JMIN0	INTEGER*4	INTEGER*4
		n > 1		INTEGER*8	INTEGER*8
		n > 1	JMIN1	REAL*4	INTEGER*4
		n > 1	KMIN1	REAL*4	INTEGER*8
		n > 1	AJMIN0	INTEGER*4	REAL*4
		n > 1	MIN0	INTEGER*4	INTEGER*4
		n > 1	AMIN1	REAL*4	REAL*4
		n > 1	AKNIN0	INTEGER*8	REAL*4
		n > 1	DMIN1	REAL*8	REAL*8
MOD	Remainder	2	IMOD	INTEGER*2, INTEGER*2	INTEGER*2
		2	JMOD	INTEGER*4, INTEGER*4	INTEGER*4
		2	KMOD	INTEGER*8, INTEGER*8	INTEGER*8
		2	AMOD	REAL*4, REAL*4	REAL*4
		2	DMOD	REAL*8, REAL*4	REAL*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
MODULO	Fortran 90/95 Modulo	2		INTEGER*2, INTEGER*2	INTEGER*2
		2		INTEGER*4, INTEGER*4	INTEGER*4
		2		INTEGER*8, INTEGER*8	INTEGER*8
		2		REAL*4, REAL*4	REAL*4
		2		REAL*8, REAL*4	REAL*8
NINT	Nearest Integer [a + .5 * sign(a)]	1		REAL	INTEGER*4
		1	IDNINT	REAL*8	INTEGER*4
REAL	Convert to REAL*4	1		NUMERIC	REAL*4
		1	FLOATI	INTEGER*2	REAL*4
		1	FLOAT	INTEGER*2	REAL*4
		1	REAL	INTEGER*4	REAL*4
		1	FLOATJ	INTEGER*4	REAL*4
		1	FLOATK	INTEGER*8	REAL*4
		1	SNGL	REAL*8	REAL*4
SIGN	Transfer of Sign	2	IISIGN	INTEGER*2	INTEGER*2
		2	JISIGN	INTEGER*4	INTEGER*4
		2	KISIGN	INTEGER*8	INTEGER*8
		2	SIGN	REAL*4	REAL*4
		2	DSIGN	REAL*8	REAL*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
ZEXT	Zero-Extend Function (Conversion)	1	JZEXT	LOGICAL*1	INTEGER*4
		1		LOGICAL*2	INTEGER*4
		1		LOGICAL*4	INTEGER*4
		1		INTEGER*2	INTEGER*4
		1		INTEGER*4	INTEGER*4

Table 6-2: Mathematical Functions

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
ACOS	ArcCosine	1	ACOS	REAL*4	REAL*4
		1	DACOS	REAL*8	REAL*8
ACOSD	ArcCosine (degree)	1	ACOSD	REAL*4	REAL*4
		1	DACOSD	REAL*8	REAL*8
ASIN	ArcSine	1	ASIN	REAL*4	REAL*4
		1	DASIN	REAL*8	REAL*8
ASIND	ArcSine (degree)	1	ASIND	REAL*4	REAL*4
		1	DASIND	REAL*8	REAL*8
ATAN	ArcTangent	1	ATAN	REAL*4	REAL*4
		1	DATAN	REAL*8	REAL*8
ATAN2	ArcTangent	2	ATAN2	REAL*4, REAL*4	REAL*4
		2	DATAN2	REAL*8, REAL*8	REAL*8
ATAN2D	ArcTangent (degree)	2	ATAN2D	REAL*4, REAL*4	REAL*4
		2	DATAN2D	REAL*8, REAL*8	REAL*8
ATAND	ArcTangent (degree)	1	ATAND	REAL*4	REAL*4
		1	DATAND	REAL*8	REAL*8
COS	Cosine	1	COS	REAL*4	REAL*4
		1	DCOS	REAL*8	REAL*8
		1	CCOS	COMPLEX*8	COMPLEX*8
		1	CDCOS	COMPLEX*16	COMPLEX*16

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
COSD	Cosine (degree)	1	COSD	REAL*4	REAL*4
		1	DCOSD	REAL*8	REAL*8
COSH	Hyperbolic Cosine	1	COSH	REAL*4	REAL*4
		1	DCOSH	REAL*8	REAL*8
DPROD	Product	2		REAL*4, REAL*4	REAL*8
EXP	Exponential	1	EXP	REAL*4	REAL*4
		1	DEXP	REAL*8	REAL*8
		1	CEXP	COMPLEX*8	COMPLEX*8
		1	CDEXP	COMPLEX*16	COMPLEX*16
LOG	Natural Logarithm	1	ALOG	REAL*4	REAL*4
		1	DLOG	REAL*8	REAL*8
		1	CLOG	COMPLEX*8	COMPLEX*8
		1	CDLOG	COMPLEX*16	COMPLEX*16
LOG10	Common Logarithm	1	ALOG10	REAL*4	REAL*4
		1	DLOG10	REAL*8	REAL*8
SIN	Sine	1	SIN	REAL*4	REAL*4
		1	DSIN	REAL*8	REAL*8
		1	CSIN	COMPLEX*8	COMPLEX*8
		1	CDSIN	COMPLEX*16	COMPLEX*16
SIND	Sine (degree)	1	SIND	REAL*4	REAL*4
		1	DSIND	REAL*8	REAL*8

Generic Name	Purpose	Number of Args	Specific Name	Argument Type	Result Type
SINH	Hyperbolic Sine	1	SINH	REAL*4	REAL*4
		1	DSINH	REAL*8	REAL*8
SQRT	Square Root	1	SQRT	REAL*4	REAL*4
		1	DSQRT	REAL*8	REAL*8
		1	CSQRT	COMPLEX*8	COMPLEX*8
		1	REAL*8	COMPLEX*16	COMPLEX*16
TAN	Tangent	1	TAN	REAL*4	REAL*4
		1	DTAN	REAL*8	REAL*8
TAND	Tangent (degree)	1	TAND	REAL*4	REAL*4
		1	DTAND	REAL*8	REAL*8
TANH	Hyperbelic Tangent	1	TANH	REAL*4	REAL*4
		1	DTANH	REAL*8	REAL*8

Table 6-3: Real Manipulation Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
EXPONENT	Exponent part	1	REAL	INTEGER
FRACTION	Fractional part	1	REAL	INTEGER
NEAREST	Nearest different machine-representable number	2	REAL, REAL	REAL
RRSPACING	Reciprocal of relative spacing	1	REAL	REAL
SCALE	Value of exponent part changed by a specified value	2	REAL, INTEGER	REAL
SET_EXPONENT	Value of exponent part set to a specified value	2	REAL, INTEGER	REAL
SPACING	Spacing near argument	1	REAL	REAL

Table 6-4: Bit Manipulation Functions

Generic Name	Purpose	Num. Args	Specific Name	Argument Type	Result Type
AND	Logical AND	2		any[1], any*** 'any, any' on page 186 ***	typeless
BIT_SIZE	Precision (in bits)	1		INTEGER	INTEGER

Generic Name	Purpose	Num. Args	Specific Name	Argument Type	Result Type
BTEST	Bit Test	2		INTEGER, INTEGER	LOGICAL
		2	BITEST	INTEGER*2, INTEGER*2	LOGICAL*2
		2	BJTEST	INTEGER*4, INTEGER*4	LOGICAL*4
		2	KBTEST	INTEGER*8, INTEGER*8	LOGICAL*8
COMPL	Logical Comple-ment	1		any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***	typeless
EQV	Logical Exclusive Nor	2		any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***, any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***	typeless
IAND	Logical AND	2		INTEGER, INTEGER INTE-GER*2,	INTEGER
		2	IIAND	INTEGER*2	INTEGER*2
		2	JIAND	INTEGER*4, INTEGER*4	INTEGER*4
		2	KIAND	INTEGER*8, INTEGER*8	INTEGER*8
IBCLR	Bit Clear	2		INTEGER, INTEGER	INTEGER
		2	IIBCLR	INTEGER*2, INTEGER*2	INTEGER*2
		2	JIBCLR	INTEGER*4, INTEGER*4	INTEGER*4
		2	KIBCLR	INTEGER*8, INTEGER*8	INTEGER*8

Generic Name	Purpose	Num. Args	Specific Name	Argument Type	Result Type
IBITS	Bit Extraction	3		INTEGER, INTEGER, INTEGER	INTEGER
		3	IIBITS	INTEGER*2, INTEGER*2, INTEGER*2	INTEGER*2
		3	JIBITS	INTEGER*4, INTEGER*4, INTEGER*4	INTEGER*4
		3	KIBITS	INTEGER*8, INTEGER*8, INTEGER*8	INTEGER*8
IBSET	Bit Set	2		INTEGER, INTEGER	INTEGER
		2	IIBSET	INTEGER*2, INTEGER*2	INTEGER*2
		2	JIBSET	INTEGER*4, INTEGER*4	INTEGER*4
		2	KIBSET	INTEGER*8, INTEGER*8	INTEGER*8
IEOR	Logical XOR	2		INTEGER, INTEGER	INTEGER
		2	IIEOR	INTEGER*2, INTEGER*2	INTEGER*2
IOR	Logical OR	2		INTEGER, INTEGER	INTEGER
		2	IIOR	INTEGER*2, INTEGER*2	INTEGER*2
		2	JIOR	INTEGER*4, INTEGER*4	INTEGER*4
		2	KIOR	INTEGER*8, INTEGER*8	INTEGER*8
ISHFT	Logical Shift	2		INTEGER, INTEGER	INTEGER
		2	IISHFT	INTEGER*2, INTEGER*2	INTEGER*2
		2	JISHFT	INTEGER*4, INTEGER*4	INTEGER*4
		2	KISHFT	INTEGER*8, INTEGER*8	INTEGER*8

Generic Name	Purpose	Num. Args	Specific Name	Argument Type	Result Type
ISHFTC	Circular Shift	3		INTEGER, INTEGER	INTEGER
		3	IISHFTC	INTEGER*2, INTEGER*2, INTEGER*2	INTEGER*2
		3	JISHFTC	INTEGER*4, INTEGER*4, INTEGER*4	INTEGER*4
		3	KISHFTC	INTEGER*8, INTEGER*8, INTEGER*8	INTEGER*8
LSHIFT	Logical Left Shift	2		INTEGER, INTEGER	INTEGER
NEQV	Logical Exclusive OR	2		any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***, any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***	typeless
NOT	Logical Comple-ment	1		INTEGER	INTEGER
		1	INOT	INTEGER*2	INTEGER*2
		1	JNOT	INTEGER*4	INTEGER*4
		1	KNOT	INTEGER*8	INTEGER*8
OR	Logical OR	2		any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***, any*** 'any, any*** 'any, any' on page 186 ***' on page 186 ***	typeless
RSHIFT	Logical Right Shift	2		INTEGER, INTEGER	INTEGER

Generic Name	Purpose	Num. Args	Specific Name	Argument Type	Result Type
SHIFT	Logical Shift	2		any[2], INTEGER	typeless
XOR	Logical Exclusive OR	2	JIEOR	INTEGER, INTEGER INTEGER*4, INTEGER*4	INTEGER INTEGER*4

1. Arguments to the intrinsics AND, COMPL, EQV, OR, and NEQV may be of any type except for CHARACTER and COMPLEX.

2. The first argument to the SHIFT intrinsic may be of any type except for CHARACTER and COMPLEX.

Table 6-5: Fortran 90/95 Bit Manipulation Subroutine

Generic Name	Purpose	Arguments
MVBITS	Copies bit sequence	INTEGER(IN), INTEGER(IN), INTEGER(IN), INTEGER(INOUT), INTEGER(IN)

The functions in the following table are specific to Fortran 90/95 unless otherwise specified.

Table 6-6: Elemental Character and Logical Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ACHAR	Return character in specified ASCII collating position.	1	INTEGER	CHARACTER
ADJUSTL	Left adjust string	1	CHARACTER	CHARACTER
ADJUSTR	Right adjust string	1	CHARACTER	CHARACTER
CHAR (f77)	Return character with specified ASCII value.	1 1	LOGICAL*1 INTEGER	CHARACTER CHARACTER
IACHAR	Return position of character in ASCII collating sequence.	1	CHARACTER	INTEGER
ICHAR	Return position of character in the character set's collating sequence.	1	CHARACTER	INTEGER
INDEX	Return starting position of substring within first string.	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
LEN_TRIM	Return length of string minus trailing blanks.	1	CHARACTER	INTEGER

Generic Name	Purpose	Num. Args	Argument Type	Result Type
LGE	Lexical comparison	2	CHARACTER, CHARACTER	LOGICAL
LGT	Lexical comparison	2	CHARACTER, CHARACTER	LOGICAL
LLE	Lexical comparison	2	CHARACTER, CHARACTER	LOGICAL
LLT	Lexical comparison	2	CHARACTER, CHARACTER	LOGICAL
LOGICAL	Logical conversion	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
SCAN	Scan string for characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
VERIFY	Determine if string contains all characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER

Table 6-7: Fortran 90/95 Vector/Matrix Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
DOT_PRODUCT	Perform dot product on two vectors	2	NONCHAR*K, NONCHAR*K	NONCHAR*K
MATMUL	Perform matrix multiply on two matrices	2	NONCHAR*K, NONCHAR*K	NONCHAR*K

Table 6-8: Fortran 90/95 Array Reduction Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ALL	Determine if all array values are true	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
ANY	Determine if any array value is true	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
COUNT	Count true values in array	1 2	LOGICAL LOGICAL, INTEGER	INTEGER INTEGER

Generic Name	Purpose	Number of Args	Argument Type	Result Type
MAXLOC	Determine position of array element with maximum value	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	INTEGER
		2	REAL, LOGICAL	INTEGER
		2	REAL, INTEGER	INTEGER
		3	REAL, INTEGER, LOGICAL	INTEGER
MAXVAL	Determine maximum value of array elements	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
		3	REAL, INTEGER, LOGICAL	REAL

Generic Name	Purpose	Number of Args	Argument Type	Result Type
MINLOC	Determine position of array element with minimum value	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	INTEGER
		2	REAL, LOGICAL	INTEGER
		2	REAL, INTEGER	INTEGER
		3	REAL, INTEGER, LOGICAL	INTEGER
MINVAL	Determine minimum value of array elements	1	INTEGER	INTEGER
		2	INTEGER, LOGICAL	INTEGER
		2	INTEGER, INTEGER	INTEGER
		3	INTEGER, INTEGER, LOGICAL	INTEGER
		1	REAL	REAL
		2	REAL, LOGICAL	REAL
		2	REAL, INTEGER	REAL
		3	REAL, INTEGER, LOGICAL	REAL
PRODUCT	Calculate the product of the elements of an array	1	NUMERIC	NUMERIC
		2	NUMERIC, LOGICAL	NUMERIC
		2	NUMERIC, INTEGER	NUMERIC
		3	NUMERIC, INTEGER, LOGICAL	NUMERIC

Generic Name	Purpose	Number of Args	Argument Type	Result Type
SUM	Calculate the sum of the elements of an array	1	NUMERIC	NUMERIC
		2	NUMERIC, LOGICAL	NUMERIC
		2	NUMERIC, INTEGER	NUMERIC
		3	NUMERIC, INTEGER, LOGICAL	NUMERIC

Table 6-9: Fortran 90/95 String Construction Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
REPEAT	Concatenate copies of a string	2	CHARACTER, INTEGER	CHARACTER
TRIM	Remove trailing blanks from a string	1	CHARACTER	CHARACTER

Table 6-10: Fortran 90/95 Array Construction/Manipulation Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
CSHIFT	Perform circular shift on array	2	ARRAY, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, INTEGER, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***

Generic Name	Purpose	Number of Args	Argument Type	Result Type
EOSHIFT	Perform end-off shift on array	2	ARRAY, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, INTEGER, any*** 'any, any, LOGICAL' on page 196 ***	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, INTEGER, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		4	ARRAY, INTEGER, any*** 'any, any, LOGICAL' on page 196 ***, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
MERGE	Merge two arguments based on logical mask	3	any, any[1], LOGICAL	any*** 'any, any, LOGICAL' on page 196 ***

Generic Name	Purpose	Number of Args	Argument Type	Result Type
PACK	Pack array into rank-one array	2	ARRAY, LOGICAL	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, LOGICAL, VECTOR*** 'any, any, LOGICAL' on page 196 ***	ARRAY*** 'any, any, LOGICAL' on page 196 ***
RESHAPE	Change the shape of an array	2	ARRAY, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, INTEGER, ARRAY*** 'any, any, LOGICAL' on page 196 ***	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		3	ARRAY, INTEGER, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
		4	ARRAY, INTEGER, ARRAY*** 'any, any, LOGICAL' on page 196 ***, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***

Generic Name	Purpose	Number of Args	Argument Type	Result Type
SPREAD	Replicates an array by adding a dimension	3	any, INTEGER, INTEGER	ARRAY*** 'any, any, LOGICAL' on page 196 ***
TRANSPOSE	Transpose an array of rank two	1	ARRAY	ARRAY*** 'any, any, LOGICAL' on page 196 ***
UNPACK	Unpack a rank-one array into an array of multiple dimensions	3	VECTOR, LOGICAL, ARRAY*** 'any, any, LOGICAL' on page 196 ***	ARRAY*** 'any, any, LOGICAL' on page 196 ***

1. Must be of the same type as the first argument.

Table 6-11: Fortran 90/95 General Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ASSOCIATED	Determine association status	12	POINTERPOINTER, TARGET	LOGICAL-LOGICAL
KIND	Determine argument's kind	1	any intrinsic type	INTEGER
PRESENT	Determine presence of optional argument	1	any	LOGICAL

Table 6-12: Fortran 90/95 Numeric Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
DIGITS	Determine number of significant digits	1 1	INTEGER REAL	INTEGER INTEGER
EPSILON	Smallest representable number	1	REAL	REAL
HUGE	Largest representable number	1 1	INTEGER REAL	INTEGER REAL
MAXEXPONENT	Value of maximum exponent	1	REAL	INTEGER
MINEXPONENT	Value of minimum exponent	1	REAL	INTEGER
PRECISION	Decimal precision	1 1	REAL COMPLEX	INTEGER INTEGER

Generic Name	Purpose	Number of Args	Argument Type	Result Type
RADIX	Base of model	1	INTEGER	INTEGER
		1	REAL	INTEGER
RANGE	Decimal exponent range	1	INTEGER	INTEGER
		1	REAL	INTEGER
		1	COMPLEX	INTEGER
SELECTED_INT_KIND	Kind type titlemeter in range	1	INTEGER	INTEGER
SELECTED_REAL_KIND	Kind type titlemeter in range	1	INTEGER	INTEGER
		2	INTEGER, INTEGER	INTEGER
TINY	Smallest representable positive number	1	REAL	REAL

Table 6-13: Fortran 90/95 Array Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ALLOCATED	Determine if array is allocated	1	ARRAY	LOGICAL
LBOUND	Determine lower bounds	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	INTEGER
SHAPE	Determine shape	1	any	INTEGER
SIZE	Determine number of elements	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	INTEGER
UBOUND	Determine upper bounds	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	INTEGER

Table 6-14: Fortran 90/95 String Inquiry Function

Generic Name	Purpose	Number of Args	Argument Type	Result Type
LEN	Length of string	1	CHARACTER	INTEGER

Table 6-15: Fortran 90/95 Subroutines

Generic Name	Purpose	Number of Args	Argument Type
CPU_TIME	Returns processor time	1	REAL (OUT)

Generic Name	Purpose	Number of Args	Argument Type
DATE_AND_TIME	Returns date and time	4 (optional)	DATE (CHARACTER, OUT) TIME (CHARACTER, OUT) ZONE (CHARACTER, OUT) VALUES (INTEGER, OUT)
RANDOM_NUMBER	Generate pseudo-random numbers	1	REAL (OUT)
RANDOM_SEED	Set or query pseudo-random number generator	0 1 1 1	 SIZE (INTEGER, OUT) PUT (INTEGER ARRAY, IN) GET (INTEGER ARRAY, OUT)
SYSTEM_CLOCK	Query real time clock	3 (optional)	COUNT (INTEGER, OUT) COUNT_RATE (INTEGER, OUT) COUNT_MAX (INTEGER, OUT)

Table 6-16: Fortran 90/95 Transfer Function

Generic Name	Purpose	Number of Args	Argument Type	Result Type
TRANSFER	Change type but maintain bit representation	23 3	any, any any, any, INTEGER	any[1]

1. Must be of the same type as the second argument.

Table 6-17: Miscellaneous Functions

Generic Name	Purpose	Lang	Number of Args	Argument Type	Result Type
LOC	Return address of argument	F77	1	NUMERIC	INTEGER
NULL	Assign disassociated status	F95	0 1	 POINTER	POINTER POINTER

FORTRAN 77 and Fortran 90/95 Intrinsics Descriptions

This section contains descriptions of each FORTRAN 77 and Fortran 90/95 intrinsic supported by the PGF77 and PGF95 compilers. Intrinsics and subroutines are listed alphabetically.

This section contains descriptions of each FORTRAN 77 and Fortran 90/95 intrinsic supported by the PGF77, PGF95 and PGHPF compilers. Intrinsics and subroutines are listed alphabetically.

ABS 77

Determine the absolute value of the supplied argument.

Synopsis

```
ABS(A)
```

Argument

The argument A must be of type integer, real, or complex.

Return Value

The return type for integer is integer, for real is real, and for complex is real.

ACHAR 90

Return the character in the ASCII collating position specified by the argument.

Synopsis

```
ACHAR(I)
```

Argument

The argument I must be of type integer.

Return Value

A single character.

ACOS 77

Return the arccosine of the specified value.

Synopsis

```
ACOS(X)
```

Arguments

The argument X must be a real value.

Return Value

The real value representing the arccosine in radians.

ACOSD 77

Return the arccosine (in degrees) of the specified value.

Synopsis

```
ACOSD(X)
```

Arguments

The argument X must be a real value.

Return Value

The real value representing the arccosine in degrees.

ADJUSTL 90

Adjust string to the left, removing all leading blanks and inserting trailing blanks.

Synopsis

```
ADJUSTL(STR)
```

Arguments

The argument STR is the string to be adjusted.

Return Value

String of same length and kind as the argument with leading blanks removed and the same number of trailing blanks added.

ADJUSTR 90

Adjust string to the right, removing all trailing blanks and inserting leading blanks.

Synopsis

```
ADJUSTR(STR)
```

Arguments

The argument STR is the string to be adjusted.

Return Value

String of same length and kind as the argument with trailing blanks removed and the same number of leading blanks added.

AIMAG 77

Determine the value of the imaginary part of a complex number.

Synopsis

```
AIMAG(Z)
```

Arguments

The argument Z must be complex.

Return Value

A real value representing the imaginary part of the supplied argument.

AINT 77

Truncate the supplied value to a whole number.

Synopsis

```
AINT(A [,KIND])
```

Arguments

The argument A is of type real. The optional KIND argument is an integer kind.

Return Value

A real value that is equal to the largest integer that is not greater than the supplied argument. If the KIND argument is present, the result is of that kind.

ALL 90

Determine if all the values in the supplied argument are logical true.

Synopsis

```
ALL(MASK [,DIM])
```

Arguments

The argument MASK is an array of type LOGICAL. The optional argument DIM specifies the dimension of the array MASK to check.

Return Value

If no DIM argument is present, the return value is a logical scalar that is true only if all values of MASK are true.

If the DIM argument is present and if MASK has rank one, then the return value is the same as ALL(MASK).

If the DIM argument is present and MASK has rank greater than one, then the return value is an array that has rank n-1, where n is the rank of MASK. The return value is defined recursively as the value of ALL for each extent of the dimension DIM (refer to the Fortran 95 Handbook for a more detailed explanation).

ALLOCATED 90

Determine if the supplied allocatable array is currently allocated.

Synopsis

```
ALLOCATED(ARRAY)
```

Argument

The argument ARRAY is an allocatable array.

Return Value

Returns a logical scalar indicating whether the array is allocated.

AND § 77

Performs a logical AND on corresponding bits of the arguments.

Synopsis

```
AND(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

ANINT 77

Return the nearest whole number to the supplied argument.

Synopsis

```
ANINT(A [,KIND])
```

Arguments

The argument A is a real number. The optional argument KIND is a kind parameter.

Return Value

The result is a real. The value is AINT(A+0.5) if A is > 0 and AINT(A-0.5) if A is < 0. If KIND is present, the result is of type KIND.

ANY 90

Determine if any value in the supplied argument MASK is true.

Synopsis

```
ANY(MASK [,DIM])
```

Arguments

The argument MASK is an array of type LOGICAL. The optional argument DIM specifies the dimension of the array MASK to check.

Return Value

If no DIM argument is present, the return value is a logical scalar that is true if any element of MASK is true.

If the DIM argument is present and if MASK has rank one, then the return value is the same as ANY(MASK).

If the DIM argument is present and MASK has rank greater than one, then the return value is an array that has rank n-1, where n is the rank of MASK. The return value is defined recursively as the value of ANY for each extent of the dimension DIM (refer to The Fortran 95 Handbook for a more detailed explanation)

ASIN 77

Return the arcsine of the specified value.

Synopsis

```
ASIN(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

The real value representing the arcsine in radians.

ASIND 77

Return the arcsine (in degrees) of the specified value.

Synopsis

```
ASIND(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

The real value representing the arcsine in degrees.

ASSOCIATED 90

Determines the association status of the supplied argument or determines if the supplied pointer is associated with the supplied target.

Synopsis

```
ASSOCIATED(POINTER [,TARGET])
```

Arguments

The POINTER argument is a pointer of any type. The optional argument TARGET is a pointer or a target. If it is a pointer it must not be undefined.

Return Value

If TARGET is not supplied the function returns logical true if POINTER is associated with a target and false otherwise.

If TARGET is present and is a target, then the function returns true if POINTER is associated with TARGET and false otherwise.

If TARGET is present and is a pointer, then the function returns true if POINTER and TARGET are associated with the same target and false otherwise.

ATAN 77

Return the arctangent of the specified value.

Synopsis

```
ATAN(X)
```

Argument

The argument X must be of type real.

Return Value

The real value representing the arctangent in radians.

ATAN2 77

Return the arctangent of the specified value.

Synopsis

```
ATAN2(Y, X)
```

Arguments

The arguments X and Y must be of type real.

Return Value

A real number that is the arctangent for pairs of reals, X and Y, expressed in radians. The result is the principal value of the nonzero complex number (X,Y).

ATAN2D 77

Return the arctangent (in degrees) of the specified value.

Synopsis

```
ATAN2D(Y, X)
```

Arguments

The arguments X and Y must be of type real.

Return Value

A real number that is the arctangent for pairs of reals, X and Y, expressed in degrees. The result is the principal value of the nonzero complex number (X,Y).

ATAND 77

Return the arctangent (in degrees) of the specified value.

Synopsis

```
ATAND(X)
```

Argument

The argument X must be of type real.

Return Value

The real value representing the arctangent in degrees.

BIT_SIZE 90

Return the number of bits (the precision) of the integer argument. This function uses the standard Fortran 90/95 bit model defined in The Fortran 95 Handbook.

Synopsis

```
BIT_SIZE(I)
```

Argument

The argument I must be of type integer.

Return Value

Returns an integer.

BTEST 77

Tests the binary value of a bit in a specified position of an integer argument. This function uses the standard Fortran 90/95 bit model defined in The Fortran 95 Handbook.

Synopsis

```
BTEST(I, POS)
```

Arguments

The argument I must be of type integer. The argument POS must be an integer with a value less than or equal to the value BIT_SIZE(I).

Return Value

Returns a logical value representing whether the bit in position POS is true or false (0 or 1).

CEILING 90

Return the least integer greater than or equal to the supplied real argument.

Synopsis

```
CEILING(A [,KIND])
```

Argument

The argument A is a real value. The optional argument KIND is a kind parameter and was added to CEILING in Fortran 95.

Return Value

The return value is an integer. If KIND is present, the result is of type KIND.

CHAR 77

Returns the character in the specified collating sequence.

Synopsis

```
CHAR(I [,KIND])
```

Arguments

The argument I is of type integer, specifying the character position to return. The argument KIND is optional.

Return Value

A character.

CMPLX 77

Convert the supplied argument or arguments to complex type.

Synopsis

```
CMPLX(X [,Y][,KIND])
```

Arguments

The argument X is of type integer, real, or complex. The optional argument Y is of type integer or real. If X is complex, Y must not be present. The optional argument KIND is the kind for the return value.

Return Value

Returns a complex number with the value specified by the arguments converted to a real part and a complex part. If the KIND parameter is not supplied, the KIND is the same as the KIND for the default complex.

COMPL § 77

Performs a logical complement on the argument.

Synopsis

```
COMPL(M)
```

Arguments

The argument M may be of any type except for character and complex.

Return Value

The return value is typeless.

CONJG 77

Return the conjugate of the supplied complex number.

Synopsis

```
CONJG(Z)
```

Argument

The argument Z is a complex number.

Return Value

The return value is the same type and kind as Z.

COS 77

Return the cosine of the specified value.

Synopsis

```
COS(X)
```

Argument

The argument X must be of type real or complex.

Return Value

A real value of the same kind as the argument. The return value for a real argument is in radians, or if complex, the real part is a value in radians.

COSD 77

Return the cosine (in degrees) of the specified value.

Synopsis

```
COSD(X)
```

Argument

The argument X must be of type real or complex.

Return Value

A real value of the same kind as the argument. The return value for a real argument is in degrees, or if complex, the real part is a value in degrees.

COSH 77

Return the hyperbolic cosine of the specified value.

Synopsis

```
COSH(X)
```

Argument

The argument X must be of type real.

Return Value

A real value.

COUNT 90

Return the number of true elements in the supplied logical argument (array), along the specified dimension if the optional argument is present.

Synopsis

```
COUNT(MASK [,DIM])
```

Arguments

The argument MASK is an array of type LOGICAL. The optional argument DIM specifies the dimension of the array MASK to count.

Return Value

If no DIM argument is present, the return value is an integer that is the count of true values in MASK.

If the DIM argument is present and if MASK has rank one, then the return value is the same as COUNT(MASK).

If the DIM argument is present and MASK has rank greater than one, then the return value is an array that has rank n-1, where n is the rank of MASK. The return value is defined recursively as the value of COUNT for each extent of the dimension DIM (refer to The Fortran 95 Handbook for a more detailed explanation).

CPU_TIME 95

This is a non-elemental intrinsic subroutine that returns the processor time. For a more detailed explanation, refer to Fortran 95 Explained.

Synopsis

```
call cpu_time (TIME)
```

Arguments

The argument TIME is a scalar real that is assigned a processor-dependent approximation of processor time.

Return Value

The returned value in seconds, or a processor-dependent value if there is no clock.

CSHIFT 90

Perform a circular shift on the specified array.

Synopsis

```
CSHIFT(ARRAY, SHIFT [,DIM])
```

Arguments

The argument ARRAY is the array to shift. It may be an array of any type. The argument SHIFT is an integer or an array of integers with rank n-1 where n is the rank of ARRAY. The optional argument DIM is an integer representing the dimension to shift.

Return Value

The shifted array with the same size and shape as the argument ARRAY.

DATE_AND_TIME 90

This is a subroutine that returns the date and time.

Synopsis

```
DATE_AND_TIME([DATE] [,TIME] [,ZONE] [,VALUES])
```

Arguments

All of the arguments are optional. The DATE argument is of type default character. It must be at least 8 characters long. The argument returns the value CCYYMMDD where CC is the century, YY is the year, MM is the month, and DD is the day.

The argument TIME is of type default character. It must be at least 10 characters long. It has the form hhmmss.sss, where hh is the hour, mm is the minute, and ss.sss is the seconds and milliseconds.

The argument ZONE is of type default character. It must be at least 5 characters long. It has the form +– hhmm where hh and mm are the hours and minutes that the local time zone differs from universal time (UTC).

The argument VALUES must be an array of type default integer. It has the following eight values:

```
VALUES(1)holds the year
VALUES(2)holds the month
VALUES(3)holds the day of the month
VALUES(4)holds the time difference with respect to UTC
VALUES(5)holds the hour of the day
VALUES(6)holds the minutes of the hour
VALUES(7)holds the seconds of the minute
VALUES(8)holds the milliseconds of the second,
 in the range 0 to 999
```

Return Value

As this is a subroutine, the values are returned in the arguments.

DBLE 77

Convert to double precision real.

Synopsis

```
DBLE(A)
```

Argument

The argument A must be of type integer, real, or complex.

Return Value

If A is of type integer or real, the return value is the value converted to a double precision real. If A is of type complex, the return value is the double precision value of the real part of the complex argument.

DCMPLX 77

Convert the supplied argument or arguments to double complex type.

Synopsis

```
DCMPLX(X [,Y])
```

Arguments

The argument X is of type integer, real, or complex. The optional argument Y is of type integer or real. If X is complex, Y must not be present.

Return Value

Returns a double complex number with the value specified by the arguments converted to a real part and a complex part.

DIGITS 90

Returns the number of significant digits in the model representing the argument.

Synopsis

```
DIGITS(X)
```

Argument

The argument X is of type integer or real.

Return Value

An integer value representing the number of digits in the model representing the specified kind.

DIM 77

This intrinsic returns the difference X-Y if the value is positive, otherwise it returns 0.

Synopsis

```
DIM(X, Y)
```

Arguments

X must be of type integer or real. Y must be of the same type and kind as X.

Return Value

The result is the same type and kind as X with the value X-Y if X>Y, otherwise zero.

DOT_PRODUCT 90

Perform a dot product on two vectors (arrays).

Synopsis

```
DOT_PRODUCT(VECTOR_A, VECTOR_B)
```

Arguments

VECTOR_A must be an array of rank one of type numeric (integer, real, complex) or logical. VECTOR_B must be numeric if VECTOR_A is numeric, or logical if VECTOR_A is logical. It must have the same rank and size as ARRAY_A.

Return Value

The dot product. For VECTOR_A of integer or real, the value is SUM(VECTOR_A * VECTOR_B). For complex, the value is SUM(CONJG(VECTOR_A) * VECTOR_B). For logical, the value is ANY(VECTOR_A .AND. VECTOR_B).

DPROD 90

Double precision real product.

Synopsis

```
DPROD(X,Y)
```

Arguments

Both arguments X and Y must be of type default real.

Return Value

The return value is a double precision real that is the product of X and Y.

EOSHIFT 90

Perform an end-off shift on the specified array.

Synopsis

```
EOSHIFT(ARRAY, SHIFT [,BOUNDARY] [,DIM])
```

Arguments

The argument ARRAY is the array to shift. It may be an array of any type. The argument SHIFT is an integer or an array of integers with rank n-1 where n is the rank of ARRAY. The optional argument BOUNDARY is of the same type as the array, it may be scalar or of rank n-1 where n is the rank of

ARRAY. The optional argument BOUNDARY is the value to fill in the shifted out positions. By default it has the following values for integer 0, for real 0.0, for complex (0.0,0.0), for logical false, for character the default is blank characters.

The optional argument DIM represents the dimension of ARRAY to shift.

Return Value

The shifted array with the same size and shape as the argument ARRAY.

EPSILON 90

Return the smallest number representable in the kind of the supplied argument.

Synopsis

```
EPSILON(X)
```

Argument

The argument X must be of type real.

Return Value

A very small number in the specified real kind.

EQV § 77

Performs a logical exclusive NOR on the arguments.

Synopsis

```
COMPL(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

EXP 77

Exponential function.

Synopsis

```
EXP(X)
```

Argument

The argument X must be of type real or complex.

Return Value

The value returned is of the same type as the argument. It has the value ex .

EXPONENT 90

Return the exponent part of a real number.

Synopsis

```
EXPONENT(X)
```

Argument

The argument X is a real number.

Return Value

The return value is an integer which has the value of the exponent part of the value of X. If the exponent is zero, the function returns zero. If the exponent is too large to be defined as an integer, the result is undefined.

FLOOR 90

Return the greatest integer less than or equal to the supplied real argument.

Synopsis

```
FLOOR(A [,KIND])
```

Argument

The argument A is a real value. The optional argument KIND is a kind parameter and was added to FLOOR in Fortran 95.

Return Value

The return value is an integer. If KIND is present, the result is of type KIND.

FRACTION 90

Return the fractional part of a real number.

Synopsis

```
FRACTION(X)
```

Argument

The argument X is a real number.

Return Value

The return value is an integer which has the value of the fractional part of the value of X. If the fraction value is zero, the function returns zero.

HUGE 90

Return the largest number representable in the kind of the supplied argument.

Synopsis

```
HUGE(X)
```

Argument

The argument X must be of type integer or real.

Return Value

A value of the same type as the argument with the maximum value possible.

IACHAR 90

Returns the position of the character in the ASCII collating sequence.

Synopsis

```
IACHAR(C)
```

Argument

The argument C must be of type character.

Return Value

An integer representing the character position.

IAND 77

Perform a bit-by-bit logical AND on the arguments.

Synopsis

```
IAND(I, J)
```

Arguments

The arguments I and J must be of type integer of the same kind.

Return Value

The return value is an integer value representing a bit-by-bit logical AND of the bits in the two integer arguments.

IBCLR 77

Clears one bit to zero.

Synopsis

```
IBCLR(I, POS)
```

Arguments

I is an integer. POS is a nonnegative integer less than BIT_SIZE(I).

Return Value

The return value is of the same type as I with a value that is the same as I except the bit in position POS is set to 0.

IBITS 77

Extracts a sequence of bits.

Synopsis

```
IBITS(I, POS, LEN)
```

Arguments

I is an integer. POS is a nonnegative integer and POS + LEN must be less than or equal to BIT_SIZE(I). LEN is of type integer and is nonnegative.

Return Value

The return value is of the same type as I with a value that is the sequence of LEN bits in I beginning at position POS, right-adjusted and with all other bits set to zero.

IBSET 77

Set one bit to one.

Synopsis

```
IBSET(I, POS)
```

Arguments

I is an integer. POS is a nonnegative integer less than BIT_SIZE(I).

Return Value

The return value is of the same type as I with a value that is the same as I except the bit in position POS is set to 1.

ICHAR 90

Returns the position of a character in the character set's collating sequence.

Synopsis

```
ICHAR(C)
```

Argument

The argument C must be of type character and length one.

Return Value

An integer representing the character position.

IEOR 77

Perform a bit-by-bit logical exclusive OR on the arguments.

Synopsis

```
IEOR(I, J)
```

Argument

The arguments I and J must be of type integer of the same kind.

Return Value

The return value is an integer value representing a bit-by-bit logical exclusive OR of the bits in the two integer arguments.

IINT § 77

Converts a value to a short integer type.

Synopsis

```
IINT(A)
```

Arguments

The argument A is of type integer, real, or complex.

Return Value

The return value is the short integer value of the supplied argument. For a real number, if the absolute value of the real is less than 1, the return value is 0. If the absolute value is greater than 1, the result is the largest short integer that does not exceed the real value. If argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

INDEX 90

Returns the starting position of a substring within a string.

Synopsis

```
INDEX(STRING, SUBSTRING [,BACK])
```

Arguments

The argument STRING must be of type character string. The argument SUBSTRING must be of type character string with the same kind as STRING. The optional argument BACK must be of type logical.

Return Value

The function returns an integer. If BACK is absent or false, the result is the starting point of the first matching SUBSTRING within STRING. Zero is returned if no match is found. 1 is returned if the SUBSTRING has zero length.

If BACK is present with the value true, the result is the last matching substring in string, or zero if no match is found.

ININT § 77

Returns the nearest short integer to the real argument.

Synopsis

```
ININT(A)
```

Arguments

The argument A must be a real.

Return Value

The result is a short integer with value (A + .5 * SIGN(A)).

INT 77

Converts a value to integer type.

Synopsis

```
INT(A [,KIND])
```

Arguments

The argument A is of type integer, real, or complex. The optional argument KIND must be a scalar integer that is a valid kind for the specified type. The KIND argument is not allowed by pgf77.

Return Value

The return value is the integer value of the supplied argument. For a real number, if the absolute value of the real is less than 1, the return value is 0. If the absolute value is greater than 1, the result is the largest integer that does not exceed the real value. If the argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

INT8 § 77

Converts a real value to a long integer type.

Synopsis

```
INT8(A)
```

Arguments

The argument A is of type real.

Return Value

The return value is the long integer value of the supplied argument.

IOR 77

Perform a bit-by-bit logical OR on the arguments.

Synopsis

```
IOR(I, J)
```

Argument

The arguments I and J must be of type integer of the same kind.

Return Value

The return value is an integer value representing a bit-by-bit logical OR of the bits in the two integer arguments.

ISHFT 77

Perform a logical shift.

Synopsis

```
ISHFT(I, SHIFT)
```

Arguments

I and SHIFT are integer values. The absolute value of SHIFT must be less than or equal to BIT_SIZE(I).

Return Value

The return value is of the same type and kind as the argument I. It is the value of the argument I logically shifted by SHIFT bits. If SHIFT is positive, the shift is to the left. If SHIFT is negative, the shift is to the right. Zeroes are shifted in at the ends and the bits shifted out are lost.

ISHFTC 77

Perform a circular shift of the rightmost bits.

Synopsis

```
ISHIFTC(I, SHIFT [,SIZE])
```

Arguments

I and SHIFT are integer values. The absolute value of SHIFT must be less than or equal to the optional argument SIZE. If present, SIZE must not exceed the value BIT_SIZE(I); if SIZE is not present, the function acts as if it were present with the value BIT_SIZE(I).

Return Value

The result is the value of the sub-group of SIZE bits shifted by SHIFT positions within the sub-group, all other bits remain the same and remain in the same position. If the SIZE argument is not present, the circular shift is over the complete group of bits and all of the bits are shifted by SHIFT positions. If SHIFT is positive, the shift is to the left. If SHIFT is negative, the shift is to the right.

IZEXT § 77

Zero-extend the argument.

Synopsis

```
IZEXT(A)
```

Arguments

The argument A is of type logical or integer.

Return Value

The return value is a zero-extended short integer of the argument.

JINT § 77

Converts a value to an integer type.

Synopsis

```
JINT(A)
```

Arguments

The argument A is of type integer, real, or complex.

Return Value

The return value is the integer value of the supplied argument. For a real number, if the absolute value of the real is less than 1, the return value is 0. If the absolute value is greater than 1, the result is the largest integer that does not exceed the real value. If argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

JNINT § 77

Returns the nearest integer to the real argument.

Synopsis

```
JNINT(A)
```

Arguments

The argument A must be a real.

Return Value

The result is an integer with value (A + .5 * SIGN(A)).

KIND 90

Returns the kind of the supplied argument.

Synopsis

```
KIND(X)
```

Argument

The argument X is of any intrinsic type.

Return Value

The result is an integer representing the kind type parameter of X.

KNINT § 77

Returns the nearest integer to the real argument.

Synopsis

```
KNINT(A)
```

Arguments

The argument A must be a real.

Return Value

The result is a long integer with value (A + .5 * SIGN(A)).

LBOUND 90

Returns the lower bounds of an array, or the lower bound for the specified dimension.

Synopsis

```
LBOUND(ARRAY [,DIM])
```

Arguments

The argument ARRAY is an array of any type. The optional argument DIM is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and n where n is the rank of the array).

Return Value

The return value is an integer, or an array of rank one and size n, where n is the rank of the argument ARRAY. For the function with a DIM argument, the return value is the value of the lower bound in the specified dimension. For the function with no DIM supplied, the return value is an array with all the lower bounds of ARRAY.

LEN 90

Returns the length of the supplied string.

Synopsis

```
LEN(STRING)
```

Argument

The argument STRING is a character string or an array.

Return Value

The return value is an integer that represents the length of the scalar string supplied, or the length of an element of STRING if STRING is an array.

LEN_TRIM 90

Returns the length of the supplied string minus the number of trailing blanks.

Synopsis

```
LEN_TRIM(STRING)
```

Arguments

The argument STRING is a character string.

Return Value

The return value is an integer that represents the length of the scalar string minus the number of trailing blanks, if any.

LGE 90

Test the supplied strings to see if the first string STRING_A is lexically greater than or equal to the second string STRING_B. A string is lexically greater than another string if the first string follows the second string alphabetically.

Synopsis

```
LGE(STRING_A, STRING_B)
```

Argument

The arguments STRING_A and STRING_B are of type default character.

Return Value

The function returns a logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

LGT 90

Test the supplied strings to see if the first string STRING_A is lexically greater than the second string STRING_B. A string is lexically greater than another string if the first string follows the second string alphabetically.

Synopsis

```
LGT(STRING_A, STRING_B)
```

Argument

The arguments STRING_A and STRING_B are of type default character.

Return Value

The function returns a logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

LLE 90

Test the supplied strings to see if the first string STRING_A is lexically less than or equal to the second string STRING_B. A string is lexically less than another string if the first string precedes the second string alphabetically.

Synopsis

```
LLE(STRING_A, STRING_B)
```

Argument

The arguments STRING_A and STRING_B are of type default character.

Return Value

The function returns a logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

LLT 90

Test the supplied strings to see if the first string STRING_A is lexically less than the second string, STRING_B. A string is lexically less than another string if the first string precedes the second string alphabetically.

Synopsis

```
LLT(STRING_A, STRING_B)
```

Argument

The arguments STRING_A and STRING_B are of type default character.

Return Value

The function returns a logical value. If the strings are not of the same length, the shorter string is padded with blanks on the right.

LOC 77

Return the 32-bit address of a data item.

Synopsis

```
LOC(X)
```

Argument

The argument X is of type integer, real or complex.

Return Value

The return value is an integer representing the address of the argument.

LOG 77

Function returns the natural logarithm.

Synopsis

```
LOG(X)
```

Argument

The argument X is of type real or complex. If X is real, it must be greater than 0. If X is complex, it must not be equal to zero.

Return Value

The return value is the natural log (base e) of X.

LOG10 77

Function returns the common logarithm.

Synopsis

```
LOG10(X)
```

Argument

The argument X is of type real and must be greater than 0.

Return Value

The return value is the common log (base 10) of X.

LOGICAL 90

Convert a logical value to the specified logical kind.

Synopsis

```
LOGICAL(L [,KIND])
```

Arguments

The argument L is the logical value to convert. The optional argument KIND must be a scalar integer that is a valid kind for the specified type.

Return Value

Returns a logical value equal to the logical value L. If KIND is specified, the kind type parameter of the return value is that of KIND, otherwise it is default logical.

LSHIFT § 77

Perform a logical shift to the left.

Synopsis

```
LSHIFT(I, SHIFT)
```

Arguments

I and SHIFT are integer values.

Return Value

The return value is of the same type and kind as the argument I. It is the value of the argument I logically shifted left by SHIFT bits.

MATMUL 90

Perform a matrix multiply of numeric or logical matrices.

Synopsis

```
MATMUL(MATRIX_A, MATRIX_B)
```

Arguments

The argument MATRIX_A must be numeric (integer, real, or complex) or logical, and have a rank of one or two. The argument MATRIX_B must be numeric (integer, real, or complex) or logical, and have a rank of one or two. If MATRIX_A has rank one, then MATRIX_B must have rank 2. If MATRIX_B has rank one, then MATRIX_A must have rank 2. The size of the first dimension of MATRIX_B must equal the size of the last dimension of MATRIX_A.

Return Value

A matrix representing the value of the matrix multiplied arguments. There are three possible result shapes: MATRIX_A(n,m) and MATRIX_B(m,k) gives a result (n,k) matrix. MATRIX_A(m) and MATRIX_B(m,k) gives a result (k), MATRIX_A(n,m) and MATRIX_B(m) gives a result(n).

MAX 77

Return the maximum value of the supplied arguments.

Synopsis

```
MAX( A1, A2 [,A3,...])
```

Arguments

Any arguments after the first two are optional. The arguments must all have the same kind and they must be integer or real.

Return Value

The return value is the same as the type and kind of the arguments. It has the value of the largest argument.

MAXEXPONENT 90

Returns the value of the maximum exponent for the type and the kind supplied.

Synopsis

```
MAXEXPONENT(X)
```

Argument

The argument X must be a scalar or an array of type real.

Return Value

The return value is an integer. It contains the value of the largest exponent in the specified kind.

MAXLOC 90

Determine the first position in the specified array that has the maximum value of the values in the array. The test elements may be limited by a dimension argument or by a logical mask argument.

Synopsis

```
MAXLOC(ARRAY [,DIM] [,MASK])
```

Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM, added in Fortran 95, is of type integer. The optional argument MASK must be of type logical and must have the same shape as ARRAY. If only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is an integer array of rank 1with size equal to the number of dimensions in ARRAY. The return values represent the positions of the first element in each dimension that is the maximum value of that dimension.

If the MASK parameter is present, the return value is the position of the first value that has the maximum value of values in ARRAY, and that also has a true value in the corresponding MASK array.

When the DIM argument is supplied, the return value is an array that has a value of MAXLOC applied recursively along the DIM dimensions of the array.

MAXVAL 90

Return the maximum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

Synopsis

```
MAXVAL(ARRAY [,DIM] [,MASK])
```

Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and n where n is the rank of the array). The optional argument MASK must be of type logical and must have the same shape as ARRAY. Fortran 95 has extended MAXVAL such that if only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is a scalar if no DIM argument is present, or has a rank of n-1 and has a shape specified by all of the dimensions except the DIM argument dimension.

The return value is the value of the largest element of the array if no optional parameters are supplied. If only the MASK parameter is supplied with the array, then the return value is the value that is the maximum of the true elements of MASK.

When the DIM argument is supplied, the return value is an array that has a value of MAXVAL applied recursively along the DIM dimensions of the array.

MERGE 90

This function merges two arguments based on the value of a logical mask.

Synopsis

```
MERGE(TSOURCE, FSOURCE, MASK)
```

Arguments

TSOURCE is the source that is merged if the mask is true. FSOURCE is the source that is merged if the mask is false. TSOURCE and FSOURCE must be of the same type and must have the same type parameters (if they are arrays they must be conformable). MASK must be of type logical.

Return Value

The result has the same type and type parameters as the source arrays.

MIN 77

Return the minimum value of the supplied arguments.

Synopsis

```
MIN(A1, A2 [,A3,...])
```

Arguments

Any arguments after the first two are optional. The arguments must all have the same kind and they must be integer or real.

Return Value

The return value is the same as the type and kind of the arguments. It has the value of the smallest argument.

MINEXPONENT 90

Returns the value of the minimum exponent for the type and the kind supplied.

Synopsis

```
MINEXPONENT(X)
```

Argument

The argument X has type real and may be a scalar or an array.

Return Value

The return value is an integer. It contains the value of the smallest exponent in the specified kind.

MINLOC 90

Return the position of the element with the minimum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

Synopsis

```
MINLOC(ARRAY [,DIM] [,MASK])
```

Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM, added in Fortran 95, is of type integer. The optional argument MASK must be of type logical and must have the same shape as ARRAY. If only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is an integer array of rank 1 and has a size equal to the number of dimensions in ARRAY. The return value is the position of the first element that is the minimum value of the array.

If the MASK parameter is present, the return value is the position of the first value that has the minimum value of values in ARRAY, and that also has a true value in the corresponding MASK array.

When the DIM argument is supplied, the return value is an array that has a value of MINLOC applied recursively along the DIM dimensions of the array.

MINVAL 90

Return the minimum value of the elements of the argument array. The test elements may be limited by a dimension argument or by a logical mask argument.

Synopsis

```
MINVAL(ARRAY [,DIM] [,MASK])
```

Arguments

The argument ARRAY must be an array of type integer or real. The optional argument DIM is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and n where n is the rank of the array). The optional argument MASK must be of type logical and must have the same shape as ARRAY. Fortran 95 has extended MINVAL such that if only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is a scalar if no DIM argument is present, or has a rank of n-1 and has a shape specified by all of the dimensions except the DIM argument dimension.

The return value is the value of the largest element of the array if no optional parameters are supplied. If only the MASK parameter is supplied with the array, then the return value is the value that is the minimum of the true elements of MASK.

When the DIM argument is supplied, the return value is an array that has a value of MINVAL applied recursively along the DIM dimensions of the array.

MOD 77

Find the remainder.

Synopsis

```
MOD(A, P)
```

Arguments

The argument A must be an integer or a real. The argument P must be of the same type and kind as A.

Return Value

The return value is the same type as the argument A and has the value (A - INT(A/P) * P).

MODULO 90

Return the modulo value of the arguments.

Synopsis

```
MODULO(A, P)
```

Arguments

The argument A must be an integer or a real. The argument P must be of the same type and kind as A.

Return Value

The return value is the same type as the argument A. If A and P are of type real, the result is (A – FLOOR(A/P) * P). If A and P are of type integer, the result is (A – FLOOR(A÷P) * P) where ÷ represents ordinary mathematical division.

MVBITS 90

Copies a bit sequence from a source data object to a destination data object.

Synopsis

```
MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)
```

Arguments

All arguments have type integer. The arguments FROMPOS, LEN , and TOPOS must be nonnegative. The TO argument must be a variable of type integer and have the same kind type parameter as the FROM argument.

Return Value

MVBITS is a subroutine and has no return value; instead, the TO argument is modified. LEN bits starting at FROMPOS in FROM are copied to TO at TOPOS. All other bits of TO remain unchanged.

NEAREST 90

Returns the nearest different machine representable number in a given direction.

Synopsis

```
NEAREST(X, S)
```

Arguments

The argument X is a real number. The argument S is a real number and not equal to zero.

Return Value

The return value is of the same type as X. It contains the value that is the closest possible different machine representable number from X in the direction given by the sign of S.

NEQV § 77

Performs a logical exclusive OR on the arguments.

Synopsis

```
COMPL(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

NINT 77

Returns the nearest integer to the real argument.

Synopsis

```
NINT(A [,KIND])
```

Arguments

The argument A must be a real. The optional argument KIND specifies the kind of the result integer.

Return Value

The result is an integer. If A > 0, NINT(A) has the value is INT(A+0.5). If A is less than or equal to 0, NINT(A) has the value INT(A-0.5).

NOT 77

Perform a bit-by-bit logical complement on the argument.

Synopsis

```
NOT(I)
```

Argument

The argument I must be of type integer.

Return Value

The return value is an integer value representing a bit-by-bit logical complement of the bits in the argument.

NULL 95

Fortran 95 added this transformational function. NULL gives the disassociated status to pointer entities. For a more detailed explanation, refer to Fortran 95 Explained.

Synopsis

```
NULL([PTR])
```

Arguments

The optional argument PTR is a pointer of any type and may have any association status including undefined.

Return Value

The return value is a disassociated pointer.

OR § 77

Performs a logical OR on each bit of the arguments.

Synopsis

```
OR(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

PACK 90

Pack an array of any number of dimensions into an array of rank one.

Synopsis

```
PACK(ARRAY, MASK [,VECTOR])
```

Arguments

The ARRAY argument is the array to be packed and may be of any type. The MASK argument is of type logical and must be conformable with ARRAY. The optional argument VECTOR is of the same type as ARRAY and has rank one.

Return Value

The return value is the packed array limited by the logical values in the array MASK. If VECTOR is present its values are part of the result array only for those elements that have an element order greater than the number of true elements in MASK. For further details, refer to the Fortran 95 Handbook.

PRECISION 90

Return the decimal precision of the real or complex argument.

Synopsis

```
PRECISION(X)
```

Argument

The argument X must be a real or complex number.

Return Value

The return value is an integer representing the decimal precision of the argument.

PRESENT 90

Determine if an optional argument is present.

Synopsis

```
PRESENT(A)
```

Argument

The argument A must be an optional argument in the procedure in which the intrinsic is called.

Return Value

A logical scalar. True if A is present and false otherwise.

PRODUCT 90

Returns the product of the elements of the supplied array.

Synopsis

```
PRODUCT(ARRAY [,DIM] [,MASK])
```

Arguments

The ARRAY argument is an array of integer, real or complex type. The optional DIM argument is a valid dimension (valid dimensions are between the values 1 and n where n is the rank of the array). The optional MASK argument is of type logical and conformable with the supplied array. Fortran 95 has extended PRODUCT such that if only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is the product of the elements of ARRAY. If the optional DIM argument is present, the product is for the specified dimension. If the optional MASK argument is present, the result is subject to the logical mask supplied.

RADIX 90

Return the base of the model representing numbers of the type and kind of the argument.

Synopsis

```
RADIX(X)
```

Argument

The argument X is of type integer or real.

Return Value

The return value is an integer with the value of the radix (base) of the number system model of the argument.

RANDOM_NUMBER 90

Returns one pseudorandom number or an array of pseudo-random numbers from the uniform distribution over the range 0 x < 1.

Synopsis

```
RANDOM_NUMBER(HARVEST)
```

Argument

The argument HARVEST must of type real. It is set to contain the resulting pseudorandom number or array of pseudorandom numbers from the uniform distribution.

Return Value

RANDOM_NUMBER is a subroutine.

Description

The random number intrinsic generates a 46 bit lagged fibonacci pseudo-random sequence with a short lag of 5 and a long lag of 17. For a given seed, including the default seed, the sequence generated is independent of the platform and number of processors. Due to limitations of some platforms' default integer type, the seed vector is of size 34. Only the least significant 23 bits of each element of the seed array are used, thus a seed array returned or used is portable between platforms. For non-degenerate seed arrays, the period of this generator is (217 - 1) * 245. If all the odd elements of the seed array are even, the period will be shorter.

For the PGHPF compiler, the best performance on distributed arrays is for block distributions. The higher the order of the first distributed dimension, the better the performance will be.

RANDOM_SEED 90

Restarts or queries the pseudorandom number generator for RANDOM_NUMBER.

Synopsis

```
RANDOM_SEED([SIZE] [,PUT] [,GET])
```

Arguments

The arguments SIZE, PUT and GET are optional. There must be one or no arguments. Multiple arguments are not allowed. SIZE is an integer value representing the number of integers that the processor uses to hold the value of the seed. PUT is an integer array of rank one and is used to set the seed. GET is an integer array of rank one and is used to get the value of the seed.

Return Value

RANDOM_SEED is a subroutine.

RANGE 90

Return the decimal exponent range for the type of number supplied as an argument.

Synopsis

```
RANGE(X)
```

Argument

The argument X must be of type integer, real, or complex.

Return Value

The result is an integer.

REAL 77

Convert the argument to real.

Synopsis

```
REAL(A [,KIND])
```

Arguments

The argument A must be of type integer, real, or complex. The optional argument KIND specifies the kind type of the result.

Return Value

The result is a real number. For a complex argument, the imaginary part is ignored.

REPEAT 90

Concatenate copies of a string.

Synopsis

```
REPEAT(STRING, NCOPIES)
```

Arguments

The argument STRING must be a scalar of type character. The argument NCOPIES is an integer.

Return Value

The return value is a character string that is NCOPIES times as long as STRING. It is the concatenation of STRING NCOPIES times.

RESHAPE 90

Reconstructs an array with the specified shape using the elements of the source array.

Synopsis

```
RESHAPE(SOURCE, SHAPE [,PAD] [,ORDER])
```

Arguments

The argument SOURCE is an array of any type. The argument SHAPE is of type integer and has rank one. It must not have more than 7 elements and no values can be negative. The optional argument PAD must be the same size and type as SOURCE. The optional argument ORDER must be of type integer and must have the same shape as SHAPE.

Return Value

The return value is an array of shape SHAPE, with the same type as SOURCE. Array elements are filled into the new array in array element order.

RRSPACING 90

Return the reciprocal of the relative spacing of model numbers near the argument value.

Synopsis

```
RRSPACING(X)
```

Argument

The argument X is of type real.

Return Value

Returns a value of the same type as X.

RSHIFT § 77

Perform a logical shift to the right.

Synopsis

```
RSHIFT(I, SHIFT)
```

Arguments

I and SHIFT are integer values.

Return Value

The return value is of the same type and kind as the argument I. It is the value of the argument I logically shifted right by SHIFT bits.

SCALE 90

Return the value X * bi where b is the base of the number system in use for X.

Synopsis

```
SCALE(X, I)
```

Arguments

The argument X is of type real. The argument I is an integer.

Return Value

The result is a real value of the same type as the argument X.

SCAN 90

Search the supplied string for a character in a set of characters.

Synopsis

```
SCAN(STRING, SET [,BACK])
```

Arguments

The argument STRING is of type character and is the string to search. The argument SET is of type character and has the same kind type parameter as STRING. The optional argument BACK is of type logical.

Return Value

The result value is an integer specifying the position in STRING of a character from SET. If the optional parameter BACK is not present, or is present and false, the result is the position of the first character found. If BACK is present and true, the return value is that of the last character in STRING matching one in SET.

SELECTED_INT_KIND 90

Returns a value that is the kind type parameter that will represent a number in the specified range, where the range is determined by the formula -10R < n < 10R, where n is an integer and R is the argument.

Synopsis

```
SELECTED_INT_KIND(R)
```

Argument

The argument R must be of type integer.

Return Value

The return value is an integer. If the value R is invalid, the return value is -1.

SELECTED_REAL_KIND 90

Returns a value that is the kind type parameter that will represent a number in the specified range, where the range is determined with decimal precision P and a decimal exponent range of at least R.

Synopsis

```
SELECTED_REAL_KIND([P] [,R])
```

Arguments

The arguments are both optional, but at least one of the optional arguments must be present. The argument P must be of type integer and specifies a precision. The argument R must be of type integer and specifies a range.

Return Value

The return value is an integer.

SET_EXPONENT 90

Returns the model number whose fractional part is the fractional part of the model representation of X and whose exponent part is I.

Synopsis

```
SET_EXPONENT(X, I)
```

Arguments

The argument X is of type real. The argument I is of type integer.

Return Value

The result has the type of the argument X.

SHAPE 90

Returns the shape of the supplied argument.

Synopsis

```
SHAPE(SOURCE)
```

Arguments

The argument SOURCE is a scalar or an array of any type.

Return Value

The result is an array whose size is equal to the rank of SOURCE and whose values represent the shape of SOURCE.

SHIFT § 77

Perform a logical shift.

Synopsis

```
RSHIFT(I, SHIFT)
```

Arguments

The argument I may be of any type except character or complex. The argument SHIFT is of type integer.

Return Value

The return value is typeless. If SHIFT is positive, the result is I logically shifted left by SHIFT bits. If SHIFT is negative, the result is I logically shifted right by SHIFT bits.

SIGN 77

Return the absolute value of A times the sign of B.

Synopsis

```
SIGN(A, B)
```

Arguments

The argument A is an integer or real number. The argument B must be of the same type as A.

Return Value

The result is the value of the absolute value of A times the sign of B and has the same type as A. If B is zero, its sign is taken as positive. Fortran 95 allows for a distinction to be made between positive and negative real zeroes. In this case, if B is a real zero, its sign is positive if it is a positive real zero or if the processor cannot distinguish between positive and negative real zeroes.

SIN 77

Return the value of the sine of the argument.

Synopsis

```
SIN(X)
```

Argument

The argument X must be of type real or complex.

Return Value

The return value has the same type as X and is expressed in radians.

SIND 77

Return the value in degrees of the sine of the argument.

Synopsis

```
SIND(X)
```

Argument

The argument X must be of type real or complex.

Return Value

The return value has the same type as X and is expressed in degrees.

SINH 77

Return the hyperbolic sine of the argument.

Synopsis

```
SINH(X)
```

Argument

The argument X must be of type real.

Return Value

The return value has the same type as X.

SIZE 90

Returns either the total number of elements in the array or the number of elements along a specified dimension.

Synopsis

```
SIZE(ARRAY [,DIM])
```

Arguments

The argument ARRAY is an array of any type. The optional DIM argument must be a valid dimension (valid dimensions are between the values 1 and n where n is the rank of the array).

Return Value

The result value is an integer. If DIM is absent, the function returns the total number of elements in the array. If DIM is present, the function returns the extent of the array in the specified dimension.

SPACING 90

Returns the spacing of model numbers near the argument.

Synopsis

```
SPACING(X)
```

Arguments

The argument X is of type real.

Return Value

The return value has the same type and kind as X.

SPREAD 90

Replicates an array by adding a new dimension.

Synopsis

```
SPREAD(SOURCE, DIM, NCOPIES)
```

Arguments

The argument SOURCE may be of any type with rank less than 7. The DIM argument is a scalar integer representing a valid dimension (valid dimensions are between the values 1 and n where n is the rank of the array). The argument NCOPIES must be scalar and of type integer.

Return Value

The return value is an array of the same type as the SOURCE, with rank n+1 where n is the rank of SOURCE.

SQRT 77

Return the square root of the argument.

Synopsis

```
SQRT(X)
```

Arguments

The argument X must be a real or complex number.

Return Value

The result is of the same type as the argument.

SUM 90

Returns the sum of the elements of the supplied array.

Synopsis

```
SUM(ARRAY [,DIM] [,MASK])
```

Arguments

The ARRAY argument is an array of integer, real or complex type. The optional DIM argument is a valid dimension (valid dimensions are between the values 1 and n where n is the rank of the array). The optional MASK argument is of type logical and conformable with the supplied array. Fortran 95 has extended SUM such that if only two arguments are supplied, the type of the second argument is used to determine if it represents DIM or MASK.

Return Value

The return value is the sum of the elements of the argument ARRAY. If the optional DIM argument is present, the sum is for the specified dimension. If the optional MASK argument is present, the result is subject to the logical mask supplied.

SYSTEM_CLOCK 90

Returns information about the real time clock.

Synopsis

```
SYSTEM_CLOCK([COUNT] [,COUNT_RATE] [,COUNT_MAX])
```

Arguments

The optional argument COUNT is a scalar integer that provides the current count of the system clock when the subroutine is called. The optional argument COUNT_RATE is a scalar integer that provides the number of clock ticks per second. The optional argument COUNT_MAX is a scalar integer that provides the value of the maximum count possible.

The number of tics per second is always 1000. This routine is implemented on most systems using gettimeofday(2); some implementations use dclock(3).

Return Value

The arguments of this subroutine are modified during the call; there is no return value.

TAN 77

Return the tangent of the specified value.

Synopsis

```
TAN(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

A real value of the same KIND as the argument.

TAND 77

Return the tangent of the specified value.

Synopsis

```
TAND(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

A real value of the same KIND as the argument.

TANH 77

Return the hyperbolic tangent of the specified value.

Synopsis

```
TANH(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

A real value of the same KIND as the argument.

TINY 90

Return the smallest positive number representable in the kind of the supplied argument.

Synopsis

```
TINY(X)
```

Argument

The argument X must be of type real.

Return Value

The return value is the smallest positive number in the number system and has the same type as the argument X.

TRANSFER 90

Return a value that has the same bit representation as the source but with a different type.

Synopsis

```
TRANSFER(SOURCE, MOLD [,SIZE])
```

Arguments

The arguments SOURCE and MOLD may be scalars or arrays of any type. The optional argument SIZE must be a scalar and of type integer.

Return Value

The return value has the type of the MOLD argument. If SIZE is present, the result is a rank-one array of size SIZE. If SIZE is not present, the result is a scalar if MOLD is a scalar and a rank-one array if MOLD is an array. Refer to The Fortran 95 Handbook for more details on the TRANSFER intrinsic.

TRANSPOSE 90

Transpose an array of rank two.

Synopsis

```
TRANSPOSE(MATRIX)
```

Arguments

The argument MATRIX is a two-dimensional array of any type.

Return Value

The result is a transformed matrix with the same type as MATRIX and dimensions (m,n) where matrix MATRIX has dimensions (n,m).

TRIM 90

Remove the trailing blanks from a string.

Synopsis

```
TRIM(STRING)
```

Arguments

The argument STRING is the string to be adjusted and must be a scalar.

Return Value

The return value is the same as the argument but with the trailing blanks removed. The size of the returned string is the size of the argument STRING minus the number of trailing blanks in STRING.

UBOUND 90

Returns the upper bounds of an array or the upper bound for the specified dimension.

Synopsis

```
UBOUND(ARRAY [,DIM])
```

Arguments

The argument ARRAY is an array of any type. The optional argument DIM is a scalar that has the value of a valid dimension of the array (valid dimensions are between the values 1 and n where n is the rank of the array).

Return Value

The return value is an integer or an array of rank one and size n, where n is the rank of the argument ARRAY. If DIM is not supplied, the return value is an array with all the upper bounds for ARRAY. If DIM is provided, the return value is the value of the upper bound in the specified dimension.

UNPACK 90

Unpack an array of rank one dimension into an array of any number of dimensions.

Synopsis

```
UNPACK(VECTOR, MASK, FIELD)
```

Arguments

The VECTOR argument is an array of any type and of rank one. It must have as many elements as there are true elements in MASK. The MASK argument is of type logical array. The FIELD argument must be the same type as VECTOR and must be conformable with MASK.

Return Value

The result array has the same type as VECTOR and the shape of MASK. For further details and information on the FIELD argument, refer to The Fortran 95 Handbook.

VERIFY 90

Verify that a character string contains all characters from a set of characters.

Synopsis

```
VERIFY(STRING, SET [,BACK])
```

Arguments

The arguments STRING and SET are of type character. The optional argument BACK is of type logical.

Return Value

The return value is an integer. The function returns the position of the first (or last) character that is not in the set. BACK is a logical that determines if the first or last character position is returned. If BACK is present and true, the position of the right-most character is returned. If BACK is not present or present and false, the position of the left-most character is returned.

XOR § 77

Performs a logical exclusive OR on each bit of the arguments.

Synopsis

```
XOR(M, N)
```

Arguments

The arguments M and N must be of integer type.

Return Value

The return value is an integer type.

ZEXT § 77

Zero-extend the argument.

Synopsis

```
ZEXT(A)
```

Arguments

The argument A is of type logical or integer.

Return Value

The return value is an integer.

Supported HPF Intrinsics

The following table lists the HPF intrinsics and Library procedures supported by the PGHPF compiler. Refer to the man pages supplied with the PGHPF software for further details on these intrinsics and procedures. Refer to Chapter 9, "HPF Directives", for the HPF_LIBRARY_LOCAL routines.

Table 6-18: HPF Intrinsics and Library Procedures

Intrinsic	Class
ALL_PREFIX	Transformational function
ALL_SCATTER	Transformational function
ALL_SUFFIX	Transformational function
ANY_PREFIX	Transformational function
ANY_SCATTER	Transformational function
ANY_SUFFIX	Transformational function
COPY_PREFIX	Transformational function
COPY_SCATTER	Transformational function
COPY_SUFFIX	Transformational function
COUNT_PREFIX	Transformational function
COUNT_SCATTER	Transformational function
COUNT_SUFFIX	Transformational function
GRADE_DOWN	Transformational function
GRADE_UP	Transformational function
HPF_ALIGNMENT	Mapping inquiry subroutine
HPF_DISTRIBUTION	Mapping inquiry subroutine
HPF_TEMPLATE	Mapping inquiry subroutine
IALL	Transformational function
IALL_PREFIX	Transformational function
IALL_SCATTER	Transformational function
IALL_SUFFIX	Transformational function

Intrinsic	Class
IANY	Transformational function
IANY_PREFIX	Transformational function
IANY_SCATTER	Transformational function
IANY_SUFFIX	Transformational function
ILEN	Elemental Intrinsic
IPARITY	Transformational function
IPARITY_PREFIX	Transformational function
IPARITY_SCATTER	Transformational function
IPARITY_SUFFIX	Transformational function
LEADZ	Elemental function
MAXLOC	Transformational function Intrinsic
MAXVAL_PREFIX	Transformational function
MAXVAL_SCATTER	Transformational function
MAXVAL_SUFFIX	Transformational function
MINLOC	Transformational function Intrinsic
MINVAL_PREFIX	Transformational function
MINVAL_SCATTER	Transformational function
MINVAL_SUFFIX	Transformational function
NUMBER_OF_PROCESSORS	System Inquiry function Intrinsic
PARITY	Transformational function
PARITY_PREFIX	Transformational function
PARITY_SCATTER	Transformational function

Intrinsic	Class
PARITY_SUFFIX	Transformational function
POPCNT	Elemental function
POPPAR	Elemental function
PROCESSORS_SHAPE	System Inquiry function Intrinsic
PRODUCT_PREFIX	Transformational function
PRODUCT_SCATTER	Transformational function
PRODUCT_SUFFIX	Transformational function
SUM_PREFIX	Transformational function
SUM_SCATTER	Transformational function
SUM_SUFFIX	Transformational function

CM Fortran Intrinsics §

This section provides information on CM Fortran intrinsics. The PGHPF compiler option –Mcmf provides limited support for CM Fortran compatibility (Thinking Machines Corporation version of Fortran). This includes support for the intrinsics DOTPRODUCT, DLBOUND, DUBOUND, and DSHAPE which have calling sequences identical to their Fortran 90/95 counterparts. It also includes support for the CM Fortran method of using square brackets in the definition of array constructors and the use of the ARRAY keyword in place of the Fortran 90/95 standard DIMENSION keyword.

There are three CM Fortran intrinsics which have names identical to their Fortran 90/95 counterparts but whose calling sequences differ. These are CSHIFT, EOSHIFT, and RESHAPE; their descriptions follow.

If PGHPF is invoked with the compiler switch -Mcmf these three intrinsics will be interpreted using the CM Fortran convention rather than the standard Fortran 90/95 convention. There are 6 additional non-standard intrinsics in CM Fortran: PROJECT, LASTLOC, FIRSTLOC, RANK, DIAGONAL, and REPLICATE. These non-standard intrinsics are not supported by PGHPF. Other features of CM Fortran that are not supported are the layout directives and the utility routines.

CSHIFT §

Perform a circular shift on the specified array.

Synopsis

```
CSHIFT(ARRAY, DIM, SHIFT)
```

Arguments

The argument ARRAY is the array to shift. It may be an array of any type. The argument DIM is an integer representing the dimension to shift. The argument SHIFT is an integer or an array of integers with rank n-1 where n is the rank of ARRAY.

Return Value

The shifted array with the same size and shape as the argument ARRAY.

EOSHIFT §

Perform an end-off shift on the specified array.

Synopsis

```
CSHIFT(ARRAY, DIM, SHIFT, BOUNDARY)
```

Arguments

The argument ARRAY is the array to shift. It may be an array of any type. The argument DIM is an integer representing the dimension to shift. The argument SHIFT is an integer or an array of integers with rank n-1 where n is the rank of ARRAY. The optional argument BOUNDARY is of the same type as the array, it may be scalar or of rank n-1 where n is the rank of ARRAY. BOUNDARY is the value to fill in the shifted out positions. By default it has the following values for integer, 0, for real, 0.0, for complex, (0.0,0.0), for logical false, for character the default is blank characters.

Return Value

The shifted array with the same size and shape as the argument ARRAY.

RESHAPE §

Reconstructs an array with the specified shape using the elements of the source array.

Synopsis

```
RESHAPE(SHAPE, SOURCE [,PAD] [,ORDER])
```

Arguments

The argument SHAPE is of type integer, rank one. It must not have more than 7 elements and no values can be negative. The argument SOURCE is an array of any type. The optional argument PAD must be the same size and type as SOURCE. The optional argument ORDER must be of type integer and must have the same shape as SHAPE.

Return Value

The return value is an array of shape SHAPE, with the same type as SOURCE. Array elements are filled into the new array in array element order.

7 3F Functions and VAX Subroutines

The PGI Fortran compilers support FORTRAN 77 3F functions and VAX/VMS system subroutines and built-in functions.

3F Routines

This section describes the functions and subroutines in the Fortran run-time library which are known as 3F routines on many systems. These routines provide an interface from Fortran programs to the system in the same manner as the C library does for C programs. These functions and subroutines are automatically loaded from PGI's Fortran run-time library if referenced by a Fortran program.

The implementation of many of the routines uses functions which reside in the C library. If a C library does not contain the necessary functions, undefined symbol errors will occur at link-time. For example, if PGI's C library is the C library available on the system, the following 3F routines exist in the Fortran run-time library, but use of these routines will result in errors at link-time:

```
besj0
besj1
besjn
besy0
besy1
besyn
dbesj0
dbesj1
dbesjn
dbesy0
dbesy1
dbesyn
derf
derfc
erf
erfc
getlog
hostnm
```

```
lstat
putenv
symlnk
ttynam
```

The routines mclock and times depend on the existence of the C function times().

The routines dtime and etime are only available in a SYSVR4 environment. These routines are not available in all environments simply because there is no standard mechanism to resolve the resolution of the value returned by the times() function.

There are several 3F routines (for example, fputc and fgetc) which perform I/O on a logical unit. These routines bypass normal Fortran I/O. If normal Fortran I/O is also performed on a logical unit which appears in any of these routines, the results are unpredictable.

abort

Terminate abruptly and write memory image to core file.

Synopsis

```
subroutine abort()
```

Description

abort cleans up the I/O buffers and then aborts, producing a core file in the current directory.

access

Determine access mode or existence of a file.

Synopsis

```
integer function access(fil, mode)
character*(*) fil
character*(*) mode
```

Description

The access function if the file, whose name is fil, for accessibility or existence as determined by mode.

The mode argument may include, in any order and in any combination, one or more of:

r test for read permission

w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either the mode argument is illegal or if the file cannot be accessed in all of the specified modes. Zero is returned if the specified access is successful.

alarm

Execute a subroutine after a specified time.

Synopsis

```
integer function alarm(time, proc)
integer time
external proc
```

Description

This routine establishes subroutine proc to be called after time seconds. If time is 0, the alarm is turned off and no routine will be called. The return value of alarm is the time remaining on the last alarm.

Bessel functions

These functions calculate Bessel functions of the first and second kinds for real and double precision arguments and integer orders.

```
besj0
besj1
besjn
besy0
besy1
besyn
dbesj0
dbesj1
dbesjn
dbesy0
dbesy1
dbesyn
```

Synopsis

```
real function besj0(x)
real x
real function besj1(x)
real x

real function besjn(n, x)
integer n
real x
real function besy0(x)
real x
real function besy1(x)
real x
real function besyn(n, x)
integer n
real x
double precision function dbesj0(x)
double precision x
double precision function dbesj1(x)
double precision x
double precision function dbesjn(n, x)
integer n
double precision x
double precision function dbesy0(x)
double precision x
double precision function dbesy1(x)
double precision x
double precision function dbesyn(n, x)
integer n
double precision x
```

chdir

Change default directory.

Synopsis

```
integer function chdir(path)
character*(*) path
```

Description

Change the default directory for creating and locating files to path. Zero is returned if successful; otherwise, an error code is returned.

chmod

Change mode of a file.

Synopsis

```
integer function chmod(nam, mode)
character*(*) nam
integer mode
```

Description

Change the file system mode of file nam. If successful, a value of 0 is returned; otherwise, an error code is returned.

ctime

Return the system time.

Synopsis

```
character*(*) function ctime(stime)
integer stime
```

Description

ctime converts a system time in stime to its ASCII form and returns the converted form. Neither newline nor NULL is included.

date

Return the date.

Synopsis

```
character*(*) function date(buf)
```

Description

Returns the ASCII representation of the current date. The form returned is dd-mmm-yy.

error functions

The functions erf and derf return the error function of x. erfc and derfc return 1.0-erf(x) and 1.0-derf(x), respectively.

Synopsis

```
real function erf(x)
real x
real function erfc(x)
real x
double precision function derf(x)
double precision x
double precision function derfc(x)
double precision x
```

etime, dtime

Get the elapsed time.

Synopsis

```
real function etime(tarray)
real function dtime(tarray)
real tarray(2)
```

Description

etime returns the total processor run-time in seconds for the calling process.

dtime (delta time) returns the processor time since the previous call to dtime. The first time it is called, it returns the processor time since the start of execution.

Both functions place values in the argument tarray: user time in the first element and system time in the second element. The return value is the sum of these two times.

exit

Terminate program with status.

Synopsis

```
subroutine exit(s)
integer s
```

Description

exit flushes and closes all of the program's files, and returns the value of s to the parent process.

fdate

Return date and time in ASCII form.

Synopsis

```
character*(*) function fdate()
```

Description

fdate returns the current date and time as a character string. Neither newline nor NULL will be included.

fgetc

Get character from a logical unit.

Synopsis

```
integer function fgetc(lu, ch)
integer lu
character*(*) ch
```

Description

Returns the next character in ch from the file connected to the logical unit lu, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

flush

Flush a logical unit.

Synopsis

```
subroutine flush(lu)
integer lu
```

Description

flush flushes the contents of the buffer associated with logical unit lu.

fork

Fork a process.

Synopsis

```
integer function fork()
```

Description

fork creates a copy of the calling process. The value returned to the parent process will be the process id of the copy. The value returned to the child process (the copy) will be zero. If the returned value is negative, an error occurred and the value is the negation of the system error code.

fputc

Write a character to a logical unit.

Synopsis

```
integer function fputc(lu, ch)
integer lu
character*(*) ch
```

Description

A character ch is written to the file connected to logical unit lu bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

free

Free memory.

Synopsis

```
subroutine free(p)
int p
```

Description

Free a pointer to a block of memory located by malloc; the value of the argument, p, is the pointer to the block of memory.

fseek

Position file at offset.

Synopsis

```
integer function fseek(lu, offset, from)
integer lu
integer offset
integer from
```

Description

fseek repositions a file connected to logical unit lu. offset is an offset in bytes relative to the position specified by from :

0	beginning of the file
1	current position
2	end of the file

If successful, the value returned by fseek will be zero; otherwise, it's a system error code.

ftell

Determine file position.

Synopsis

```
integer function ftell(lu)
integer lu
```

Description

ftell returns the current position of the file connected to the logical unit lu. The value returned is an offset, in units of bytes, from the beginning of the file. If the value returned is negative, it is the negation of the system error code.

gerror

Return system error message.

Synopsis

```
character*(*) function gerror()
```

Description

Return the system error message of the last detected system error.

getarg

Get the nth command line argument.

Synopsis

```
subroutine getarg(n, arg)
integer n
character*(*) arg
```

Description

Return the nth command line argument in arg, where the 0th argument is the command name.

iargc

Index of the last command line argument.

Synopsis

```
integer function iargc()
```

Description

Return the index of the last command line argument, which is also the number of arguments after the command name.

getc

Get character from unit 5.

Synopsis

```
integer function getc(ch)
character*(*) ch
```

Description

Returns the next character in ch from the file connected to the logical unit 5, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

getcwd

Get pathname of current working directory.

Synopsis

```
integer function getcwd(dir)
character*(*) dir
```

Description

The pathname of the current working directory is returned in dir. If successful, the return value is zero; otherwise, an error code is returned.

getenv

Get value of environment variable.

Synopsis

```
subroutine getenv(en, ev)
character*(*) en
character*(*) ev
```

Description

getenv checks for the existence of the environment variable en. If it does not exist or if its value is not present, ev is filled with blanks. Otherwise, the string value of en is returned in ev.

getgid

Get group id.

Synopsis

```
integer function getgid()
```

Description

Return the group id of the user of the process.

getlog

Get user's login name.

Synopsis

```
character*(*) function getlog()
```

Description

getlog returns the user's login name or blanks if the process is running detached from a terminal.

getpid

Get process id.

Synopsis

```
integer function getpid()
```

Description

Return the process id of the current process.

getuid

Get user id.

Synopsis

```
integer function getuid()
```

Description

Return the user id of the user of the process.

gmtime

Return system time.

Synopsis

```
subroutine gmtime(stime, tarray)
integer stime
integer tarray(9)
```

Description

Dissect the UNIX time, stime , into month, day, etc., for GMT and return in tarray.

hostnm

Get name of current host.

Synopsis

```
integer function hostnm(nm)
character*(*) nm
```

Description

hostnm returns the name of the current host in nm. If successful, a value of zero is returned; otherwise an error occurred.

idate

Return date in numerical form.

Synopsis

```
subroutine idate(im, id, iy)
integer im, id, iy
```

Description

Returns the current date in the variables im, id, and iy, which indicate the month, day, and year, respectively. The month is in the range 1-12; only the last 2 digits of the year are returned.

ierrno

Get error number.

Synopsis

```
integer function ierrno()
```

Description

Return the number of the last detected system error.

ioinit

Initialize I/O

Synopsis

```
subroutine ioinit(cctl, bzro, apnd, prefix, vrbose)
integer cctl
integer bzro
integer apnd
character*(*) prefix
integer vrbose
```

Description

Currently, no action is performed.

isatty

Is logical unit a tty.

Synopsis

```
logical function isatty(lu)
integer lu
```

Description

Returns .TRUE. if logical unit lu is connected to a terminal; otherwise, .FALSE. is returned.

itime

Return time in numerical form.

Synopsis

```
subroutine itime(iarray)
integer iarray(3)
```

Description

Return current time in the array iarray. The order is hour, minute, and second.

kill

Send signal to a process.

Synopsis

```
integer function kill(pid, sig)
integer pid
integer sig
```

Description

Send signal number sig to the process whose process id is pid. If successful, the value zero is returned; otherwise, an error code is returned.

link

Make link

Synopsis

```
integer function link(n1, n2)
character*(*) n1
character*(*) n2
```

Description

Create a link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

lnblnk

Return index of last non-blank.

Synopsis

```
integer function lnblnk(a1)
character*(*) a1
```

Description

Return the index of the last non-blank character in string a1.

loc

Address of an object.

Synopsis

```
integer function loc(a)
integer a
```

Description

Return the value which is the address of a.

ltime

Return system time.

Synopsis

```
subroutine ltime(stime, tarray)
integer stime
integer tarray(9)
```

Description

Dissect the UNIX time, stime , into month, day, etc., for the local time zone and return in tarray.

malloc

Allocate memory.

Synopsis

```
integer function malloc(n)
integer n
```

Description

Allocate a block of n bytes of memory and return the pointer to the block of memory.

mclock

Get elapsed time.

Synopsis

```
integer function mclock()
```

Description

mclock returns the sum of the user's cpu time and the user and system times of all child processes. The return value is in units of clock ticks per second.

mvbits

Move bits.

Synopsis

```
subroutine mvbits(src, pos, len, dest, posd)
integer src
integer pos
integer len
integer dest
integer posd
```

Description

len bits are moved beginning at position pos of argument src to position posd of argument dest.

outstr

Print a character string.

Synopsis

```
integer function outstr(ch)
character*(*) ch
```

Description

Output the character string to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error occurred.

perror

Print error message.

Synopsis

```
subroutine perror(str)
character*(*) str
```

Description

Write the message indicated by str to logical unit 0 and the message for the last detected system error.

putc

Write a character to logical unit 6.

Synopsis

```
integer function putc(ch)
character*(*) ch
```

Description

A character ch is written to the file connected to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

putenv

Change or add environment variable.

Synopsis

```
integer function putenv(str)
character*(*) str
```

Description

str contains a character string of the form name=value. This function makes the value of the environment variable name equal to value. If successful, zero is returned.

qsort

Quick sort.

Synopsis

```
subroutine qsort(array, len, isize, compar)
dimension array(*)
integer len
integer isize
external compar
integer compar
```

Description

qsort sorts the elements of the one dimensional array, array. len is the number of elements in the array and isize is the size of an element. compar is the name of an integer function that determines the sorting order. This function is called with 2 arguments (arg1 and arg2) which are elements of array. The function returns:

negative	if arg1 is considered to precede arg2
zero	if arg1 is equivalent to arg2
positive	if arg1 is considered to follow arg2

rand, irand, srand

Random number generator.

Synopsis

```
double precision function rand()
integer function irand()
subroutine srand(iseed)
integer iseed
```

Description

The functions rand and irand generates successive pseudo-random integers or double precision numbers. srand uses its argument, iseed, to re-initialize the seed for successive invocations of rand and irand.

irand	returns a positive integer in the range 0 through 2147483647.
rand	returns a value in the range 0 through 1.0.

random, irandm, drandm

Return the next random number value. If the argument, flag , is nonzero, the random number generator is restarted before the next random number is generated. Integer values will range from 0 thru 2147483647; floating point values will range from 0.0 thru 1.0.

Synopsis

```
real function random(flag)
integer flag
integer function irandm(flag)
integer flag
double precision function drandm(flag)
integer flag
```

range

Range functions.

Synopsis

```
real function flmin()
real function flmax()
real function ffrac()
double precision function dflmin()
double precision function dflmax()
double precision function dffrac()
integer function inmax()
```

Description

flmin	minimum single precision value
flmax	maximum single precision value
ffrac	smallest positive single precision value
dflmin	minimum double precision value
dflmax	maximum double precision value
dffrac	smallest positive double precision value
inmax	maximum integer

rename

Rename a file.

Synopsis

```
integer function rename(from, to)
character*(*) from
character*(*) to
```

Description

Rename the existing file from where the new name is to. If successful, zero is returned; otherwise, the return value is an error code.

rindex

Return index of substring.

Synopsis

```
integer function rindex(a1, a2)
character*(*) a1
character*(*) a2
```

Description

Return the index of the last occurrence of string a2 in string a1.

secnds, dsecnds

Return elapsed time.

Synopsis

```
real function secnds(x)
real x
double precision function dsecnds(x)
double precision x
```

Description

Returns the elapsed time, in seconds, since midnight, minus the value of x.

setvbuf3f

Change I/O buffering behavior.

Synopsis

```
interface
 function setvbuf3f(lu, typ, size)
 integer setvbuf3f, lu, typ, size
 end function
 end interface
```

Description

Fortran I/O supports 3 types of buffering:

- Fully buffered: on output, data is written once the buffer is full. On input, the buffer is filled when an input operation is requested and the buffer is empty.
- Line buffered: on output, data is written when a newline character is inserted in the buffer or when the buffer is full. On input, if an input operation is encountered and the buffer is empty, the buffer is filled until a newline character is encountered.
- Unbuffered: No buffer is used. Each I/O operation is completed as sopon as possible. In this case, the typ and size arguments are ignored.

Logical units 5 (stdin) and 6 (stdout) are line buffered. Logical unit 0 (stderr) is unbuffered. Disk files are fully buffered. These defaults generally give the expected behavior. You can use setvbuf3f to change a unit's buffering type and size of the buffer.

This function must be called after the unit is opened and before any I/O is done on the unit.

The typ parameter can have the following values, 0 specifies full buffering, 1 specifies line buffering, and 2 specifies unbuffered. The size parameter specifies the size of the buffer. Note, the underlying stdio implementation may silently restrict your choice of buffer size.

This function will return zero on success and non-zero on failure.

An example of a program in which this function might be useful is a long-running program that periodically writes a small amount of data to a log file. If the log file is line buffered, you could check the log file for progress. If the log file is fully buffered (the default), the data may not be written to disk until the program terminates.

signal

Signal facility.

Synopsis

```
integer function signal(signum, proc, flag)
integer signum
external proc
integer flag
```

Description

signal allows the calling process to choose how the receipt of a specific signal is handled; signum is the signal and proc is the choice. If flag is negative, proc is a Fortran subprogram and is established as the signal handler for the signal. Otherwise, proc is ignored and the value of flag is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. There are two special cases of flag: 0 means use the default action and 1 means ignore this signal.

The return value is the previous action. If this is a value greater than one, then it is the address of a routine that was to have been called. The return value can be used in subsequent calls to signal to restore a previous action. A negative return value indicates a system error.

sleep

Suspend execution for a period of time.

Synopsis

```
subroutine sleep(itime)
integer itime
```

Description

Suspends the process for t seconds.

stat, lstat, fstat

Get file status.

Synopsis

```
integer function stat(nm, statb)
character*(*) nm
integer statb(*)
integer function lstat(nm, statb)
character*(*) nm
```

```
integer statb(*)
integer function fstat(lu, statb)
integer lu
integer statb(*)
```

Description

Return the file status of the file in the array statb. If successful, zero is returned; otherwise, the value of -1 is returned. stat obtains information about the file whose name is nm; if the file is a symbolic link, information is obtained about the file the link references. lstat is similar to stat except lstat returns information about the link. fstat obtains information about the file which is connected to logical unit lu.

stime

Set time.

Synopsis

```
integer function stime(tp)
integer tp
```

Description

Set the system time and date. tp is the value of the time measured in seconds from 00:00:00 GMT January 1, 1970.

symlnk

Make symbolic link.

Synopsis

```
integer function symlnk(n1, n2)
character*(*) n1
character*(*) n2
```

Description

Create a symbolic link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

system

Issue a shell command.

Synopsis

```
integer function system(str)
character*(*) str
```

Description

system causes the string, str, to be given to the shell as input. The current process waits until the shell has completed and returns the exit status of the shell.

time

Return system time.

Synopsis

```
integer function time()
```

Description

Return the time since 00:00:00 GMT, January 1, 1970, measured in seconds.

times

Get process and child process time

Synopsis

```
integer function times(buff)
integer buff(*)
```

Description

Returns the time-accounting information for the current process and for any terminated child processes of the current process in the array buff. If successful, zero is returned; otherwise, the negation of the error code is returned.

ttynam

Get name of a terminal

Synopsis

```
character*(*) ttynam(lu)
integer lu
```

Description

Returns a blank padded path name of the terminal device connected to the logical unit lu. The lu is not connected to a terminal, blanks are returned.

unlink

Remove a file.

Synopsis

```
integer function unlink(fil)
character*(*) fil
```

Description

Removes the file specified by the pathname fil. If successful, zero is returned; otherwise, an error code is returned.

wait

Wait for process to terminate.

Synopsis

```
integer function wait(st)
integer st
```

Description

wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last wait, return is immediate. If there are no child processes, return is immediate with an error code.

If the return value is positive, it is the process id of the child and st is its termination status. If the return value is negative, it is the negation of an error code.

VAX System Subroutines

The PGI FORTRAN77 compiler, pgf77, supports a variety of VAX/VMS system subroutines and built-in functions.

Built-In Functions

The built-in functions perform inter-language utilities for argument passing and location calculations. The following built-in functions are available:

%LOC(arg)

Compute the address of the argument arg.

%REF(a)

Pass the argument a by reference.

%VAL(a)

Pass the argument as a 32-bit immediate value (64-bit if a is double precision.) A value of 64-bits is also possible if supported for integer and logical values.

VAX/VMS System Subroutines

DATE

The DATE subroutine returns a nine-byte string containing the ASCII representation of the current date. It has the form:

```
CALL DATE(buf)
```

where buf is a nine-byte variable, array, array element, or character substring. The date is returned as a nine-byte ASCII character string of the form:

```
dd-mmm-yy
```

Where:

dd	is the two-digit day of the month
mmm	is the three-character abbreviation of the month
yy	is the last two digits of the year

EXIT

The EXIT subroutine causes program termination, closes all open files, and returns control to the operating system. It has the form:

```
CALL EXIT[(exit_status)]
```

where:

exit_status

GETARG

The GETARG subroutine returns the Nth command line argument in character variable ARG. For N equal to zero, the name of the program is returned.

```
SUBROUTINE GETARG(N, ARG)
INTEGER*4 N
CHARACTER*(*) ARG
```

IARGC

The IARGC subroutine returns the number of command line arguments following the program name.

```
INTEGER*4 FUNCTION IARGC()
```

IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. It has the form:

```
CALL IDATE(IMONTH, IDAY, IYEAR)
```

If the current date were October 9, 2004, the values of the integer variables upon return would be:

```
IMONTH = 10
IDAY = 9
IYEAR = 04
```

MVBITS

The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). MVBITS transfers a3 bits from positions a2 through (a2 + a3 - 1) of the source, src, to positions a5 through (a5 + a3 - 1) of the destination, dest. Other bits of the destination location remain unchanged. The values of (a2 + a3) and (a5 + a3) must be less than or equal to 32 (less than or equal to 64 if the source or destination is INTEGER*8). It has the form:

```
CALL MVBITS(src, a2, a3, dest, a5)
```

Where:

src	is an integer variable or array element that represents the source location.
a2	is an integer expression that identifies the first position in the field transferred from `src`.
a3	is an integer expression that identifies the length of the field transferred from `src`.
dest	is an integer variable or array element that represents the destination location.
a5	is an integer expression that identifies the starting position within `a4`, for the bits being transferred.

RAN

The RAN subroutine returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1. The result is a floating point number that is uniformly distributed in the range between 0.0 and 1.0 exclusive. It has the form:

```
y = RAN(i)
```

where y is set equal to the value associated by the function with the seed argument i. The argument i must be an INTEGER*4 variable or INTEGER*4 array element.

The argument i should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 6969 * SEED + 1 ! MOD
2**32
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

If the command-line option to treat all REAL declarations as DOUBLE PRECISION declarations is in effect, RAN returns a DOUBLE PRECISION value.

SECNDS

The SECNDS subroutine provides system time of day, or elapsed time, as a floating point value in seconds. It has the form:

```
y = SECNDS(x)
```

where (REAL or DOUBLE PRECISION) y is set equal to the time in seconds since midnight, minus the user supplied value of the (REAL or DOUBLE PRECISION) x. Elapsed time computations can be performed with the following sequence of calls.

```
X = SECNDS(0.0)
...
...! Code to be timed
...
DELTA = SECNDS(X)
```

The accuracy of this call is the same as the resolution of the system clock.

TIME

The TIME subroutine returns the current system time as an ASCII string. It has the form:

```
CALL TIME(buf)
```

where buf is an eight-byte variable, array, array element, or character substring. The TIME call returns the time as an eight-byte ASCII character string of the form:

```
hh:mm:ss
```

For example:

```
16:45:23
```

Note that a 24-hour clock is used.

8 OpenMP Directives for Fortran

The PGF77 and PGF95 Fortran compilers support the OpenMP Fortran Application Program Interface. The OpenMP shared-memory parallel programming model is defined by a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran programs. The directives include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs. The data environment is controlled using clauses on the directives or with additional directives. Run-time library routines are provided to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region. Finally, environment variables are provided to control the execution behavior of parallel programs. For more information on OpenMP, see

http://www.openmp.org

For an introduction to how to execute programs that use multiple processors along with some pointers to example code, see "Parallel Programming Using PGI Compilers" in the PGI User's Guide.

Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the PGI Fortran compilers when the option -mp is specified on the command line. The form of a parallelization directive is:

```
sentineldirective_name[clauses]
```

With the exception of the SGI-compatible DOACROSS directive, the sentinel must be !OMP, COMP, or *$OMP, must start in column 1 (one), and must appear as a single word without embedded white space. The sentinel marking a DOACROSS directive is C$. Standard Fortran syntax restrictions (line length, case insensitivity, etc.) apply to the directive line. Initial directive lines must have a space or zero in column six and continuation directive lines must have a character other than space or zero in column six. Continuation lines for C$DOACROSS directives are specified using the C$& sentinel.

The order in which clauses appear in the parallelization directives is not significant. Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause. Clauses on directives may be repeated as needed subject to the restrictions listed in the description of each clause.

The compiler option -mp enables recognition of the parallelization directives. The use of this option also implies:

-Mreentrant	local variables are placed on the stack and optimizations that may result in non-reentrant code are disabled (e.g., -Mnoframe);
-Miomutex	critical sections are generated around Fortran I/O statements.

Many of the directives are presented in pairs and must be used in pairs. In the examples given with each section, the routines omp_get_num_threads() and omp_get_thread_num() are used; refer to Run-time Library Routines for more information. These routines return the number of threads currently in the team executing the parallel region and the thread number within the team, respectively.

PARALLEL ... END PARALLEL

The OpenMP PARALLEL END PARALLEL directive is supported using the following syntax.

Syntax:

```
!$OMP PARALLEL [Clauses]
< Fortran code executed in body of parallel region >
!$OMP END PARALLEL
```

Clauses:

```
PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
REDUCTION([{operator | intrinsic}:] list)
COPYIN (list)
IF (scalar_logical_expression)
```

This directive pair declares a region of parallel execution. It directs the compiler to create an executable in which the statements between PARALLEL and END PARALLEL are executed by multiple lightweight threads. The code that lies between PARALLEL and END PARALLEL is called a parallel region.

The OpenMP parallelization directives support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered. At the entrance to the parallel region, a system-dependent number of symmetric parallel threads begin executing all statements in the parallel region redundantly. These threads share work by means of work-sharing constructs such as parallel DO loops in the following example. The number of threads in the team is controlled by the OMP_NUM_THREADS environment variable. If OMP_NUM_THREADS is not defined, the program will execute parallel regions using only one processor. Branching into or out of a parallel region is not supported.

All other shared-memory parallelization directives must occur within the scope of a parallel region. Nested PARALLEL...END PARALLEL directive pairs are not supported and are ignored. The END PARALLEL directive denotes the end of the parallel region, and is an implicit barrier. When all threads have completed execution of the parallel region, a single thread resumes execution of the statements that follow.

Note that, by default, there is no work distribution in a parallel region. Each active thread executes the entire region redundantly until it encounters a directive that specifies work distribution. For work distribution, see the DO, PARALLEL DO, or DOACROSS directives.

```
      PROGRAM WHICH_PROCESSOR_AM_I
      INTEGER A(0:1)
      INTEGER omp_get_thread_num
      A(0) = -1
      A(1) = -1
!$OMP PARALLEL
      A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP END PARALLEL
      PRINT *, "A(0)=",A(0), " A(1)=",A(1)
      END
```

The variables specified in a PRIVATE list are private to each thread in a team. In effect, the compiler creates a separate copy of each of these variables for each thread in the team. When an assignment to a private variable occurs, each thread assigns to its local copy of the variable. When operations involving a private variable occur, each thread performs the operations using its local copy of the variable.

Important points about private variables are:

- Variables declared private in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand side expression, the results of the expression will be undefined (i.e., this is probably a coding error).

- Likewise, variables declared private in a parallel region are undefined when serial execution resumes at the end of the parallel region.

The variables specified in a SHARED list are shared between all threads in a team, meaning that all threads access the same storage area for SHARED data.

The DEFAULT clause lets you specify the default attribute for variables in the lexical extent of the parallel region. Individual clauses specifying PRIVATE, SHARED, etc. status override the declared DEFAULT. Specifying DEFAULT(NONE) declares that there is no implicit default, and in this case, each variable in the parallel region must be explicitly listed with an attribute of PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

Variables that appear in the list of a FIRSTPRIVATE clause are subject to the same semantics as PRIVATE variables, but in addition, are initialized from the original object existing prior to entering the parallel region. Variables that appear in the list of a REDUCTION clause must be SHARED. A private copy of each variable in list is created for each thread as if the PRIVATE clause had been specified. Each private copy is initialized according to the operator as specified in the following table:

Table 8-1: Initialization of REDUCTION Variables

Operator / Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	smallest representable number
MIN	largest representable number
IAND	all bits on
IOR	0
IEOR	0

At the end of the parallel region, a reduction is performed on the instances of variables appearing in list using operator or intrinsic as specified in the REDUCTION clause. The initial value of each REDUCTION variable is included in the reduction operation. If the {operator | intrinsic}: portion of the REDUCTION clause is omitted, the default reduction operator is "+" (addition).

The COPYIN clause applies only to THREADPRIVATE common blocks. In the presence of the COPYIN clause, data from the master thread's copy of the common block is copied to the threadprivate copies upon entry to the parallel region.

In the presence of an IF clause, the parallel region will be executed in parallel only if the corresponding scalar_logical_expression evaluates to .TRUE.. Otherwise, the code within the region will be executed by a single processor regardless of the value of the environment variable OMP_NUM_THREADS.

CRITICAL ... END CRITICAL

The OpenMP END CRITICAL directive uses the following syntax.

```
!$OMP CRITICAL [(name)]
< Fortran code executed in body of critical section >
!$OMP END CRITICAL [(name)]
```

Within a parallel region, you may have code that will not execute properly when multiple threads act upon the same sub-region of code. This is often due to a shared variable that is written and then read again.

The CRITICAL...END CRITICAL directive pair defines a subsection of code within a parallel region, referred to as a critical section, which will be executed one thread at a time. The optional name argument identifies the critical section. The first thread to arrive at a critical section will be the first to execute the code within the section. The second thread to arrive will not begin execution of statements in the critical section until the first thread has exited the critical section. Likewise, each of the remaining threads will wait its turn to execute the statements in the critical section.

Critical sections cannot be nested, and any such specifications are ignored. Branching into or out of a critical section is illegal. If a name argument appears on a CRITICAL directive, the same name must appear on the END CRITICAL directive.

```
 PROGRAM CRITICAL_USE
 REAL A(100,100), MX, LMX
 INTEGER I, J
 MX = -1.0
 LMX = -1.0
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(A)
!$OMP PARALLEL PRIVATE(I), FIRSTPRIVATE(LMX)
!$OMP DO
 DO J=1,100
 DO I=1,100
 LMX = MAX(A(I,J), LMX)
 END DO
 END DO
!$OMP CRITICAL
 MX = MAX(MX, LMX)
```

```
!$OMP END CRITICAL
!$OMP END PARALLEL
 PRINT *, "MAX VALUE OF A IS ", MX
 END
```

Note that this program could also be implemented without the critical region by declaring MX as a reduction variable and performing the MAX calculation in the loop using MX directly rather than using LMX. See "DO ... END DO" on page 302, and "PARALLEL ... END PARALLEL" on page 296, for more information on how to use the REDUCTION clause on a parallel DO loop.

MASTER ... END MASTER

The OpenMP MASTER...END MASTER directive uses the following syntax.

```
!$OMP MASTER
< Fortran code in body of MASTER section >
!$OMP END MASTER
```

In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before the sub-region and then starting it up again after the sub-region, the MASTER...END MASTER directive pair lets you conveniently designate code that executes on the master thread and is skipped by the other threads. There is no implied barrier on entry to or exit from a MASTER...END MASTER section of code. Nested master sections are ignored. Branching into or out of a master section is not supported.

```
 PROGRAM MASTER_USE
 INTEGER A(0:1)
 INTEGER omp_get_thread_num
 A=-1
!$OMP PARALLEL
 A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP MASTER
 PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END MASTER
!$OMP END PARALLEL
 PRINT *, "A(0)=", A(0), " A(1)=",
A(1)
 END
```

SINGLE ... END SINGLE

The OpenMP SINGLE...END SINGLE directive uses the following syntax:

```
!$OMP SINGLE [Clauses]
< Fortran code in body of SINGLE processor section >
!$OMP END SINGLE [NOWAIT]
```

Clauses:

```
PRIVATE(list)
FIRSTPRIVATE(list)
```

In a parallel region of code, there may be a sub-region of code that will only execute correctly on a single thread. Instead of ending the parallel region before the sub-region and then starting it up again after the sub-region, the SINGLE...END SINGLE directive pair lets you conveniently designate code that executes on a single thread and is skipped by the other threads. There is an implied barrier on exit from a SINGLE...END SINGLE section of code unless the optional NOWAIT clause is specified.

Nested single process sections are ignored. Branching into or out of a single process section is not supported.

```
 PROGRAM SINGLE_USE
 INTEGER A(0:1)
 INTEGER omp_get_thread_num()
!$OMP PARALLEL
 A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP SINGLE
 PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
 PRINT *, "A(0)=", A(0), " A(1)=",
A(1)
 END
```

The PRIVATE and FIRSTPRIVATE clauses are as described in the PARALLEL...END PARALLEL section.

DO ... END DO

The OpenMP DO...END DO directive uses the following syntax.

Syntax:

```
!$OMP DO [Clauses ]
< Fortran DO loop to be executed in parallel >
!$OMP END DO [NOWAIT]
```

Clauses:

```
PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic } : list)
SCHEDULE (type [, chunk])
ORDERED
```

The real purpose of supporting parallel execution is the distribution of work across the available threads. You can explicitly manage work distribution with constructs such as:

```
IF (omp_get_thread_num() .EQ.
0) THEN
 ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
 ...
ENDIF
```

However, these constructs are not in the form of directives. The DO...END DO directive pair provides a convenient mechanism for the distribution of loop iterations across the available threads in a parallel region. Items to note about clauses are:

Variables declared in a PRIVATE list are treated as private to each processor participating in parallel execution of the loop, meaning that a separate copy of the variable exists on each processor.

Variables declared in a FIRSTPRIVATE list are PRIVATE, and in addition are initialized from the original object existing before the construct.

Variables declared in a LASTPRIVATE list are PRIVATE, and in addition the thread that executes the sequentially last iteration updates the version of the object that existed before the construct.

The REDUCTION clause is as described in the PARALLEL...END PARALLEL section.

The SCHEDULE clause is explained in the following section.

If ORDERED code blocks are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ORDERED code blocks, see the ORDERED section.

The DO...END DO directive pair directs the compiler to distribute the iterative DO loop immediately following the !$OMP DO directive across the threads available to the program. The DO loop is executed in parallel by the team that was started by an enclosing parallel region. If the !$OMP END DO directive is not specified, the !$OMP DO is assumed to end with the enclosed DO loop. DO...END DO directive pairs may not be nested. Branching into or out of a !$OMP DO loop is not supported.

By default, there is an implicit barrier after the end of the parallel loop; the first thread to complete its portion of the work will wait until the other threads have finished their portion of work. If NOWAIT is specified, the threads will not synchronize at the end of the parallel loop.

Other items to note about !$OMP DO loops:

- The DO loop index variable is always private.
- !$OMP DO loops must be executed by all threads participating in the parallel region or none at all.
- The END DO directive is optional, but if it is present it must appear immediately after the end of the enclosed DO loop.

```
 PROGRAM DO_USE
 REAL A(1000), B(1000)
 DO I=1,1000
 B(I) = FLOAT(I)
 END DO
!$OMP PARALLEL
!$OMP DO
 DO I=1,1000
 A(I) = SQRT(B(I));
 END DO
 ...
!$OMP END PARALLEL
 ...
 END
```

The SCHEDULE clause specifies how iterations of the DO loop are divided up between processors. Given a SCHEDULE (type [, chunk]) clause, type can be STATIC, DYNAMIC, GUIDED, or RUNTIME.

These are defined as follows:

When SCHEDULE (STATIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. The blocks of iterations are statically assigned to threads in a round-robin fashion in order of the thread ID numbers. The chunk must be a scalar integer expression. If chunk is not specified, a default chunk size is chosen equal to:

```
(number_of_iterations + omp_num_threads()
- 1) / omp_num_threads()
```

When SCHEDULE (DYNAMIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. The chunk must be a scalar integer expression. If no chunk is specified, a default chunk size is chosen equal to 1.

When SCHEDULE (GUIDED, chunk) is specified, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. Chunk specifies the minimum number of iterations to dispatch each time, except when there are less than chunk iterations remaining to be processed, at which point all remaining iterations are assigned. If no chunk is specified, a default chunk size is chosen equal to 1.

When SCHEDULE (RUNTIME) is specified, the decision regarding iteration scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is equivalent to SCHEDULE(STATIC).

BARRIER

The OpenMP BARRIER directive uses the following syntax.

```
!$OMP BARRIER
```

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The BARRIER directive synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The BARRIER directive must either be executed by all threads executing the parallel region or by none of them.

DOACROSS

The C$DOACROSS directive is not part of the OpenMP standard, but is supported for compatibility with programs parallelized using legacy SGI-style directives.

Syntax:

```
C$DOACROSS [ Clauses ]
< Fortran DO loop to be executed in parallel >
```

Clauses:

```
[ {PRIVATE | LOCAL} (list) ]
[ {SHARED | SHARE} (list) ]
[ MP_SCHEDTYPE={SIMPLE | INTERLEAVE} ]
[ CHUNK=<integer_expression> ]
[ IF (logical_expression) ]
```

The C$DOACROSS directive has the effect of a combined parallel region and parallel DO loop applied to the loop immediately following the directive. It is very similar to the OpenMP PARALLEL DO directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort. The C$DOACROSS directive must not appear within a parallel region. It is a shorthand notation that tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region. While this syntax is more convenient, it should be noted that if multiple successive DO loops are to be parallelized it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP DO directive.

A variable declared PRIVATE or LOCAL to a C$DOACROSS loop is treated the same as a private variable in a parallel region or DO (see above). A variable declared SHARED or SHARE to a C$DOACROSS loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as PRIVATE in a parallel region or DO (this same default status is used in C$DOACROSS loops as well).

PARALLEL DO

The OpenMP PARALLEL DO directive uses the following syntax.

Syntax:

```
!$OMP PARALLEL DO [CLAUSES]
< Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

Clauses:

```
PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF (scalar_logical_expression)
SCHEDULE (type [, chunk])
ORDERED
```

The semantics of the PARALLEL DO directive are identical to those of a parallel region containing only a single parallel DO loop and directive. Note that the END PARALLEL DO directive is optional. The available clauses are as defined in the DO...END DO and PARALLEL...END PARALLEL sections.

SECTIONS ... END SECTIONS

The OpenMP SECTIONS...END SECTIONS directive pair uses the following syntax:

Syntax:

```
!$OMP SECTIONS [ Clauses ]
[!$OMP SECTION]
< Fortran code block executed by processor i >
[!$OMP SECTION]
< Fortran code block executed by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

Clauses:

```
PRIVATE (list)
FIRSTPRIVATE (list)
LASTPRIVATE (list)
REDUCTION({operator | intrinsic} : list)
```

The SECTIONS...END SECTIONS directive pair defines a non-iterative work-sharing construct within a parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing SECTIONS...END SECTIONS directives. In addition, the code within the SECTIONS...END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

The available clauses are as defined in the DO...END DO and PARALLEL...END PARALLEL sections.

PARALLEL SECTIONS

The OpenMP PARALLEL SECTIONS...END SECTIONS directive pair uses the following syntax:

Syntax:

```
!$OMP PARALLEL SECTIONS [CLAUSES]
[!$OMP SECTION]
< Fortran code block executed by processor i >
[!$OMP SECTION]
< Fortran code block executed by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

Clauses:

```
PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF (scalar_logical_expression)
```

The PARALLEL SECTIONS...END SECTIONS directives define a non-iterative work-sharing construct without the need to define an enclosing parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing PARALLEL SECTIONS...END SECTIONS directives. In addition, the code within the PARALLEL SECTIONS...END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

The available clauses are as defined in DO...END DO and PARALELL...END PARALLEL sections.

ORDERED

The OpenMP ORDERED directive is supported using the following syntax:

```
!$OMP ORDERED
< Fortran code block executed by processor >
!$OMP END ORDERED
```

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The code block between the ORDERED...END ORDERED directives is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ORDERED directive:

- The ORDERED code block must be a structured block. It is illegal to branch into or out of the block.
- A given iteration of a loop with a DO directive cannot execute the same ORDERED directive more than once, and cannot execute more than one ORDERED directive.

ATOMIC

The OpenMP ATOMIC directive uses following syntax:

```
!$OMP ATOMIC
```

The ATOMIC directive is semantically equivalent to enclosing the following single statement in a CRITICAL...END CRITICAL directive pair. The statement must have one of the following forms:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

where x is a scalar variable of intrinsic type, expr is a scalar expression that does not reference x, intrinsic is one of MAX, MIN, IAND, IOR, or IEOR, and operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV..

FLUSH

The OpenMP FLUSH directive uses the following syntax:

```
!$OMP FLUSH [(list)]
```

The FLUSH directive ensures that all processor-visible data items, or only those specified in list when it's present, are written back to memory at the point at which the directive appears.

THREADPRIVATE

The OpenMP THREADPRIVATE directive uses the following syntax:

```
!$OMP THREADPRIVATE ( [ /common_block1/ [, /common_block2/] …])
```

where common_blockn is the name of a common block to be made private to each thread but global within the thread. This directive must appear in the declarations section of a program unit after the declaration of any common blocks listed. On entry to a parallel region, data in a THREADPRIVATE common block is undefined unless COPYIN is specified on the PARALLEL directive. When a common block that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use.

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- Only named common blocks can be made thread private.
- It is illegal for a THREADPRIVATE common block or its constituent variables to appear in any clause other than a COPYIN clause.

Run-time Library Routines

User-callable functions are available to the Fortran programmer to query and alter the parallel execution environment.

```
integer omp_get_num_threads()
```

returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable OMP_NUM_THREADS or to the value set by the last previous call to the omp_set_num_threads() subroutine defined in the following section.

```
subroutine omp_set_num_threads(scalar_integer_exp)
```

sets the number of threads to use for the next parallel region. This subroutine can only be called from a serial region of code. If it is called from within a parallel region, or within a subroutine or function that is called from within a parallel region, the results are undefined. This subroutine has precedence over the OMP_NUM_THREADS environment variable.

```
integer omp_get_thread_num()
```

returns the thread number within the team. The thread number lies between 0 and omp_get_num_threads()-1. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region.

```
integer function omp_get_max_threads()
```

returns the maximum value that can be returned by calls to omp_get_num_threads(). If omp_set_num_threads() is used to change the number of processors, subsequent calls to omp_get_max_threads() will return the new value. This function returns the maximum value whether executing from a parallel or serial region of code.

```
integer function omp_get_num_procs()
```

returns the number of processors that are available to the program.

```
logical function omp_in_parallel()
```

returns .TRUE. if called from within a parallel region and .FALSE. if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating .FALSE., the function will return .FALSE..

```
subroutine omp_set_dynamic(scalar_logical_exp)
```

is designed to allow automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect.

```
logical function omp_get_dynamic()
```

is designed to allow the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns .FALSE..

```
subroutine omp_set_nested(scalar_logical_exp)
```

is designed to allow enabling/disabling of nested parallel regions. This function is recognized, but currently has no effect.

```
logical function omp_get_nested()
```

is designed to allow the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns .FALSE..

```
subroutine omp_init_lock(integer_var)
```

initializes a lock associated with the variable integer_var for use in subsequent calls to lock routines. This initial state of integer_var is unlocked. It is illegal to make a call to this routine if integer_var is already associated with a lock.

```
subroutine omp_destroy_lock(integer_var)
```

disassociates a lock associated with the variable integer_var.

```
subroutine omp_set_lock(integer_var)
```

causes the calling thread to wait until the specified lock is available. The thread gains ownership of the lock when it is available. It is illegal to make a call to this routine if integer_var has not been associated with a lock.

```
subroutine omp_unset_lock(integer_var)
```

causes the calling thread to release ownership of the lock associated with integer_var. It is illegal to make a call to this routine if integer_var has not been associated with a lock.

```
logical function omp_test_lock(integer_var)
```

causes the calling thread to try to gain ownership of the lock associated with integer_var. The function returns .TRUE. if the thread gains ownership of the lock, and .FALSE. otherwise. It is illegal to make a call to this routine if integer_var has not been associated with a lock.

Environment Variables

OMP_NUM_THREADS specifies the number of threads to use during execution of parallel regions. The default value for this variable is 1. For historical reasons, the environment variable NCPUS is supported with the same functionality. In the event that both OMP_NUM_THREADS and NCPUS are defined, the value of OMP_NUM_THREADS takes precedence.

NOTE

OMP_NUM_THREADS threads will be used to execute the program regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they will execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient.

OMP_SCHEDULE specifies the type of iteration scheduling to use for DO and PARALLEL DO loops which include the SCHEDULE(RUNTIME) clause. The default value for this variable is "STATIC". If the optional chunk size is not set, a chunk size of 1 is assumed except in the case of a STATIC schedule. For a STATIC schedule, the default is as defined in the DO...END DO and PARALLEL...END PARALLEL sections. Examples of the use of OMP_SCHEDULE are as follows:

```
$ setenv OMP_SCHEDULE "STATIC, 5"
$ setenv OMP_SCHEDULE "GUIDED, 8"
$ setenv OMP_SCHEDULE "DYNAMIC"
```

OMP_DYNAMIC currently has no effect.

OMP_NESTED currently has no effect.

MPSTKZ increases the size of the stacks used by threads executing in parallel regions. It is for use with programs that utilize large amounts of thread-local storage in the form of private variables or local variables in functions or subroutines called within parallel regions. The value should be an integer n concatenated with M or m to specify stack sizes of n megabytes. For example:

```
$ setenv MPSTKZ 8M
```

9 HPF Directives

HPF directives are Fortran 90/95 comments which convey information to the PGHPF compiler. Directives are the heart of an HPF program, indicating data parallelism by specifying how data is assigned and allocated among processors on a parallel system, and the interrelationships between various data elements.

Adding HPF Directives to Programs

Directives in an HPF program may have any of the following forms:

```
CHPF$directive
!HPF$directive
*HPF$directive
```

Since HPF supports two source forms, fixed source form, and free source form, there are a variety of methods to enter a directive. Section 3.4 of the Fortran 95 Handbook outlines methods for entering code that is valid for both free and fixed form Fortran. The C, !, or * must be in column 1 for fixed source form directives. In free source form, Fortran limits the comment character to !. If you use the !HPF$ form for the directive origin, and follow the rules outlined in the Fortran 95 Handbook, your code will be universally valid. The body of the directive may immediately follow the directive origin. Alternatively, using free source form, any number of blanks may precede the HPF directive. Any names in the body of the directive, including the directive name, may not contain embedded blanks. Blanks may surround any special characters, such as a comma or an equals sign.

The directive name, including the directive origin, may contain upper or lower case letters (case is not significant).

HPF Directive Summary

Table 9-1: HPF Directive Summary

DIRECTIVE	FUNCTION
ALIGN	Specifies that a data object is mapped in the same fashion as an associated data object. This is a specification statement. By default, objects are aligned to themselves.
DIMENSION	Specifies the dimensions of a template or processor "array". This is a specification statement.
DISTRIBUTE	Specifies the mapping of data objects to processors. This is a specification statement. By default, objects are not distributed.
DYNAMIC	Specifies that an object may be dynamically realigned or redistributed.
INDEPENDENT	Preceding a DO loop or FORALL , this directive specifies that the DO loop's iterations do not interact in any way and that the FORALL index computations do not interfere with each other, and thus the FORALL may be executed in parallel. This is an executable statement. By default, FORALL and DO loops are not assumed to be independent.
INHERIT	Specifies that a subprogram's dummy argument use the template associated with the actual argument for its alignment. This is a specification statement.
NOSEQUENCE	Specifies variables that are not sequential. Note that using PGHPF, by default variables is not sequential. Variables will be sequential if the compiler option -Msequence is supplied.
PROCESSORS	Specifies the number and rank of a processor arrangement. This is a specification statement.
REALIGN	This is similar to ALIGN, but is executable. An array can be realigned at any time, if it is declared using the DYNAMIC attribute.
REDISTRIBUTE	This is similar to DISTRIBUTE, but is executable. An array can be redistributed at any time, if it is declared using the DYNAMIC attribute.

DIRECTIVE	FUNCTION
SEQUENCE	Specifies that a variable or common block is sequential and requires linear, standard FORTRAN 77, treatment. This is a specification statement.
TEMPLATE	Defines an entity that may be used as an abstract align-target for a distribution or a redistribution. This is a specification statement.

ALIGN - REALIGN

The ALIGN directive specifies how data objects are mapped in relation to other data objects. The data objects that are most often aligned in HPF programs are arrays. Alignment suggests to the compiler that entire objects or elements of arrays be stored on the same processor. Operations on objects that are aligned should be more efficient than operations on objects that are not aligned, assuming that objects that are not aligned may reside on different processors.

REALIGN is similar to ALIGN, but is executable. An array can be realigned at any time, if it is declared using the DYNAMIC attribute.

Syntax

```
!HPF$ ALIGN alignee align-directive-stuff
```

or

```
!HPF$ ALIGN align-attribute-stuff :: alignee-list
```

where:

alignee	is an object-name.
align-directive-stuff	is (align-source-list) align -with-clause
align-attribute-stuff	is [(align-source-list)] align -with-clause

Each align-source has the form:

```
:
*
align-dummy
```

Each align-with-clause has the form:

```
WITH align-target [ ( align-subscript-list) ]
```

An align-subscript has the form:

```
int-exp
align-subscript-use
subscript-triplet
*
```

Type

Specification

Default

The default PGHPF alignment specifies that a data object is replicated across all processor memories. For example, for an array RAY1 with a single dimension and a template T with matching size and shape, the following alignment specifies replication when T is distributed in any manner across processors.

```
!HPF$ALIGN RAY1(*) WITH T(*)
!HPF$DISTRIBUTE T(BLOCK)
```

See Also

For details on the ALIGN syntax specifications, refer either to section 4.5 of The High Performance Fortran Handbook, or section 3.4 of the HPF Language Specification.

Example

```
 PROGRAM TEST
 INTEGER A(1000)
!HPF$ PROCESSORS PROC(10)
!HPF$TEMPLATE T(1000)
!HPF$ ALIGN A(:) WITH T(:)
!HPF$DISTRIBUTE (BLOCK) ONTO PROC:: T
```

DIMENSION

The DIMENSION attribute specifies the dimensions and extents for each dimension of a TEMPLATE or PROCESSORS directive.

Syntax

```
!HPF$ DIMENSION ( explicit-shape-spec-list )
```

Type

Specification

Default

The default for a TEMPLATE or PROCESSORS arrangement is a scalar.

See Also

The TEMPLATE and PROCESSORS directives.

Example

```
 REAL A(100,100)
!HPF$PROCESSORS, DIMENSION(10,10):: PROC
!HPF$TEMPLATE, DIMENSION(10,10):: T
!HPF$ALIGN WITH T:: A
!HPF$DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

DYNAMIC

The DYNAMIC attribute specifies that an object may be dynamically realigned or redistributed.

Syntax

```
!HPF$ DYNAMIC alignee-or-distributeee-list
```

Type

Specification

Default

By default an object is not dynamic.

See Also

The REALIGN and REDISTRIBUTE directives.

Example

```
 REAL A(100,100)
!HPF$ DYNAMIC A
!HPF$PROCESSORS, DIMENSION(10,10):: PROC
!HPF$TEMPLATE, DIMENSION(10,10):: T
!HPF$ALIGN WITH T:: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

DISTRIBUTE - REDISTRIBUTE

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. Distribution partitions an object, in the usual case an array (actually a template), among a set of processors.

REDISTRIBUTE is similar to DISTRIBUTE, but is executable. An array can be redistributed at any time, if it is declared using the DYNAMIC attribute

Syntax

```
!HPF$ DISTRIBUTE distributee dist-directive-stuff
```

or

```
!HPF$ DISTRIBUTE dist-attribute-stuff :: distributee-list
```

where dist-directive-stuff is one of:

```
(dist-format-list)
(dist-format-list) ONTO processors-name
```

The form of dist-attribute-stuff is one of:

```
(dist-format-list)
(dist-format-list) ONTO processors-name
ONTO dist-target
```

The dist-format may be one of:

```
BLOCK [ (int-expr) ]
CYCLIC [ (int-expr)]
```

Type

Specification

Default

By default, each object is replicated and distributed to every processor.

See Also

For details on the DISTRIBUTE syntax specifications, refer either to Section 4.4 of The High Performance Fortran Handbook, or Section 3.3 of the HPF Language Specification.

Example

```
 REAL A(100,100)
!HPF$PROCESSORS PROC(10,10)
!HPF$TEMPLATE T(10,10)
!HPF$ALIGN WITH T:: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PROC:: T
```

INDEPENDENT

The INDEPENDENT directive specifies that the iterations of a DO loop, or the computations for the active index values of a FORALL, do not interfere with each other in any way. Refer to the PGHPF Release notes for details on extensions to the INDEPENDENT directive.

Syntax

```
!HPF$ INDEPENDENT [, NEW ( variable-list
) ]
```

Type

Executable

Default

By default, DO and FORALL statements are not independent.

See Also

For details on the INDEPENDENT syntax specifications, refer either to Section 6.4 of The High Performance Fortran Handbook, or Section 4.4 of the HPF Language Specification. Also refer to the PGHPF Release notes for details on extensions to the INDEPENDENT directive.

Example

```
!HPF$INDEPENDENT
 DO I = 2, N-1
 X(I) = Y(I-1) + Y(I) + Y(I+1)
 END DO
```

INHERIT

The INHERIT directive specifies that the template for a dummy argument should be the same as the template for the corresponding actual argument.

Syntax

```
!HPF$ INHERIT dummy-argument-name-list
```

Default

If the INHERIT attribute is not used, and ALIGN and DISTRIBUTE are not used for a dummy argument, then the dummy's template has the same shape as the dummy argument and it is ultimately aligned with itself.

Type

Specification

See Also

For details on the INHERIT syntax specifications, refer either to Section 5.4 of The High Performance Fortran Handbook, or Section 3.9 of the HPF Language Specification.

Example

```
  REAL VAR1(100)
!HPF$ DISTRIBUTE VAR1(BLOCK)10))
 CALL SUB1( VAR1(10:20:2))
 SUBROUTINE SUB1(PARAM1)
 REAL PARAM1(5)
!HPF$INHERIT PARAM1
```

PROCESSORS

The PROCESSORS directive specifies one or more processor arrangements, by name, rank, and size.

Syntax

```
!HPF$PROCESSORS processors-decl-list
```

Default

The default for PROCESSORS is the number of processors on which the program is running, as specified by the runtime command-line options.

Type

Specification

See Also

For details on the PROCESSOR syntax specifications, refer either to Section 4.8 of The High Performance Fortran Handbook, or Section 3.7 of the HPF Language Specification

For finding more information on processors while running a program, refer to the NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE intrinsics.

Examples

```
!HPF$PROCESSORS PROCN(128)
!HPF$ PROCESSORS PROC2(3,3,3)
!HPF$PROCESSORS:: PROC3(-8:12,100:200)
```

NO SEQUENCE

In environments where variables are by default sequential, the NO SEQUENCE directive specifies that non-sequential access should apply to a scoping unit or to variables and common blocks within the scoping unit.

Syntax

```
!HPF$ NO SEQUENCE
```

or

```
!HPF$ NOSEQUENCE [::] association-name-list
```

Type

Specification

See Also

For details on the NO SEQUENCE syntax specifications, refer either to Section 4.10.2 of The High Performance Fortran Handbook, or Section 7.1.3 of the HPF Language Specification

The SEQUENCE directive.

Example

```
   INTEGER FLAG, I, A(1000)
   COMMON /FOO/ A,I,FLAG
 !HPF$NOSEQUENCE FOO
```

SEQUENCE

The SEQUENCE directive allows a user to declare explicitly that variables or common blocks are to be treated by the compiler as sequential.

Syntax

```
!HPF$ SEQUENCE
```

or

```
!HPF$ SEQUENCE [::] association-name-list
```

Type

Specification

See Also

For details on the SEQUENCE syntax specifications, refer either to Section 4.10.2 of The High Performance Fortran Handbook, or Section 7.1.3 of the HPF Language Specification.

The NO SEQUENCE directive.

Example

```
   INTEGER FLAG, I, A(1000)
   COMMON /FOO/ A,I,FLAG
 !HPF$ SEQUENCE FOO
```

TEMPLATE

The TEMPLATE directive declares one or more templates, specifying for each a name, rank, and size for each dimension.

Syntax

```
!HPF$ TEMPLATE template-decl-list
```

Default

By default for each object, a new template is created and in the absence of an explicit ALIGN directive, the object is ultimately aligned to itself.

Type

Specification

See Also

For details on the TEMPLATE syntax specifications, refer either to Section 4.9 of The High Performance Fortran Handbook, or Section 3.8 of the HPF Language Specification.

Examples

```
!HPF$TEMPLATE VAR1(N)
!HPF$TEMPLATE VAR2(N,N)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,BLOCK):: BOARD(8,8)
```

Appendix A. HPF_LOCAL

This appendix lists the HPF_LOCAL_LIBRARY procedures. For complete descriptions of the HPF_LOCAL_LIBRARY routines, and the current standards for HPF_LOCAL extrinsics, refer to Annex A, "Coding Local Routines in HPF and Fortran 90", in the High Performance Fortran Language Specification. Table A-1 , “HPF_LOCAL_LIBRARY Procedures”, briefly lists the procedures. Refer to the man pages supplied with the PGHPF software for further details on these procedures. Refer to Chapter 6, “Fortran Intrinsics”, for details on the intrinsics defined in the Fortran 90/95 Language Specification and for HPF LIBRARY procedures.

For complete descriptions of the HPF_LOCAL_LIBRARY routines, and the current standards for HPF_LOCAL extrinsics, refer to Annex A, "Coding Local Routines in HPF and Fortran 90", in the High Performance Fortran Language Specification.

Table A-1: HPF_LOCAL_LIBRARY Procedures

Intrinsic	Description
ABSTRACT_TO_PHYSICAL	Returns processor identification for the physical processor associated with a specified abstract processor.
GLOBAL_ALIGNMENT	Returns information about the global HPF array argument.
GLOBAL_DISTRIBUTION	Returns information about the global HPF array argument.
GLOBAL_LBOUND	Returns lower bounds of the actual HPF global array associated with a dummy array.
GLOBAL_SHAPE	Returns the shape of the global HPF actual argument.
GLOBAL_SIZE	Returns the global extent of the specified argument.
GLOBAL_TEMPLATE	Returns template information for the global HPF array argument.
GLOBAL_TO_LOCAL	Converts a set of global coordinates within a global HPF actual argument.
GLOBAL_UBOUND	Returns upper bounds of the actual HPF global array associated with a dummy array.
LOCAL_BLKCNT	Returns the number of blocks of elements in each dimension on a given processor.
LOCAL_LINDEX	Returns the lowest local index of all blocks of an array dummy.
LOCAL_TO_GLOBAL	Converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates.

Intrinsic	Description
LOCAL_UINDEX	Returns the highest local index of all blocks of an array dummy argument.
MY_PROCESSOR	Returns the identifying number of the calling physical processor.
PHYSICAL_TO_ABSTRACT	Returns coordinates for an abstract processor, relative to a global actual argument array.

ABSTRACT_TO_PHYSICAL

Subroutine returns processor identification for the physical processor associated with a specified abstract processor relative to a global actual argument array.

Synopsis

```
ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)
```

Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

INDEX must be a rank-1 integer array containing the coordinates of an abstract processor in the processors arrangement onto which the global HPF array is mapped. It is an INTENT(IN) argument. The size of INDEX must equal the rank of the processors arrangement.

PROC must be scalar and of type integer. It is an INTENT(OUT) argument. It receives the identifying value for the physical processor associated with the abstract processor specified by INDEX.

GLOBAL_ALIGNMENT

This has the same interface and behavior as the HPF inquiry subroutine HPF_ALIGNMENT, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

Synopsis

```
GLOBAL_ALIGNMENT(ARRAY, ...)
```

GLOBAL_DISTRIBUTION

This has the same interface and behavior as the HPF inquiry subroutine HPF_DISTRIBUTION, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

Synopsis

```
GLOBAL_DISTRIBUTION(ARRAY, ...)
```

GLOBAL_LBOUND

Inquiry function, returns all the lower bounds or a specified lower bound of the actual HPF global array.

Synopsis

```
GLOBAL_LBOUND(ARRAY, DIM)
```

Arguments

Optional argument. DIM

ARRAY may be of any type. It must not be a scalar. It must be a dummy array argument of an HPF_LOCAL procedure which is argument associated with a global HPF array actual argument.

DIM (optional) must be scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Return Type

The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size n, where n is the rank of ARRAY.

Return Value

If the actual argument associated with the actual argument associated with ARRAY is an array section or an array expression, other than a whole array or an array structure component, GLOBAL_LBOUND(ARRAY, DIM) has the value 1; otherwise it has a value equal to the lower bound for subscript DIM of the actual argument associated with the actual argument associated with ARRAY.

GLOBAL_LBOUND(ARRAY) has a value whose i th component is equal to GLOBAL_LBOUND(ARRAY, i), for i = 1,2,...n where n is the rank of ARRAY.

GLOBAL_SHAPE

Returns the shape of the global HPF actual argument associated with an array or scalar dummy argument of an HPF_LOCAL procedure.

Synopsis

```
GLOBAL_SHAPE(SOURCE)
```

Argument

SOURCE may be of any type. It may be array valued or a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.

Return Type

The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

Return Value

The value of the result is the shape of the global actual argument associated with the actual argument associated with SOURCE.

GLOBAL_SIZE

Inquiry function returns the extent along a specified dimension of the global HPF actual array argument associated with a dummy array argument of an HPF_LOCAL procedure.

Synopsis

```
GLOBAL_SIZE(ARRAY, DIM)
```

Arguments

ARRAY may be of any type. It must not be a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.

DIM (optional) must be scalar and of type integer with a value in the range 1<= DIM <= n, where n is the rank of ARRAY.

Return Type

Default integer scalar.

Return Value

The result has a value equal to the extent of dimension DIM of the actual argument associated with the actual argument associated with ARRAY or, if DIM is absent, the total number of elements in the actual argument associated with the actual argument associated with ARRAY.

GLOBAL_TEMPLATE

This has the same interface and behavior as the HPF inquiry subroutine HPF_TEMPLATE, but it returns information about the global HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

Synopsis

```
GLOBAL_TEMPLATE(ARRAY, ...)
```

Arguments

Refer to HPF_TEMPLATE.

GLOBAL_TO_LOCAL

Subroutine converts a set of global coordinates within a global HPF actual argument array to an equivalent set of local coordinates within the associated local dummy array.

Synopsis

```
GLOBAL_TO_LOCAL(ARRAY, G_INDEX,
L_INDEX, LOCAL,
 NCOPIES, PROCS)
```

Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

G_INDEX must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(IN) argument. It contains the coordinates of an element within the global HPF array actual argument associated with the local dummy array ARRAY.

L_INDEX (optional) must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(OUT) argument. It receives the coordinates within a local dummy array of the element identified within the global actual argument array by G_INDEX. (These coordinates are identical on any processor that holds a copy of the identified global array element.)

LOCAL (optional) must be scalar and of type LOGICAL. It is an INTENT(OUT) argument. It is set to .TRUE. if the local array contains a copy of the global array element and to .FALSE. otherwise.

NCOPIES (optional) must be scalar and of type integer. It is an INTENT(OUT) argument. It is set to the number of processors that hold a copy of the identified element of the global actual array.

PROCS (optional) must be a rank-1 integer array whose size is a least the number of processors that hold copies of the identified element of the global actual array. The identifying numbers of those processors are placed in PROCS. The order in which they appear is implementation dependent.

GLOBAL_UBOUND

Inquiry function returns all the upper bounds or a specified upper bound of the actual HPF global array argument associated with an HPF_LOCAL dummy array argument.

Synopsis

```
GLOBAL_UBOUND(ARRAY, DIM)
```

Arguments

Optional argument. DIM

ARRAY may be of any type. It must not be a scalar. It must be a dummy array argument of an HPF_LOCAL procedure which is argument associated with a global HPF array actual argument.

DIM (optional) must be scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Return Type

The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size n, where n is the rank of ARRAY.

Return Value

If the actual argument associated with the actual argument associated with ARRAY is an array section or an array expression, other than a whole array or an array structure component, GLOBAL_UBOUND(ARRAY, DIM) has a value equal to the number of elements in the given dimension; otherwise it has a value equal to the upper bound for subscript DIM of the actual argument associated with the actual argument associated with ARRAY, if dimension DIM does not have size zero and has the value zero if dimension DIM has size zero.

GLOBAL_UBOUND(ARRAY) has a value whose i th component is equal to GLOBAL_UBOUND(ARRAY, i), for i = 1,2,...n where n is the rank of ARRAY.

LOCAL_BLKCNT

Pure function returns the number of blocks of elements in each dimension, or of a specific dimension of the array on a given processor.

Synopsis

```
LOCAL_BLKCNT(ARRAY, DIM, PROC)
```

Arguments

Optional arguments. DIM, PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM (optional) must be scalar and of type integer with a value in the range 1<= DIM <= n where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

Return Type

The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size n, where n is the rank of ARRAY.

Return Value

The value of LOCAL_BLKCNT(ARRAY, DIM, PROC) is the number of blocks of the ultimate align target of ARRAY in dimension DIM that are mapped to processor PROC and which have at least one element of ARRAY aligned to them.

LOCAL_BLKCNT(ARRAY, DIM) returns the same value as LOCAL_BLKCNT(ARRAY, DIM, PROC=MY_PROCESSOR()).

LOCAL_BLKCNT(ARRAY) has a value whose i th component is equal to LOCAL_BLKCNT(ARRAY, i), for i = 1,...,n, where n is the rank of ARRAY.

LOCAL_LINDEX

Pure function returns the lowest local index of all blocks of an array dummy argument in a given dimension on a processor.

Synopsis

```
LOCAL_LINDEX(ARRAY, DIM, PROC)
```

Arguments

Optional argument. PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM must be scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

Return Type

The result is a rank-one array of type default integer and size 1<= DIM <= n, where n is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC]).

Return Value

The value of LOCAL_LINDEX(ARRAY, DIM, PROC) has a value whose i th component is the local index of the first element of the i th block in dimension DIM of ARRAY on processor PROC.

LOCAL_LINDEX(ARRAY, DIM) returns the same value as LOCAL_LINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

LOCAL_TO_GLOBAL

Subroutine converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates within the associated global HPF actual argument array.

Synopsis

```
LOCAL_TO_GLOBAL(ARRAY, L_INDEX,
G_INDEX)
```

Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

L_INDEX must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(IN) argument. It contains the coordinates of an element within the local dummy array ARRAY.

G_INDEX must be a rank-1 integer array whose size is equal to the rank of ARRAY. It is an INTENT(OUT) argument. It receives the coordinates within the global HPF array actual argument of the element identified within the local array by L_INDEX.

LOCAL_UINDEX

Pure function returns the highest local index of all blocks of an array dummy argument in a given dimension on a processor.

Synopsis

```
LOCAL_UINDEX(ARRAY, DIM, PROC)
```

Arguments

Optional argument. PROC.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM must be scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

Return Type

The result is a rank-one array of type default integer and size b , where b is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC])

Return Value

The value of LOCAL_UINDEX(ARRAY, DIM, PROC) has a value whose i th component is the local index of the last element of the i th block in dimension DIM of ARRAY on processor PROC.

LOCAL_UINDEX(ARRAY, DIM) returns the same value as LOCAL_UINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

MY_PROCESSOR

Pure function returns the identifying number of the calling physical processor.

Synopsis

```
MY_PROCESSOR()
```

Return Type

The result is scalar and of type default integer.

Return Value

Returns the identifying number of the physical processor from which the call is made. This value is in the range where is the value returned by NUMBER_OF_PROCESSORS().

PHYSICAL_TO_ABSTRACT

Subroutine returns coordinates for an abstract processor, relative to a global actual argument array, corresponding to a specified physical processor. This procedure can be used only on systems where there is a one-to-one correspondence between abstract processors and physical processors. On systems where this correspondence is one-to-many an equivalent, system-dependent procedure should be provided.

Synopsis

```
PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)
```

Arguments

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.

PROC must be scalar and of type default integer. It is an INTENT(IN) argument. It contains an identifying value for a physical processor.

INDEX must be a rank-1 integer array. It is an INTENT(OUT) argument. The size of INDEX must equal the rank of the processor arrangement onto which the global HPF array is mapped. INDEX receives the coordinates within this processors arrangement of the abstract processor associated with the physical processor specified by PROC.

Index