Fortran 90 Programmer's Reference

First Edition

Product Number: B3909DB
Fortran 90 Compiler for HP-UX
Document Number: B3908-90002
October 1998

Edition: First

Document Number: B3908-90002
Remarks: Released October 1998. Initial release.

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Contents

Tables

Tables

Tables

Tables

Preface

The *HP Fortran 90 Programmer's Reference* is a language reference for programmers using HP Fortran V2.0 and higher. It describes the features and requirements of the HP Fortran 90 programming language.

The *HP Fortran 90 Programmer's Reference* is intended for use by experienced Fortran 90 programmers who are interested in writing or porting HP Fortran 90 applications. This manual includes information on the parallel concepts and directives, as well as optimization of programs that use them.

You need not be familiar with the HP parallel architecture, programming models, or optimization concepts to understand the concepts introduced in this book.

New in HP Fortran 90 V2.0

The HP Fortran 90 V2.0 features described in this reference are upgrades from the previous version of HP Fortran, V1.0. These include the following:

- Compatibility directives that control vectorizing and parallelizing optimizations.
- The `BUFFER IN` and `BUFFER OUT` statements for compatibility with the Cray implementation of Fortran.
- The `OPTIONS` statement for compatibility with various versions of Fortran.
- Support for parallel execution, including the `+Oparallel` compile-line option.
- Support for 64-bit code generation, including the `+DA2.0W` compile-line option.

Scope

This guide covers programming methods for the HP Fortran 90 compiler on the V-Class V2200 and V2250, and K-Class machines running HP-UX 11.0 and higher.

HP Fortran 90 supports an extensive shared-memory programming model. HP-UX 11.0 and higher includes the required assembler, linker, and libraries.

Notational conventions

This section discusses notational conventions used in this book.

`bold monospace`	In command examples, **`bold monospace`** identifies input that must be typed exactly as shown.
`monospace`	In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages.
italic	In paragraph text, *italic* identifies titles of documents. In command syntax diagrams, *italic* identifies variables that you must provide. The following command example uses brackets to indicate that the variable *output_file* is optional: `command` *input_file* [*output_file*]
Brackets ([ ])	In command examples, square brackets designate optional entries.
Curly brackets ({}), Pipe (|)	In command syntax diagrams, text surrounded by curly brackets indicates a choice. The choices available are shown inside the curly brackets and separated by the pipe sign (|). The following command example indicates that you can enter either `a` or `b`: `command {a	b}`

Horizontal ellipses (...)	In command examples, horizontal ellipses show repetition of the preceding items.
Vertical ellipses	Vertical ellipses show that lines of code have been left out of an example.
Keycap	**Keycap** indicates the keyboard keys you must press to execute the command example.

The term “Fortran” refers to Fortran 90.

References to man pages appear in the form mnpgname(1), where “mnpgname” is the name of the man page and is followed by its section number enclosed in parentheses. To view this man page, type:

`%` **`man 1 mnpgname`**

NOTE A Note highlights important supplemental information.

Command syntax

Consider this example:

`COMMAND` *input_file* `[...] {a | b} [`*output_file*`]`

- `COMMAND` must be typed as it appears.
- *input_file* indicates a file name that must be supplied by the user.
- The horizontal ellipsis in brackets indicates that additional, optional input file names may be supplied.
- Either `a` or `b` must be supplied.
- [*output_file*] indicates an optional file name.

Associated documents

The following documents are listed as additional resources to help you use the compilers and associated tools:

- *HP aC++ Online Programmer's Guide*—Presents reference and tutorial information on aC++. This manual is only available in html format.
- *HP C/HP-UX Programmer's Guide*—Contains detailed discussions of selected C topics.
- *HP C/HP-UX Reference Manual*—Presents reference information on the C programming language, as implemented by HP.
- *CXperf Command Reference*—Provides both introductory and reference information for using the CXPerf performance analyzer.
- *CXperf User's Guide*—Provides information on how to use the CXperf performance analysis tool.
- *HP-UX Floating Point Guide*—Describes how floating-point arithmetic is implemented on HP 9000 Series 700/800 systems. It discusses how floating-point behavior affects the programmer. Additional useful includes that which assists the programmer in writing or porting floating-point intensive programs.
- *Fortran 90 Programmer's Guide*—Provides extensive usage information (including how to compile and link), suggestions and tools for migrating to HP Fortran 90, and how to call C and HP-UX routines for HP Fortran 90.
- *HP MPI User's Guide*—Discusses message-passing programming using HP's Message-Passing Interface library.
- *HP-UX Linker and Libraries User's Guide*—Describes how to develop software on HP-UX, using the HP compilers, assemblers, linker, libraries, and object files.
- *Parallel Programming Guide for HP-UX Systems*—Describes efficient methods for shared-memory programming using the HP-UX suite of compilers: HP Fortran 90. HP aC++ (ANSI C++), and HP C. This guide is intended for use by experienced Fortran 90, C, and C++ programmers and is intended for use on HP-UX 11.0 and higher.

- *Programming with Threads on HP-UX*—Discusses programming with POSIX threads.
- *Threadtime* by Scott J. Norton and Mark D. DiPasquale—Provides detailed guidelines on the basics of thread management, including POSIX thread structure; thread management functions; and the creation, termination and synchronization of threads.

NOTE Many of these documents are accessible through the HP document World Wide Web site at http://docs.hp.com. To locate a particular document at this location, use this site's search link to search for the document name or subject matter.

1 Introduction to HP Fortran 90

This chapter summarizes standard features of HP Fortran 90 that are not found in FORTRAN 77. This includes the following topics:

- Source format
- Data types
- Pointers
- Arrays
- Control constructs
- Operators
- Procedures
- Modules
- I/O features
- Intrinsics

HP Fortran 90 features

The following summarizes features of Fortran 90 that are not in standard FORTRAN 77 and indicates where they are described in this manual.

Source format

The fixed source form of FORTRAN 77 is extended by the addition of the semicolon (;) statement separator and the exclamation point (!) trailing comment.

HP Fortran 90 also supports free format source code. The format used in a source program file is normally indicated by the file suffix, but the default format can be overridden by the `+source` compile-line option.

For information about source format, see “Source format of program file” on page 13.

Data types

- Data declarations can include a kind type parameter—an integer value that determines the range and precision of values for the declared data object. The kind type parameter value is the number of bytes representing an entity of that type, except for `COMPLEX` entities, where the number of bytes required is double the kind type value.

  In principle, multibyte character data for languages with large character sets can be implemented in Fortran 90 by means of a kind type parameter for the `CHARACTER` data type. HP Fortran 90, however, uses the Extended Unix Code (EUC) characters in file names, comments, and string literals.

- Fortran 90 supports derived types, which are composed of entities of the intrinsic types (`INTEGER`, `REAL`, `COMPLEX`, `LOGICAL`, and `CHARACTER`) or entities of previously defined derived types. You declare derived-type objects in the same way that you declare intrinsic-type objects.

For information about intrinsic and derived types, see “Intrinsic data types” on page 22 and “Derived types” on page 39.

Pointers

Pointers are variables that contain addresses of other variables of the same type. Pointers are declared in Fortran 90 with the POINTER attribute. A pointer is an alias, and the variable (or allocated space) for which it is an alias is its target. The pointer enables data to be accessed and handled dynamically. For more information, see "Pointers" on page 47.

Arrays

The Fortran 90 standard has defined these new array features:

- Array sections that permit operations for processing whole arrays or a subset of array elements; expressions, functions, and assignments can be array-valued. The `WHERE` construct and statement are used for masked-array assignment.
- Array constructors—unnamed, rank-one arrays whose elements can be constants or variables. You can use the `RESHAPE` intrinsic function to transform the array constructor to an array value of higher rank.
- New types of array:
  - Assumed-shape arrays are dummy arguments that take on the size and shape of the corresponding actual arguments.
  - Deferred-shape arrays become defined when they are associated with target array objects.
  - Automatic arrays have at least one bound that is not a constant.

Arrays are discussed in Chapter 4, "Arrays," on page 51.

Control constructs

Control constructs

- The `CASE` construct selects and executes one or more associated statements on the basis of a case selector value, which can be of type `INTEGER`, `CHARACTER` or `LOGICAL`.
- Additional forms of the `DO` statement allow branching to the end of a `DO` loop and branching out of a `DO` loop.

These constructs are described in “Control constructs and statement blocks” on page 104.

Operators

You can write your own procedures to define new operations for intrinsic operators, including assignment, for use with operands of intrinsic data types or derived data types; see “Defined operators” on page 155 and “Defined assignment” on page 157.

Procedures

- Fortran 90 includes a feature called the *procedure interface block*, which provides an explicit interface for external procedures. The names and properties of the dummy arguments are then available to the calling procedure, allowing the compiler to check that the dummy and actual arguments match. For information about interface blocks, see “Procedure interface” on page 151.
- Actual arguments can be omitted from the argument list or can be arranged in a different order from the dummy arguments.
- You can implement user-defined operators or extend intrinsic operators, including the assignment operator; see “Defined operators” on page 155 and “Defined assignment” on page 157.
- Dummy arguments to procedures can be given an `INTENT` attribute (`IN`, `OUT` or `INOUT`); see “INTENT attribute” on page 148.
- Subprograms can appear within a module subprogram, an external subprogram, or a main program unit; see “Internal procedures” on page 135.
- Recursive procedures (an extension in HP FORTRAN 77) are a standard feature of Fortran 90. For more information, see “Recursive reference” on page 131.

Modules

A module is a program unit that can be used to specify data objects, named constants, derived types, procedures, operators, and namelist groups. Partial or complete access to these module entities is provided by the `USE` statement. An entity may be declared `PRIVATE` to limit visibility to the module itself.

One use of the module is to provide controlled access to global data, making it a safer alternative to the COMMON block. The module also provides a convenient way to encapsulate the specification of derived types with their associated operations.

For information about modules, see “Modules” on page 161.

I/O features

- Nonadvancing I/O

  After a record-based I/O operation in FORTRAN 77, the file pointer moves to the start of the next record. In Fortran 90, you can use the `ADVANCE=NO` specifier to position the file pointer after the characters just read or written rather than at the start of the next record. Nonadvancing I/O thus allows you to determine the length of a variable-length record. See “Nonadvancing I/O” on page 187 for more information.

- Namelist-directed I/O

  Namelist-directed I/O—previously available as an extension to FORTRAN 77—is a standard feature of Fortran 90. This feature enables you to perform repeated I/O operations on a named group of variables. See “Namelist-directed I/O” on page 183 for more information.

Intrinsics

Fortran 90 provides a large number of new intrinsic procedures for manipulating arrays. Many of them are elemental, taking either scalar or array arguments. In the latter case, the result is as if the procedure were applied separately to each element of the array.

Other additions include transformational functions that operate on arrays as a whole, and inquiry functions that return information about the properties of the arguments rather than values computed from them.

Table 9 on page 78 lists the array-inquiry intrinsic functions. For descriptions of all intrinsic procedures, see Chapter 11, “Intrinsic procedures,” on page 475.

2 Language elements

This chapter describes the basic elements of an HP Fortran 90 program. This includes the following topics:

- Character set
- Lexical tokens
- Program structure
- Statement labels
- Statements
- Source format of program file
- INCLUDE line

Character set

The Fortran 90 standard character set, shown in Table 1, consists of letters, digits, the underscore character, and special characters. The HP Fortran 90 character set consists of the Fortran 90 character set, plus:

- Control characters (Tab, Newline , and Carriage return). Carriage return and Tab are usually treated as “white space” in a source program. You can use them freely to make the source easier to read.
- The pound sign (#) character in column 1 to initiate a comment. This is an HP extension that allows C preprocessor directives embedded in source files to be treated as comments.
- Any other characters in the HP character set listed in Appendix B. These characters may be used in character constants, character string edit descriptors, comments, and I/O records.

Table 1 **Fortran 90 character set**

Category	Characters
Letters	A to Z, a to z
Digits	0 to 9
Underscore	_
Special characters	blank (space) : ! “ % & ; < > ? $ = + - * / ( ) , . '

Lowercase alphabetic characters are equivalent to uppercase characters except when they appear in character strings or Hollerith constants.

HP Fortran 90 supports only the default character type, `CHARACTER(KIND=1)`, as described in “Type declaration for intrinsic types” on page 24. Support is provided, however, for Extended Unix Code (EUC) and Shift-JIS encoding.

Lexical tokens

Lexical tokens consist of sequences of characters and are the building blocks of a program. They denote names, operators, literal constants, labels, keywords, delimiters, and may also include the following characters and character combinations:

```
, = => : :: ; %
```

Names

In Fortran 90, names denote entities such as variables, procedures, derived types, named constants, and COMMON blocks. A name must start with a letter but can consist of any combination of letters, digits, and underscore (_) characters. As an extension in HP Fortran 90, the dollar sign may also be used in a name, but not as the first character.

The Fortran 90 Standard allows a maximum length of 31 characters in a name. In HP Fortran 90 this limit is extended to 255 characters, and all are significant—that is, two names that differ only in their 255th character are treated as distinct. Names and keywords are case insensitive: for example, `Title$23_Name` and `TITLE$23_NAME` are the same name.

The `CASE`, `IF`, and `DO` constructs can optionally be given names. The construct name appears before the first statement of the construct, followed by a colon (`:`). The same name must appear at the end of the final statement of the construct. For more information about these constructs, refer to “Control constructs and statement blocks” on page 104.

Program structure

A complete executable Fortran program contains one and only one main program unit and may also contain one or more of the following other types of program units:

- External function subprogram unit
- External subroutine subprogram unit
- Block data program unit
- Module program unit

Each program unit can be compiled separately. Execution of the program starts in the main program. Control may be passed to other program units.

The Fortran 90 program units, and the transfer of control between them, are described in Chapter 7, “Program units and procedures,” on page 121.

Statement labels

A Fortran 90 statement may have a preceding label, composed of one to five digits. All statement labels in the same scoping unit must be unique; leading zeroes are not significant. Although most statements can be labeled, not all statements can be branched to.

The `FORMAT` statement must have a label. The `INCLUDE` line, which is not a statement but a compiler directive, must not have a label.

Statements

All HP Fortran 90 statements are fully described in Chapter 10, “HP Fortran 90 statements,” on page 241, in alphabetical order.

The required order for statements in a standard Fortran 90 program unit is illustrated in Table 2. Vertical lines separate statements that can be interspersed, and horizontal lines separate statements that cannot be interspersed. For example, the DATA statement can appear among executable statements but may not be interspersed with CONTAIN statements. Also, the USE statement, if present, must immediately follow the program unit heading.

Table 2 **Statement order in a program unit**

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statement</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE statement</td></tr>
<tr><td>PARAMETER statement</td><td>IMPLICIT statement</td></tr>
<tr><td>PARAMETER and
DATA statements</td><td>Derived-type definitions,
Interface blocks,
Type declarations,
Statement functions, and
Specification statements</td></tr>
<tr><td>DATA statements</td><td>Executable constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Table 2 does not show where comments, the `INCLUDE` line, and directives may appear. Comments may appear anywhere in a source file, including after the `END` statement. The `INCLUDE` line may appear anywhere before the `END PROGRAM` statement.

Table 3 identifies which statements may appear within a scoping unit; a check mark indicates that a statement is allowed in the specified scoping unit. For the purpose of this table, *type declarations* include the `PARAMETER` statement, the `IMPLICIT` statement, type declaration statements, derived-type definitions, and specification statements.

Table 3 Statements allowed in scoping units

Statements	Scoping units						
	Main program	External procedure	Module	Module procedure	Internal procedure	Interface body	Block data program unit
`CONTAINS`	✓	✓	✓	✓			
`DATA`	✓	✓	✓	✓	✓		✓
`ENTRY`		✓		✓			
Executable	✓	✓		✓	✓		
`FORMAT`	✓	✓		✓	✓		
Interface block	✓	✓	✓	✓	✓	✓	
Statement function	✓	✓		✓	✓		
Type declaration	✓	✓	✓	✓	✓	✓	✓
`USE`	✓	✓	✓	✓	✓	✓	✓

Source format of program file

The HP Fortran 90 compiler accepts source files in fixed form (the standard source form for FORTRAN 77 programs) or free form. The following sections describe both forms.

The compiler assumes that source files whose names end in the `.f90` extension are in free source form and that files whose names end in the `.f` or `.F` extension are in fixed form. You can override these assumptions by compiling with the `+source=free` or `+source=free` option. See the *HP Fortran 90 Programmer's Guide* for more information.

Although the two forms are quite different, you can format a Fortran 90 source file so that the compiler would accept it as either fixed or free form. This would be necessary, for example, when preparing a source file containing code that will be inserted through the `INCLUDE` line into a file for which the form is not known. To format a source file to be acceptable as either free or fixed source form, use the following rules:

- Put labels in columns1-5.
- Put statement bodies in columns 7-72.
- Begin comments with an exclamation mark in anycolumn except column 6.
- Indicate all continuations with an ampersand character (&) in column 73 of the line to be continued and an ampersand character in column 6 of the continuing line.
- Do not insert blanks in tokens.
- Separate adjacent names and keywords with a space.

Free source form

In free source form, the source line is not divided into fields of predefined width, as in the fixed form. This makes entering text at an interactive terminal more convenient.

Source lines

Freeform lines can contain from 0 to 132 characters. The `+extend_source` option extends the line to column 254. This is described in the *HP Fortran 90 Programmer's Guide*. Several statements can appear on a single source line, separated by semicolons. A single Fortran 90 statement can extend over more than one source line, as described below in “Statement continuation” on page 15.

Multiple statements may appear on the same line, separated by a semicolon (`;`).

Statement labels

Statement labels are not required to be in columns 1-5, but must be separated from the statement by at least one space.

Spaces

Spaces are significant:

- They may not appear within a lexical token, such as a name or an operator.
- In general, one or more spaces are required to separate adjacent statement keywords, names, constants, or labels. Within the keyword pairs listed in Table 4, however, the space is optional. The keyword following `END` can be: `BLOCK DATA`, `DO`, `FILE`, `FUNCTION`, `IF`, `INTERFACE`, `MAP`, `MODULE`, `PROGRAM`, `SELECT`, `SUBROUTINE`, `STRUCTURE`, `TYPE`, `UNION`, or `WHERE`.

Table 4 Keywords allowing optional spacing

`BLOCK DATA`	`GO TO`
`DOUBLE PRECISION`	`IN OUT`
`ELSE IF`	`SELECT CASE`
`END` *keyword*	

- Spaces are not required between a name and an operator because the latter begins and ends with special symbols that cannot be part of a name. Multiple spaces, unless in a character context, are equivalent to a single space.

Consider the spaces (designated by *b*) in the following statement:

```
IFbb(TEXT.EQ.'bbbYES') ...    ! Valid
```

The two spaces after `IF` are valid and are equivalent to one space. No spaces are required before or after `.EQ.`, because there is no ambiguity. However, the three spaces in the character constant are significant.

In the next example

```
IF(MbARY.bGE.MIKE) ...        ! Faulty in free source form
```

the spaces are invalid in free source form but valid in fixed source form.

Comments

An exclamation mark (`!`) indicates the beginning of a comment in free source form, except where it appears in a character context. The compiler considers the rest of the line following the exclamation mark as part of the comment. Embedding a comment inside program text within a single source line is not allowed, but it can follow program text on a source line. A statement on a line with a trailing comment can be continued on subsequent lines.

Statement continuation

A statement can be split over two or more source lines by appending an ampersand character (&) to each source line except the last. The ampersand must not be within a character constant.

A statement can occupy up to 40 source lines. As an extension, HP Fortran 90 increases this limit to 100 source lines. The `END` statement cannot be split by means of a continuation line. Comments are not statements and cannot be continued.

The text of the source statement in a continuation line is assumed to resume from column 1. However, if the first nonblank symbol in the line is an ampersand, the text resumes from the first column after the ampersand.

Consider the following two statements:

```
INTEGER   marks, total, difference,&   ! work variables
    mean, average

INTEGER   marks, total, difference, mean_&  ! work variables
    &value, average
```

The second statement declares the integer variable, `mean_value`. Any spaces appearing in the variable name as a result of the continuation would be invalid. This is the reason for the ampersand character in the continuation line. (Alternatively, `value` could have been positioned at column 1.) Using the ampersand character to split lexical tokens and character constants across source lines is permitted, but not recommended.

Fixed source form

Statements or parts of statements must be written between character columns 7 and 72. Any text following column 72 is ignored. The `+[no]extend_source` option extends the statement to column 254. Columns 1-6 are reserved for special use.

NOTE Programs that depend on the compiler's ignoring any characters after column 72 will not compile correctly with the `+extend_source` option.

Multiple statements may appear on the same line, separated by a semicolon (`;`).

Spaces

Spaces are not significant except within a character context. For example, the two statements

```
RETURN
R E T U R N
```

are equivalent, but

```
c = "abc"
c = "a b c"
```

are not.

Source lines

There are three types of lines in fixed source form:

- Initial line
- Continuation line
- Comment line

The following sections describe each type of source lines.

Initial line

An initial line has the following form:

- Columns 1 to 5 may contain a statement label.
- Column 6 contains a space or the digit zero.
- Columns 7 to 72 (optionally, to 254) contain the statement.

Continuation line

A continuation line has the following form:

- Columns 1 to 5 are blank.
- Column 6 contains any character other than zero or a space. One practice is to number continuation lines consecutively from 1.
- Columns 7 to 72 (optionally, to 254) contain the continuation of a statement.

The Standard specifies that a statement must not have more than 19 continuation lines. As an extension to the Standard, HP Fortran 90 allows as many as 99 continuation lines.

Comment line

Comment lines may be included in a program. Comment lines do not affect compilation in any way, but usually include explanatory notes. The letter `C`, or `c` , or an asterisk (`*`) in column 1 of a line, designates that line as a comment line; the comment text is written in columns 1 to 72. The compiler treats a line containing only blank characters in columns 1 to 72 as a comment line. In addition, a line is considered to be a comment when there is an exclamation mark (`!`) in column 1 or in any column except column 6.

The following are HP extensions to the comment:

- A line with `D` or `d` in column 1 is by default treated as a comment. The `+dlines` option causes the compiler to treat such lines as statements to be compiled. This extension to the comment—called *debugging lines*—is useful for including `PRINT` statements that are to be compiled during the debugging stage to display the program state.

- A line with a pound sign (#) character in column 1 is treated as a comment. This extension allows compilation of source files that have been preprocessed with the C preprocessor (`cpp`).
- HP Fortran 90 allows tab formatting. That is, a tab character may be entered in the first column of a line to skip past the statement label columns. If the character following the tab character is a digit, this digit is assumed to be in column 6, the continuation indicator column. Any other character following the tab character is assumed to be in column 7, the start of a new statement. A tab character in any other column of a line is treated as a space.

INCLUDE line

The INCLUDE line is a directive to the compiler, not a Fortran 90 statement. It causes the compiler to insert text into a program before compilation. The inserted text is substituted for the INCLUDE line and becomes part of the compilable source text. The format of an INCLUDE line is:

INCLUDE *char-literal-const*

where *char-literal-const* is the name of a file containing the text to be included. The character literal constant must not have a kind parameter that is a named constant.

If *char-literal-const* is only a filename (in other words, no pathname is specified), the compiler searches a user-specified path. You can use the -I*dir* option to tell the compiler where to search for files to be included.

The INCLUDE line must appear on one line with no other text except possibly a trailing comment. There must be no statement label. This means, for example, that it is not possible to branch to it, and it cannot be the action statement that is part of an IF statement. Putting a second INCLUDE or another Fortran 90 statement on the same line using a semicolon as a separator is not permitted. Continuing an INCLUDE line using an ampersand is also not permitted.

The text of the included file must consist of complete Fortran 90 statements.

INCLUDE lines may also be nested. That is, a second INCLUDE line may appear within the text to be included, and the text that it includes may also have an INCLUDE line, and so on. HP Fortran 90 has a maximum INCLUDE line nesting level of 10. However, the text inclusion must not be recursive at any level; for example, included text A must not include text B if B includes text A.

The following are example INCLUDE lines:

```
INCLUDE "MY_COMMON_BLOCKS"
INCLUDE "/usr/include/machine_parameters.h"
```

In the next example, the `INCLUDE` line occurs in the executable part of a program and supplies the code that uses the input value from the preceding `READ` statement:

```
READ *, theta
INCLUDE "FUNCTION_CALCULATION"
```

3 Data types and data objects

This chapter describes how data is represented and stored in HP Fortran 90 programs, and includes the following topics:

- Intrinsic data types
- Derived types
- Pointers

Arrays are described in Chapter 4, “Arrays,” on page 51. The `RECORD` and `STRUCTURE` statements—HP Fortran 90 extensions—are fully described in Chapter 10, “HP Fortran 90 statements,” on page 241. Intrinsics procedures are described in Chapter 11, “Intrinsic procedures,” on page 475.

Intrinsic data types

The intrinsic data types are the data types predefined by the HP Fortran 90 language, in contrast with derived types, which are user-defined (see “Derived types” on page 39). The intrinsic data types include numeric types:

- Integer
- Real
- Complex

and nonnumeric types:

- Character
- Logical

Each type allows the specification of a kind parameter to select a data representation for that type (see “Type declaration for intrinsic types” on page 24 for the format of the kind parameter). If the kind parameter is not specified, each type has a default data representation. Table 5 identifies the data representation for each type, including the default case where a kind parameter is not specified. The types are listed by keyword and applicable kind parameter. The table also includes the data representation for the HP extensions, `BYTE` and `DOUBLE COMPLEX`.

As shown in Table 5, HP Fortran 90 aligns data on natural boundaries. Entities of the intrinsic data types are aligned in memory on byte boundaries of 1, 2, 4, or 8, depending on their size. Array variables are aligned on an address that is a multiple of the alignment required for the scalar variable with the same type and kind parameters.

NOTE The ASCII character set uses only the values 0 to 127 (7 bits), but the HP Fortran 90 implementation allows use of all 8 bits of a character entity. The processing of character sets requiring multibyte representation for each character makes use of all 8 bits.

For additional information about data representation models, see “Data representation models” on page 480.

Table 5 Intrinsic data types

Type	Range of values	Precision (in decimal digits)	Bytes	Alignment
`INTEGER(1)`	-128 to 127	Not applicable	1	1
`INTEGER(2)`	-2^{15} to $2^{15}-1$	Not applicable	2	2
`INTEGER(4)` (default)	-2^{31} to $2^{31}-1$	Not applicable	4	4
`INTEGER(8)`	-2^{63} to $2^{63}-1$	Not applicable	8	8
`REAL(4)` (default)	$-3.402823\text{x}10^{38}$ *to* $-1.175495\text{x}10^{-38}$ 0.0 $+1.175495\text{x}10^{-38}$ *to* $+3.402823\text{x}10^{38}$	6 to 9	4	4
`REAL(8)`	$-1.797693\text{x}10^{+308}$ *to* $-2.225073\text{x}10^{-308}$ 0.0 $+2.225073\text{x}10^{-308}$ *to* $+1.797693\text{x}10^{+308}$	15 to 17	8	8
`REAL(16)`	$-1.189731\text{x}10^{+4932}$ *to* $-3.362103\text{x}10^{-4932}$ 0.0 $+3.362103\text{x}10^{-4932}$ *to* $+1.189731\text{x}10^{+4932}$	33 to 35	16	8
`DOUBLE PRECISION`	Same as for `REAL(8)`	15 to 17	8	8
`COMPLEX(4)`	Same as for `REAL(4)`	Same as for `REAL(4)`	8	4
`COMPLEX(8)`	Same as for `REAL(8)`	Same as for `REAL(8)`	16	8

DOUBLE COMPLEX	Same as for REAL(8)	Same as for REAL(8)	16	8
CHARACTER(1) (default)	ASCII character set	Not applicable	1	1
LOGICAL(1)	.TRUE. and .FALSE.	Not applicable	1	1
LOGICAL(2)	.TRUE. and .FALSE.	Not applicable	2	2
LOGICAL(4) (default)	.TRUE. and .FALSE.	Not applicable	4	4
LOGICAL(8)	.TRUE. and .FALSE.	Not applicable	8	8

Type declaration for intrinsic types

The following is the general form of a type declaration statement for the intrinsic data types:

type-spec[[,*attribute-spec*] ... ::] *entity-list*

type-spec

is one of :

- INTEGER [*kind-selector*]
- REAL [*kind-selector*]
- DOUBLE PRECISION [*kind-selector*]
- CHARACTER [*char-selector*]
- LOGICAL [*kind-selector*]
- COMPLEX [kind-selector]
- DOUBLE COMPLEX
- BYTE

BYTE and DOUBLE COMPLEX are HP extensions. BYTE is equivalent to INTEGER(KIND=1). DOUBLE PRECISION is equivalent to REAL(KIND=8), and DOUBLE COMPLEX is equivalent to COMPLEX(KIND=8), except when +autodbl or +autodbl4 is used. Refer to the *HP Fortran Programmer's Guide* for information about using these options to increase sizes. Refer to Chapter 10, “HP Fortran 90 statements,” on page 241 for information about each *type-spec*.

If *type-spec* is present, it overrides the implicit-typing rules; see “Implicit typing” on page 28.

As an HP extension to the Standard, *type-spec* can also take the form:

*type***length*

where *type* is an intrinsic type excluding `BYTE`, `CHARACTER`, `DOUBLE COMPLEX`, and `DOUBLE PRECISION`; and *length* is the number of bytes of storage required, as shown in Table 5. Alternatively, **length* may appear after the entity name. If the entity is an array with an array specification following it, **length* may appear after the array specification. If **length* appears with the entity name, it overrides the length specified by *kind-selector*.

kind-selector

is

`([KIND=]`*scalar-int-init-expr*`)`

scalar-int-init-expr

is a scalar integer initialization expression that must evaluate to one of the kind parameters available (see Table 5). For information about initialization expressions, see “Initialization expressions” on page 90.

char-selector

specifies the length and kind of the character variable, when *type-spec* is `CHARACTER`; see “CHARACTER” on page 268 for details.

attribute-spec

is one or more of the attributes listed in Table 6. Some attributes are incompatible with others; for information about which attributes are compatible as well as full descriptions of all the attributes, see Chapter 10, “HP Fortran 90 statements,” on page 241.

entity-list

is a comma-separated list of entity names of the form:

- *var-name* [(*array-spec*)] [*char-len*] [= *init-expr*]
- *function-name*[(*array-spec*)] [*char-len*]

where*array-spec* is described in “Array declarations” on page 54; *char-len* is described with the CHARACTER statement in Chapter 10; and *init-expr* is described in “Initialization expressions” on page 90. If you include *init-expr* in *entity*, you must also include the double colon (::) separator.

As an extension to the Standard, HP Fortran 90 permits the use of slashes to delimit *init-expr*. The double colon separator, array constructors, and structure constructors are not allowed in this form of initialization. Arrays may be initialized by defining a list of values that are sequence associated with the elements of the array.

Table 6 Attributes in type declaration statement

Attribute	Description
AUTOMATIC	Makes procedure variables automatic (extension).
ALLOCATABLE	Declares an array that can be allocated during execution.
DIMENSION(*array-spec*)	Declares an array; see “Array declarations” on page 54. If *entity-list* also includes an *array-spec*, it overrides the DIMENSION attribute.
EXTERNAL	Specifies a subprogram or block data located in another program unit.
INTENT	Defines the mode of use of a dummy argument.
INTRINSIC	Allows a specific intrinsic name as an actual argument.
OPTIONAL	Declares the presence of an actual argument as optional.
PARAMETER	Defines named constants.
POINTER	Declares the entity to be a pointer.

Attribute	Description
PRIVATE	Inhibits visibility outside a module.
PUBLIC	Provides visibility outside a module.
SAVE	Ensures the entity retains its value between calls of a procedure.
STATIC	Ensures the entity retains its value between calls of a procedure (extension).
TARGET	Enables the entity to be the target of a pointer.
VOLATILE	Provides for data sharing between asynchronous processes (extension).

The following are examples of type declaration statements:

```
! Default, KIND=4, integers i j k.
INTEGER i, j, k

! Using optional separator.
INTEGER :: i,j,k

! An 8-byte initialized integer.
INTEGER(KIND=8) :: i=2**40

! 10 element array of 8-byte integers.
INTEGER(8),DIMENSION(10) :: i

! Using an array constructor for initialization.
REAL, DIMENSION(2,2):: a = RESHAPE((/1.,2.,3.,4./),(/2,2/))

! Initialized complex.
COMPLEX :: z=(1.0,2.0)

! SYNTAX ERROR - no :: present.
COMPLEX z=(1.0,2.0)     ! ILLEGAL

! Initialization using the HP slash extension
INTEGER i/1/,j/2/
REAL a(2,2)/1.1,2.1,1.2,2.2/ ! a(i,j)=i.j

! One character (default length).
CHARACTER(KIND=1) :: c

! A 10-byte character string.
CHARACTER(LEN=10) :: c

! Length can be * for a named constant; title is a 13-byte
!   character string
CHARACTER(*),PARAMETER :: title='Ftn 90 MANUAL'

! next four declarations are all equivalent, but only the last
!   is standard-conforming
REAL*8 r8(10)
```

```
REAL r8*8(10)
REAL r8(10)*8
REAL(8), DIMENSION(10) :: r8

! If the statement is in a subprogram, n must be known at entry;
!    otherwise, it must be a constant.
CHARACTER(LEN=n) ::  c
SUBROUTINE x(c)
  CHARACTER*(*) :: c
  ! c assumes the length of the actual argument.
END

! A single entity, of derived type node.
TYPE(node):: list_element

! Declaration and initialization of a user-defined variable
TYPE(coord) :: origin = coord(0.0,0.0)
```

Implicit typing

In Fortran 90, an entity may be used without having been declared in a type declaration statement. The compiler determines the type of the entity by applying implicit typing rules. The default implicit typing rules are:

- Names with initial letter `A` to `H` or `O` to `Z` are of type real.
- Names with initial letter `I` to `N` are of type integer.

Because Fortran 90 is a case-insensitive language, the same rules apply to both uppercase and lowercase letters.

The following statements

```
DIMENSION a(5), i(10)
k = 1
b = k
```

implicitly declare `a` and `b` as default reals and `i` and `k` as default integers.

As described in Chapter 10, the `IMPLICIT` statement enables you to change or cancel the default implicit typing rules. The `IMPLICIT` statement takes effect for the scoping unit in which it appears, except where overridden by explicit type statements.

You can override the implicit typing rules and enforce explicit typing—that is, declare entities in type declaration statements—with the `IMPLICIT NONE` statement. If this statement is included in a scoping unit, all names in that unit must have their types explicitly declared. You can also enforce explicit typing for all names within a source file by compiling with the `+implicit_none` option. This option has the effect of including an `IMPLICIT NONE` statement in every program unit within a source file.

For a full description of the `IMPLICIT` and `IMPLICIT NONE` statements, see Chapter 10, “HP Fortran 90 statements,” on page 241. The `+implicit_none` option is described in the *HP Fortran 90 Programmer’s Guide*.

Constants

Constants can be either literal or named. A **literal constant** is a sequence of characters that represents a value. A **named constant** is a variable that has been initialized and has the `PARAMETER` attribute. This section describes the formats of literal constants for each of the intrinsic data types. For more information about named constants and the `PARAMETER` statement and attribute, see Chapter 10.

Integer constants

The format of a signed integer literal constant is:

[*sign*] *digit-string* [_*kind-parameter*]

sign

is either + or -.

digit-string

takes the form:

digit[*digit*] ...

kind-parameter

is one of:

- *digit-string*
- the name of a scalar integer constant

The following are examples of integer constants:

```
-123
123_1
123_ILEN
```

In the last example, `ILEN` is a named integer constant used as a kind parameter. It must have a value of 1, 2, 4, or 8.

BOZ constants

Fortran 90 allows DATA statements to include constants that are formatted in binary, octal, or hexadecimal base. Such constants are called **BOZ constants**.

A binary constant is:

leading-letter{'*digit-string*'|"*digit-string*"}

where *leading-letter* is the single character B, O, or Z, indicating binary, octal, or hexadecimal base, respectively. *digit-string* must consist of digits that can represent the base, namely:

- Binary: 0 and 1.
- Octal: 0 through 7.
- Hexadecimal: 0 through 9, and A through F. The letters can be uppercase or lowercase.

In the following, the three DATA statements use BOZ constants to initialize i, j, and k to the decimal value 74:

```
INTEGER i, j, k
DATA i/B'01001010'/
DATA j/O'112'/
DATA k/Z'4A'/
```

As an extension, HP Fortran 90 allows octal constants with a trailing O, and hexadecimal constants with a trailing X. The following DATA statements initialize j and k to the decimal value 74:

```
DATA j/'112'O/
DATA k/'4A'X/
```

HP Fortran 90 also allows the use of BOZ constants in contexts other than the DATA statement; see “Typeless constants” on page 31.

Hollerith constants

Hollerith constants have the form:

*len*H*string*

where *len* is the number of characters in the constant and *string* contains exactly *len* characters. The value of the constant is the value of the pattern of bytes generated by the ASCII values of the characters.

As an extension, HP Fortran 90 allows Hollerith constants to appear in the same contexts as BOZ constants (see “Typeless constants” on page 31), as well as wherever a character string is valid. If *len* is greater than the number of characters in *string*, the constant is padded on the right with space characters. If *len* is less than the number of characters in *string*, the constant is truncated on the right.

If a Hollerith constant appears as an argument to the conversion functions `INT` and `LOGICAL`, the kind parameter is `KIND=1` if the length of the constant is 1 byte, `KIND=2` if the length is 2 bytes, `KIND=4` if 3 0r 4 bytes, and `KIND=8` if greater than 4.

Following are examples of Hollerith constants:

```
3HABC
5HABCbb !bb = two space characters, making the length equal to 5
```

Typeless constants

HP Fortran 90 extends the uses of binary, octal, and hexadecimal constants (BOZ) beyond those prescribed in the Fortran 90 Standard; see “BOZ constants” on page 30. HP Fortran 90 allows BOZ constants to be used as **typeless constants** wherever an intrinsic literal constant of any numeric or logical type is permitted.

If possible, the type attached to a typeless constant is derived from the magnitude of the constant and the context in which it appears. When used as one operand of a binary operator, it assumes the type of the other operand. If it is used as the right-hand side of an assignment, the type of the object on the left-hand side is assumed. When used to define the value within a structure constructor, it assumes the type of the corresponding component. If appearing in an array constructor, it assumes the type of the first element of the constructor.

The following rules and restrictions also apply:

- If the context does not determine the type, a warning is issued and the type attached to the constant is:
  - `INTEGER(4)` if the constant occupies 1-4 bytes.
  - `INTEGER(8)` if the constant occupies more than 4 bytes.

  Leading zeros are considered significant in determining the size.

  For example, `Z'00000001'` assumes `INTEGER(4)`, and `Z'000000001'` assumes `INTEGER(8)`.

- The compiler truncates and issues a warning if more than 8 bytes are required to represent a constant—for example, Z'12345678123456781234'. The resulting truncated value differs from that specified in the source code.
- When the size of the type determined by context does not match the size of the actual constant, the constant is either extended with zeroes on the left or truncated from the left as necessary.
- If a single constant is assigned to a complex entity, it is assumed to represent the real part only and will assume the real type with the same length as the complex entity.
- When the compiler attempts to resolve a generic procedure, a BOZ constant in the argument list is considered to match a logical or numeric dummy argument. An ambiguous reference is likely to occur. See “Generic procedures” on page 154 for information about generic procedures.
- Except for the intrinsic conversion procedures, a BOZ constant used as an actual argument for an intrinsic procedure assumes the integer type.
- The intrinsic functions INT, LOGICAL, REAL, DBLE, DREAL, CMPLX, and DCMPLX are available to force a BOZ constant to a specific type. If a BOZ constant is specified as an argument to these functions, its assumed type is determined as follows:
  - For functions INT and LOGICAL the assumed type will be (respectively) INTEGER(KIND=4) and LOGICAL(KIND=4), if the constant occupies 1 to 4 bytes; otherwise, the type is assumed to be INTEGER(KIND=8) and LOGICAL(KIND=8).
  - For the functions REAL, DBLE, DREAL, CMPLX, and DCMPLX an argument of type REAL(KIND=4) is assumed if the constant occupies 1 to 4 bytes, REAL(KIND=8) if it occupies 5 to 8 bytes, and REAL(KIND=16) otherwise.

The following examples illustrate the extended use of BOZ constants:

```
! The value is 20 (constant treated as INTEGER(2) and
!   truncated on the left).
10_2 + Z'1000A'

LOGICAL(2) :: lgl2
! Constant treated as LOGICAL(2), the type of the variable.
lgl2 = B'1'
```

```
! Constant treated as INTEGER(4); IABS is used.
ABS(Z'41')

! Constant treated as REAL(8) as it is more than 4 bytes.
REAL(Z'3FF0000000000000')
```

Real constants

A signed real literal constant is one of:

[*sign*]*digit-string*[[.[*digit-string*]]][*exponent*][_*kind-parameter*]

exponent

takes the form:

exponent-letter [*sign*] *digit-string*

exponent-letter

is the character E, D, or Q. Q is an HP Fortran 90 extension.

sign and *digit-string*

are explained in “Integer constants” on page 29.

If no kind parameter is present, or if the *exponent letter* E is present, the default kind representation is used; see Table 5. If the *exponent letter* is D, the kind parameter is 8, and if the *exponent letter* is Q, the kind parameter is 16. If both an *exponent* and a kind parameter are specified, the *exponent letter* must be E.

Following are examples of real constants:

```
3.4E-4       !0.00034
42.E2        !4200
1.234_8      !1.234 with approximately 15 digits precision
-2.53Q-300   !-2.53 x 10 to the -300th, with approximately 34
             !   digits precision
```

Complex constants

A complex literal constant has the form:

(*real-part*, *imaginary-part*)

real-part and *imaginary-part*
are each one of:

- *signed-integer-literal-constant*
- *signed-real-literal-constant*

The kind parameter of the complex value corresponds to the kind parameter of the part with the larger storage requirement.

Following are examples of complex constants:

```
(1.0E2, 2.3E-2)  !default complex value
(3.0_8,4.2_4)    !complex value with KIND=8
```

Character constants

A character literal constant is one of:

[*kind-parameter*_]'*character-string*'

[*kind-parameter*_]"*character-string*"

The delimiting characters are not part of the constant. If you need to place a single quote in a string delimited by single quotes, use two single quotes; the same rule applies for double quotes.

Following are examples of character constants:

```
1_'A.N.Other'
'Bach''s Preludes'   ! actual constant is:  Bach's Preludes
""                   ! a zero length constant
```

For compatibility with C-language syntax, HP Fortran 90 allows the backslash character (\) as an escape character in character strings. You must use the `+escape` option to enable this feature. When this option is enabled, the compiler ignores the backslash character and either substitutes an alternative value for the character following, or interprets the character as a quoted value. For example:

```
'ISN\'T'
```

is a valid string when compiled with the `+escape` option.

The backslash is not counted in the length of the string. Also, if \& appears at the end of a line when the +escape option is enabled, the ampersand is not treated as a continuation indicator.

Table 7 lists recognized escape sequences.

Table 7 **Escape characters**

Escape character	Effect
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe (does not terminate a string)
\"	Double quote (does not terminate a string)
\\	\
x	*x*, where *x* is any character other than 1

Logical constants

The format of a logical literal constant is:

{.TRUE.|.FALSE.}[_*kind-parameter*]

The following are examples of logical constants:

```
.TRUE.
.FALSE._2
```

In standard-conforming programs, a logical value of .TRUE. is represented by 1, and .FALSE. is represented by 0. In nonstandard-conforming programs involving arithmetic operators with logical operands, a logical variable may be assigned a value other than 0 or 1. In this case, any nonzero value is .TRUE., and only the value zero is .FALSE.

Character substrings

A character substring is a contiguous subset of a character string. The substring is defined by the character positions of its start and end within the string, formatted as follows:

string ([*starting-position*] : [*ending-position*])

starting-position
: is a scalar expression. If *starting-position* is omitted, a value of 1 is assumed. The *starting-position* must be greater than or equal to 1, unless the substring has zero length.

ending-position
: is a scalar integer expression. If *ending-position* is omitted, the value of the length of the character string is assumed.

The length of the substring is:

MAX (*ending-position* - *starting-position* + 1, 0)

The following example, substring.f90, illustrates the basic operation on a substring.

substring.f90

```
PROGRAM main
 CHARACTER(LEN=15) :: city_name

  city_name = 'CopXXXagen'
  PRINT *, “The city's name is: “, city_name
  city_name(4:6) = 'enh'  ! assign to a substring of city_name
  PRINT *, “The city's name is: “, city_name
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 substring.f90
$ a.out
 The city’s name is: CopXXXagen
 The city’s name is: Copenhagen
```

For information about substring operations on an array of strings, see “Array sections” on page 63.

Character strings as automatic data objects

An automatic data object can be either an automatic array (see “Explicit-shape arrays” on page 55) or a character string that is local to a subprogram and whose size is nonconstant. The size of a character string is determined when the subprogram is called and can vary from call to call.

An automatic character string must not be:

- A dummy argument
- Declared with the SAVE attribute
- Initialized in a type declaration statement or DATA statement

The following example, swap_names.f90, illustrates the use of automatic character strings:

swap_names.f90

```
PROGRAM main
    ! actual arguments to pass to swap_names
    CHARACTER(6) :: n1 = "George", n2 = "Martha"
    CHARACTER(4) :: n3 = "pork", n4 = "salt"

    PRINT *, "Before: n1 = “, n1, "  n2 = “, n2
    CALL swap_names(n1, n2)
    PRINT *, "After:  n1 = “, n1, "  n2 = “, n2

    PRINT *, "Before: n3 = “, n3, "  n4 = “, n4
    CALL swap_names(n3, n4)
    PRINT *, "After:  n3 = “, n3, "  n4 = “, n4
END PROGRAM main

! swap the arguments - two character strings of the same length
SUBROUTINE swap_names (name1, name2)
    CHARACTER(*) :: name1, name2  ! the arguments
    ! declare another character string, temp, to be used in the
    !   exchange.  temp is an automatic data object, its length
    !   can vary from call to call
    CHARACTER(LEN(name1)) :: temp

    ! the exchange
    temp = name1
    name1 = name2
    name2 = temp
END SUBROUTINE swap_names
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 swap_names.f90
$ a.out
 Before: n1 = George  n2 = Martha
 After:  n1 = Martha  n2 = George
 Before: n3 = pork  n4 = salt
 After:  n3 = salt  n4 = pork
```

Derived types

Derived types are user-defined types that are constructed from entities of intrinsic data types (see “Intrinsic data types” on page 22) or entities of previously defined derived types. For example, the following is a definition of a derived type for manipulating coordinates consisting of two real numbers:

```
TYPE coord
  REAL :: x,y
END TYPE coord
```

x and y are the components of the derived type coord.

The next statement declares two variables (a and b) of the derived type coord:

```
TYPE(coord) :: a, b
```

The next statement copies the values of a to b, as in any assignment statement:

```
a = b
```

The components of a and b are referenced as a%x, a%y, b%x, and b%y. By using the defined operation facility of Fortran 90, it is possible to extend the standard operators to work with derived types. For example, if the + and = operators were re-defined to operate on derived type operands, the following statement

```
a = a + b
```

would be equivalent to

```
a%x = a%x + b%x;  a%y = a%y + b%y
```

The following sections describe:

- The syntax of defining a derived type
- Sequence types
- Structure constructors
- Referencing a structure component
- Alignment of derived type objects

The last section provides an example program that illustrates different features of derived types.

Defining a derived type

The format for defining a derived type is:

TYPE [[, *access-spec*] ::] *type-name*

[*private-sequence-statement*] ...

comp-definition-statement

[*comp-definition-statement*] ...

END TYPE [*type-name*]

access-spec

is one of:

- PRIVATE
- PUBLIC

access-spec is allowed only if the definition appears within a module. For more information about modules, see “Modules” on page 161. The PRIVATE and PUBLIC attributes are described in Chapter 10.

type-name

is the name of the type being defined. *type-name* must not conflict with the intrinsic type names.

private-sequence-statement

is a PRIVATE or SEQUENCE statement. The PRIVATE statement is allowed only if the definition appears within a module. For more information about the SEQUENCE statement, see “Sequence derived type” on page 41. Both statements are fully described in Chapter 10.

comp-definition-statement

takes the form:

type-spec [[*comp-attr-list*]::]*comp-decl*

Notice that the syntax does not allow for initialization.

comp-attr-list

can only contain the DIMENSION and POINTER attributes. A component array without the POINTER attribute must have an explicit-shape specification with constant bounds. If a component is of the same derived type as the type being defined then the component must have the `POINTER` attribute. Both attributes are fully described in Chapter 10.

comp-declaration

takes the form:

comp-name [(*array-spec*)] [* *char-len*]

where *array-spec* is an array specification, as described in “Array declarations” on page 54; and *char-len* is used when *comp-name* is of type character to specify its length, as explained in “CHARACTER” on page 268.

For a full description of the `TYPE` statement that is used to define a derived type, see “TYPE (definition)” on page 457.

Sequence derived type

As shown in “Defining a derived type” on page 40, the `SEQUENCE` statement may appear in the definition of a derived type. When storage for a variable of derived type is allocated, the presence of the `SEQUENCE` statement in the definition of the derived type causes the compiler to arrange all components in a storage sequence that is the same as the order in which they are defined. Such a derived type is called a **sequence derived type**.

A sequence derived type may appear in a common block or in an equivalence set. The Standard makes requirements about the type—numeric or character—of the components in a sequence type. As an extension, HP Fortran 90 makes no restrictions on the types of the components other than that the definition of the derived type must include the `SEQUENCE` statement. For more information about sequence derived types, refer to “SEQUENCE” on page 432.

Structure component

A component of a derived-type object may be referenced and used like any other variable—in an expression, on the lefthand side of an assignment statement, or as procedure argument. It may be a scalar, an array, or itself a derived-type object. The component name has the same scope as the derived-type object in which it is declared.

To reference a structure component, use the form:

parent-name[%*comp-name*]...%*comp-name*

parent-name is a derived type. This part of a structure component reference is the parent and is joined to *comp-name* by the component selector operator (%). The *comp-name* component to which the parent is joined on its immediate right must be a component of *parent-name*. If *parent-name* has the INTENT, TARGET, or PARAMETER attribute, then the structure component being referenced—the rightmost *comp-name*—also has that attribute.

comp-name is the name of a component. If more than one *comp-name* appears in a structure component reference, the reference is to the rightmost *comp-name*. If more than one *comp-name* appears in the reference, each one (except the rightmost) must be a derived-type object, and the *comp-name* to its immediate right must be one of its declared components.

If *parent-name* and *comp-name* are arrays, each can be followed by a *section-subscript-list* enclosed in parentheses. See “Array sections” on page 63 for information about the syntax of *section-subscript-list*. The Standard imposes certain restrictions on structure component references that are array-valued, as described in “Array-valued structure component references” on page 68.

If the definition of a derived type contains a component that is of the same derived type, the component must have the POINTER attribute. The following example defines the derived type node, which includes a component (next) of the same derived type:

```
TYPE node  ! for use in a singly linked list
  INTEGER :: value
  TYPE(node), POINTER :: next  ! must have the POINTER attribute
END TYPE node
```

Declaring a derived type-object

To declare an object of derived type, use the TYPE statement, as follows:

TYPE (*type-name*) [[, *attrib-list*] ::] *entity-list*

where *type-name*, *attrib-list*, and *entity-list* all have the same meaning as in a type declaration statement that is used to declare an object of an intrinsic type; see “Type declaration for intrinsic types” on page 24. For more detailed information, see “TYPE (definition)” on page 457.

Structure constructor

A structure constructor constructs a scalar value of derived type. The value is constructed of a sequence of values for each component of the type. The syntax of a structure constructor is:

type-name (*expression-list*)

type-name
: is the name of the derived type. The name must have been previously defined.

expression-list
: is a comma-separated list of expressions that must agree in number, order, and rank with the components in *type-name*. For information about expressions, see “Expressions” on page 80 and “Special forms of expression” on page 89.

The following restrictions apply to the use of the structure constructor:

- If a component is of derived type, an embedded structure constructor must be used to specify a value for the derived-type component.
- If a component is an array, an array constructor must appear in *expression-list* that satisfies the array. For more information about array constructors, see “Array constructors” on page 71.
- If a component is a pointer, the corresponding expression in *expression-list* must evaluate to an allowable target.

Alignment of derived-type objects

Derived type objects have the same alignment as the component that has the most restrictive alignment requirement. (This rule also applies to records.) To ensure natural alignment, the compiler may add padding to each element in an array of derived type.

The following illustrates the alignment of an array of derived type. The definition of the derived type includes the SEQUENCE statement to ensure the order in which components are laid out in memory is the same as in the definition. The SEQUENCE statement has no effect on alignment:

```
! definition of a derived type
TYPE t
   SEQUENCE
   CHARACTER(LEN=7) :: c
   INTEGER(2) :: i2
   REAL(8) :: r8
   REAL(4) :: r4
END TYPE t

! declaration of an array variable of derived type
TYPE (t), DIMENSION(5) :: ta
```

Each element of t is allocated storage as shown in Table 8. The first component of t starts at an address that is a multiple of 8. The four trailing padding bytes are necessary to preserve the alignment of r8 in each element of the array.

Table 8 **Example of structure storage**

Component	Byte offset	Length
c	0	7
i2	8	2
r8	16	8
r4	24	4
padding	28	4

A derived-type example

The example below, traffic.f90, illustrates how to define a derived type, declare a variable of the type, specify a value for the variable using the structure constructor, pass the variable as an argument to another procedure, and reference a structure component. The derived type is defined in a module so that it can be made accessible by use association.

For more information about modules and the USE statement, see “Modules” on page 161. The MODULE and USE statements are also described in Chapter 10.

traffic.f90

```
PROGRAM traffic
! Illustrates derived types:  defines a derived type, declares an
! to array variable of derived type, uses a structure constructor
! assign to its components, and passes a component which is
! itself another derived type to a subprogram.

! Make the definition of the derived type called hours accessible
!   to this program unit
USE hours_def

LOGICAL :: busy
INTEGER :: choice

! Define another derived type that uses hours as a component
TYPE hiway
  INTEGER :: rte_num
  TYPE(hours) :: busy_hours
END TYPE hiway

! Declare an array of derived-type structures.
TYPE(hiway), DIMENSION(3) :: route

! Use the structure constructor to specify values for each
! element of route
route(1) = hiway(128, hours(.TRUE., .FALSE.))
route(2) = hiway(93, hours(.FALSE., .TRUE.))
route(3) = hiway(97, hours(.FALSE., .FALSE.))

PRINT *, 'What road do you want to travel?'

PRINT *, '1. Rte. 128'
PRINT *, '2. Rte. 93'
PRINT *, '3. Rte 97'
READ *, choice

! Pass the busy_hours component of the selected route to
! the function busy.
IF (busy(route(choice)%busy_hours)) THEN
  PRINT *,’Heavy commute on rte.’, route(choice)%rte_num
ELSE
```

```
  PRINT *,’Easy commute on rte.’, route(choice)%rte_num
END IF

END PROGRAM traffic

LOGICAL FUNCTION busy(when)
! This function accepts a derived-type argument whose definition
! is defined in the module hours_def, made accessible here by
! use association.  It returns .TRUE. or .FALSE., depending on
! on the value of the user-selected component of the argument.

! Make the definition of hours accessible to this function.
USE hours_def

TYPE(hours) :: when

INTEGER :: choice

PRINT *, 'When do you want to commute:'
PRINT *, '1. Morning'
PRINT *, '2. Evening'
READ *, choice

! Find out if the route is busy at that time of day.
IF (choice .EQ. 1) THEN
  busy = when%am
ELSE
  busy = when%pm
END IF

END FUNCTION busy

MODULE hours_def
  ! Define a derived type, which will be passed as an argument.
  TYPE hours
    LOGICAL :: am
    LOGICAL :: pm
  END TYPE hours
END MODULE hours_def
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 traffic.f90
$ a.out
 What road do you want to travel?
 1. Rte. 128
 2. Rte. 93
 3. Rte 97
1
 When do you want to commute:
 1. Morning
 2. Evening
1
 Heavy commute on rte. 128
```

Pointers

Pointers in Fortran 90 are more strongly typed than in other languages. While it is true that the Fortran 90 pointer holds the address of another variable (the **target**), it also holds additional information about the target. For this reason, declaring a pointer requires not only the POINTER attribute but also the type, kind parameter, and (if its target is an array) rank of the target it can point to.

If a pointer is declared as an array with the POINTER attribute, it is an **array pointer**. As explained in “Deferred-shape arrays” on page 58, the declaration for an array pointer specifies its specifies rank but not the bounds. Following is the declaration of the array pointer ptr:

```
REAL(KIND=16), POINTER, DIMENSION(:,:) :: ptr
```

To become assignable to an array pointer, a target must be declared with the TARGET attribute and must have the same type, kind parameter, and rank as the array pointer. Given the previous declaration of ptr, the following are legal statements:

```
! declare a target with the same type, kind parameter, and
!   rank as ptr
REAL(KIND=16), TARGET, DIMENSION(4,3) :: x
...
ptr => x  ! assign x to ptr in a pointer assignment statement
```

Once the assignment statement executes, you can use either ptr or x to access the same storage, effectively making ptr an alias of x.

You can also allocate storage to a pointer by means of the ALLOCATE statement. To deallocate that storage after you are finished with it, use the DEALLOCATE statement. Although allocating storage to a pointer does not involve a target object, the declaration of the pointer must still specify its type, kind parameter, and (if you want to allocate an array) rank. The ALLOCATE statement specifies the bounds for the dimensions. Here is an example of the ALLOCATE statement used to allocate storage for ptr:

```
INTEGER :: j = 10, k = 20
...
! allocate storage for ptr
ALLOCATE (ptr(j,k))
```

ptr can now be referenced as though it were an array, using Fortran 90 array notation.

As an extension, HP Fortran 90 provides the Cray-style pointer variables; for more information, see Chapter 10. For information about aspects of pointers, refer to:

- "Array pointers" on page 59 for information about allocating array pointers.
- "Pointer assignment" on page 97 for information about associating a pointer with a target by means of pointer assignment.
- Chapter 10, "HP Fortran 90 statements," on page 241 for a full description of the ALLOCATE and DEALLOCATE statements as well as the POINTER and TARGET attributes.

The following section discusses pointer status and includes an example program.

Pointer association status

Certain pointer operations can only be performed depending on the status of the pointer. A pointer's status is called its *association status*, and it can take three forms:

Undefined The status of a pointer is undefined on entry to the program unit in which the pointer is declared or if:

- Its target is never allocated.
- Its target was deallocated (except through the pointer.
- The target goes out of scope, causing it to become undefined.

If the association status is undefined, the pointer must not be referenced or deallocated. It may be nullified, assigned a target, or allocated storage with the ALLOCATE statement.

Associated The status of a pointer is associated if it has been allocated storage with the ALLOCATE statement or is assigned a target. If the target is allocatable, it must be currently allocated.

If the association status is associated, the pointer may be referenced, deallocated, nullified, or pointer assigned.

Disassociated The status of a pointer is disassociated if the pointer has been nullified with the NULLIFY statement or deallocated, either by means of the DEALLOCATE statement or by being assigned to a disassociated pointer.

If the association status is disassociated, the same restrictions apply as for a status of undefined. That is, the pointer must not be referenced or deallocated, but it may be nullified, assigned a target, or allocated storage with the ALLOCATE statement.

You can use the ASSOCIATED intrinsic function to determine the association status of a pointer; see Chapter 11, “Intrinsic procedures,” on page 475 for a description of this intrinsic.

A pointer example

The example below, ptr_sts.f90, illustrates different pointer operations, including calls to the ASSOCIATED intrinsic to determine pointer status.

ptr_sts.f90

```
PROGRAM main
  ! This program performs simple pointer operations, including
  ! calls to the ASSOCIATED intrinsic to determine status.
  !
  ! Declare pointer as a deferred shape array with POINTER
  ! attribute.
  REAL, POINTER :: ptr(:)
  REAL, TARGET :: tgt(2) = (/ -2.2, -1.1 /) ! initialize target

  PRINT *, "Initial status of pointer:"
  call get_ptr_sts

  ptr => tgt   ! pointer assignment
  PRINT *, "Status after pointer assignment:"
  call get_ptr_sts

  PRINT *, "Contents of target by reference to pointer:", ptr

  ! use an array constructor to assign to tgt by reference to ptr
  ptr = (/ 1.1, 2.2 /)

  PRINT *, “Contents of target after assignment to pointer:”, tgt

  NULLIFY(ptr)
  PRINT *, "Status after pointer is nullified:"
  call get_ptr_sts

  ALLOCATE(ptr(5))  ! allocate pointer
```

```
PRINT *, "Status after pointer is allocated:"
  ! To learn if pointer is allocated, call the ASSOCIATED
  ! intrinsic without the second argument
  IF (ASSOCIATED(ptr)) PRINT *, "  Pointer is allocated."

  ptr = (/ 3.3, 4.4, 5.5, 6.6, 7.7 /)   ! array assignment
  PRINT *, 'Contents of array pointer:', ptr

  DEALLOCATE(ptr)
  PRINT *, "Status after array pointer is deallocated:"
  IF (.NOT. ASSOCIATED(ptr)) PRINT *, "  Pointer is deallocated."

CONTAINS
  ! Internal subroutine to test pointer's association status.
  ! Pointers can be passed to a procedure only if its interface
  ! is explicit to the caller.  Internal procedures have an
  ! explicit interface.  If this were an external procedure,
  ! its interface would have to be declared in an interface
  ! block to be explicit.
  SUBROUTINE get_ptr_sts
    IF (ASSOCIATED(ptr, tgt)) THEN
      PRINT *, "  Pointer is associated with target."
    ELSE
      PRINT *, "  Pointer is disassociated from target."
    END IF
  END SUBROUTINE get_ptr_sts
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 ptr_sts.f90
$ a.out
 Initial status of pointer:
   Pointer is disassociated from target.
 Status after pointer assignment:
   Pointer is associated with target.
 Contents of target by reference to pointer: -2.2 -1.1
 Contents of target after assignment to pointer: 1.1 2.2
 Status after pointer is nullified:
   Pointer is disassociated from target.
 Status after pointer is allocated:
   Pointer is allocated.
 Contents of array pointer: 3.3 4.4 5.5 6.6 7.7
 Status after array pointer is deallocated:
   Pointer is deallocated.
```

4 Arrays

This chapter describes arrays and the array-handling features of HP Fortran 90. This includes the following topics:

- Array fundamentals
- Array declarations
- Array-valued structure component references
- Array constructors
- Array expressions
- Array-valued functions
- Array inquiry intrinsics

Array fundamentals

An array consists of a set of **elements**, each of which is a scalar and has the same type and type parameter as declared for the array. Elements are organized into **dimensions**. Fortran 90 allows arrays up to seven dimensions. The number of dimensions in an array determines its **rank**.

Dimensions have an **upper bound** and a **lower bound**. The total number of elements in a dimension—its **extent**—is calculated by the formula:

upper-bound `-` *lower-bound* `+ 1`

The **size** of an array is the product of its extents. If the extent of any dimension is zero, the array contains no elements and is a **zero-sized array**.

Elements within an array are referenced by **subscripts**—one for each dimension. A subscript is a specification expression and is enclosed in parentheses. As an extension, HP Fortran 90 allows a subscript expression of type real; the expression is converted to type integer after it has been evaluated.

The **shape** of an array is determined by its rank and by the extents of each dimension of the array. An array's shape may be expressed as a vector where each element is the extent of the corresponding dimension. For example, given the declaration:

```
REAL, DIMENSION(10,2,5) :: x
```

the shape of `x` can be represented by the vector `[10, 2, 5]`.

Two arrays are **conformable** if they have the same shape, although the lower and upper bounds of the corresponding dimensions need not be the same. A scalar is conformable with any array.

A **whole array** is an array referenced by its name only, as in the following statements:

```
REAL, DIMENSION(10) :: x, y, z
PRINT *, x
x = y + z
```

The **array element order** used by HP Fortran 90 for storing arrays is **column-major order**; that is, the subscripts along the first dimension vary most rapidly, and the subscripts along the last dimension vary most slowly. For example, given the declaration:

```
INTEGER, DIMENSION(3,2) :: a
```

the order of the elements would be:

```
a(1,1)
a(2,1)
a(3,1)
a(1,2)
a(2,2)
a(3,2)
```

The following array declarations illustrate some of the concepts presented in this section:

```
! The rank of a1 is 1 as it only has one dimension, the extent of
! the single dimension is 10, and the size of a1 is also 10.
! a1 has a shape represented by the vector [10].
REAL, DIMENSION(10) :: a1

! a2 is declared with two dimensions and consequently has a rank
! of 2, the extents of the dimensions are 2 and 4
! respectively,and the size of a2 is 8.
! The array's shape can be represented by the vector [2, 4].
INTEGER, DIMENSION(2,4) :: a2

! a3 has a rank of 3, the extent of the first two dimensions is
! 5,and the extent of the third dimension is zero.  The size of
! a3 is the product of all the extents and is therefore zero.
! The shape of a3 can be represented by the vector [5, 5, 0].
LOGICAL, DIMENSION(5,5,0) :: a3

! a and b are conformable, c and d are conformable.  The shape of
! a and b can be represented by the vector [3, 4].  The shape of
! c and d can be represented by the vector [6, 8].
REAL, DIMENSION :: a(3,4), b(3,4), c(6,8), d(-2:3,10:17)
```

Array declarations

An array is a data object with the dimension attribute. Its rank—and possibly the extents—are defined by an array specification. The array specification is enclosed in parentheses and can be attached either to the `DIMENSION` attribute, as in:

```
INTEGER, DIMENSION(17) :: a, b
```

or to the array name, as in:

```
REAL :: y(3,25)
```

If the array specification is attached both to the `DIMENSION` attribute and to the array name in the same declaration statement, the specification attached to the name takes precedence. In the following example:

```
INTEGER, DIMENSION(4,7) :: a, b, c(15)
```

`a` and `b` are declared as two-dimensional arrays, but `c` is declared as a one-dimensional array.

An array specification can declare an array as one of the following:

- Explicit-shape array
- Assumed-shape array
- Deferred-shape array
- Assumed-size array

The following sections describe these types and the form of the array specification for each type. For information about initializing arrays with the array constructor, see “Array constructors” on page 71.

Explicit-shape arrays

An **explicit-shape array** has explicitly declared bounds for each dimension; the bounds are neither taken from an actual array argument (“assumed”) nor otherwise specified prior to use (“deferred”). Each dimension of an explicit-shape array has the following form:

[*lower-bound* :] *upper-bound*

where *lower-bound* and *upper-bound* are specification expressions and may be positive, negative, or zero. The default for *lower-bound* is 1.

For a given dimension, the values of *lower-bound* and *upper-bound* define the range of the array in that dimension. Usually, *lower-bound* is less than *upper-bound*; if *lower-bound* is the same as *upper-bound*, then the dimension contains only one element; if it is greater, then the dimension contains no elements, the extent of the dimension is zero, and the array is zero-sized.

The simplest form is represented by an array declaration in which the name of the array is not a dummy argument and all bounds are constant expressions, as in the following example:

```
INTEGER :: a(100,4,5)
```

This form of array may have the SAVE attribute and may be declared in any program unit.

Other forms of the explicit-shape array include:

- An **automatic array**: An array that is declared in a subprogram but is not a dummy argument and has at least one nonconstant bound. Automatic arrays may be declared in a subroutine or function, but may not have the SAVE attribute nor be initialized.

  Character strings can also be declared as automatic data objects; see “Character strings as automatic data objects” on page 37 and “CHARACTER” on page 268.

- A **dummy array**: An array that is identified by its appearance in a dummy argument list; its bounds may be constants or expressions. Dummy arrays may only be declared in a subroutine or function.

- An **adjustable array**: A particular form of a dummy array. Its name is specified in a dummy argument list, and at least one of its bounds is a nonconstant specification expression.

Explicit-shape arrays may also be used as function results, as described in “Array-valued functions” on page 76 and in “Array dummy argument” on page 140.

The following code segment illustrates different forms of explicit-shape arrays:

```
SUBROUTINE sort(list1,list2,m,n)
!   examples of arrays with explicit shape
INTEGER :: m,n
INTEGER :: cnt1(2:99)
! a rank-one array, having an explicit shape represented by
! the vector [98]
REAL :: list1(100), list2(0:m-1,-m:n)
! two dummy arrays with explicit shape:  list1 is a rank-one
! array with an extent of 100; list2 is a rank-two array with an
! extent of m * (m+n+1).  list2 is also an adjustable array.

REAL :: work(100,n)
! work is an automatic array; it does not appear in the dummy
! argument list and at least one of its bounds is not constant

INTEGER, PARAMETER :: buffsize = 0
REAL :: buffer (1: buffsize)
!  buffer has explicit shape, but no elements and is zero-sized
  .
  .
  .
END SUBROUTINE sort
```

Assumed-shape arrays

An **assumed-shape array** is a dummy argument that assumes the shape of the corresponding actual argument. It must not have the POINTER attribute. Each dimension of an assumed-shape array has the form:

[*lower-bound*] :

where *lower-bound* is a specification expression. The default for *lower-bound* is 1.

The actual argument and the corresponding dummy argument may have different bounds for each dimension. An assumed-shape array subscript may extend from the specified *lower-bound* to an upper bound that is equal to *lower-bound* plus the extent in that dimension of the actual argument minus one.

The following code segment illustrates different declarations of assumed-shape arrays.

```
SUBROUTINE initialize (a,b,c,n)
! examples of assumed-shape arrays
INTEGER :: n

INTEGER :: a(:)
! the array a is a rank-one assumed-shape array, it takes its
! shape and size from the corresponding actual argument; its
! lower bound is 1 regardless of the lower bound defined for
! the actual argument

COMPLEX :: b(ABS(n):)
! a rank-one assumed-shape array, the lower bound is ABS(n) and
! the upper bound will be the lower bound plus the extent of
! the corresponding actual argument minus one

REAL, DIMENSION(:,:,:,:,:) :: c
! an assumed-shape array with 5 dimensions; the lower bound for
! each dimension is 1
  .
  .
  .
END SUBROUTINE initialize
```

If a procedure has an argument that is an assumed-shape array, its interface must be explicit within the calling program unit. A procedure’s interface is explicit if it is an internal procedure within the caller procedure or if the interface is declared in an interface block within the caller.

For example, to call the external subroutine `initialize` in the previous example, its interface must appear in an interface block, as in the following:

```
PROGRAM main
INTEGER :: parts(0:100)
COMPLEX :: coeffs(100)
REAL    :: omega(-2:+3, -1:+3, 0:3, 1:3, 2:3)
INTERFACE
  SUBROUTINE initialize (a,b,c,n)
    INTEGER :: n
    INTEGER :: a(:)
    COMPLEX :: b(ABS(n):)
    REAL, DIMENSION(:,:,:,:,:) :: c
  END SUBROUTINE initialize
END INTERFACE
CALL initialize(parts,coeffs,omega,lbound(omega,1))
  .
  .
  .
END PROGRAM main
```

```
SUBROUTINE initialize (a,b,c,n)
  INTEGER :: n
  INTEGER :: a(:)
  COMPLEX :: b(ABS(n):)
  REAL, DIMENSION(:,:,:,:,:) :: c
.
.
.
END SUBROUTINE initialize
```

For more information about:

- Internal procedures, see “Internal procedures” on page 135
- Interface blocks, see “Procedure interface” on page 151
- Arrays used as dummy arguments, see “Array dummy argument” on page 140

Deferred-shape arrays

A **deferred-shape array** has either the POINTER attribute or the ALLOCATABLE attribute. Its shape is not specified until the array is pointer assigned or allocated. Although a deferred-shape array can have the same form as an assumed-shape array, the two are different. The assumed-shape array is a dummy argument and must not have the POINTER attribute.

The array specification for a deferred-shape array has the form:

```
: [ , : ] ...
```

The specification for a deferred-shape array defines its rank but not the bounds. The bounds are defined either when the array is allocated or when an array pointer becomes associated with a target.

Array pointers and allocatable arrays are described in the following sections.

Array pointers

An **array pointer** is a deferred-shape array with the POINTER attribute. Its bounds and shape are defined only when the array is associated with a target in a pointer assignment statement or in an ALLOCATE statement. An array pointer must not be referenced until it is associated.

Following are example declarations of array pointers:

```
! p1 is declared as a pointer to a rank-one
! array of type real; p1 is not associated with any target
REAL, POINTER, DIMENSION(:) :: p1

! p2 is a pointer to an integer array of rank-two;
! it must be associated with a target before it can be referenced
INTEGER, POINTER :: p2(:,:)

! err is a pointer to a rank-3 array of type err_type
TYPE err_type
   INTEGER :: class
   REAL :: code
END TYPE err_type
TYPE(err_type), POINTER, DIMENSION(:,:,:) :: err

! The next statement is ILLEGAL:  pointers cannot have an
! explicit shape.
INTEGER, POINTER :: p3(n)
```

For information about associating an array pointer with a target, see “Pointers” on page 47. For information about the POINTER attribute and ALLOCATE statement, see Chapter 10, “HP Fortran 90 statements,” on page 241.

Allocatable arrays

An **allocatable array** is a deferred-shape array with the ALLOCATABLE attribute. Its bounds and shape are defined when it is allocated with the ALLOCATE statement. Once allocated, the allocatable array may be used in any context in which any other array may appear. An allocatable array can also be deallocated with the DEALLOCATE statement.

An allocatable array has an allocation status that can be tested with the ALLOCATED intrinsic inquiry function. Its status is *unallocated* when the array is first declared and after it is deallocated in a DEALLOCATE statement. After the execution of the ALLOCATE statement, its status is *allocated*. An allocatable array with the *unallocated* status may not be referenced except as an argument to the ALLOCATED intrinsic or in an ALLOCATE statement. If it has the *allocated* status, it may not be referenced in the ALLOCATE statement. It is an error to allocate an allocatable array that is already allocated, or to deallocate an allocatable array either before it is allocated or after it is deallocated.

In HP Fortran 90, an allocatable array that is *unallocated*, is local to a procedure, and does not have the SAVE attribute. It is automatically deallocated when the procedure exits.

The following example, alloc_array.f90, calls a subroutine that allocates and deallocates an allocatable array and uses the ALLOCATED intrinsic function to test its allocation status:

alloc_array.f90

```
PROGRAM main
! driver program for calling a subroutine that allocates and
! deallocates an allocatable array
  CALL test_alloc_array
END PROGRAM main

SUBROUTINE test_alloc_array
! demonstrate how to allocate and deallocate an allocatable array

  ! the array matrix is rank-2 allocatable array, with no
  ! shape or storage
  REAL, ALLOCATABLE, DIMENSION(:,:) :: matrix

  INTEGER :: n
  LOGICAL :: sts

  ! sts is assigned the value .FALSE. as the array is not yet
  ! allocated
  sts = ALLOCATED(matrix)
  PRINT *, 'Initial status of matrix:  ', sts

  PRINT *, 'Enter an integer (rank of array to be allocated):'
  READ *,n

  ! dynamically create the array matrix; after allocation, array
  ! will have the shape [ n, n ]
  ALLOCATE(matrix(n,n))


  ! test allocation by assigning to array
  matrix(n,n) = 9.1
  PRINT *, 'matrix(',n,',',n,') = ', matrix(n,n)
  ! sts is assigned the value .TRUE. as the allocatable array
  ! does exist and its allocation status is therefore allocated
  sts = ALLOCATED(matrix)
  PRINT *, 'Status of matrix after ALLOCATE:  ', sts

  DEALLOCATE (matrix)

  ! sts is assigned the value .FALSE. as the
```

```
  ! allocation status of a deallocated array
  sts = ALLOCATED (matrix)
  PRINT *, 'Status of matrix after DEALLOCATE:  ', sts
END SUBROUTINE test_alloc_array
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 alloc_array.f90
$ a.out
 Initial status of matrix:   F
 Enter an integer (rank of array to be allocated):
4
 matrix( 4 , 4 ) =  9.1
 Status of matrix after ALLOCATE:   T
 Status of matrix after DEALLOCATE:   F
```

For information about the ALLOCATABLE, ALLOCATE, DEALLOCATE statements, see Chapter 10, “HP Fortran 90 statements,” on page 241. See also “ALLOCATED(ARRAY)” on page 493.

Assumed-size arrays

An **assumed-size array** is a dummy argument whose size is taken from the associated actual argument. Its declaration specifies the rank and the extents for each dimension except the last. The extent of the last dimension is represented by an asterisk (*), as in the following:

```
INTEGER :: a(2,5,*)
```

All dummy array arguments and their corresponding actual arguments share the same initial element and are storage associated. In the case of explicit-shape and assumed-size arrays, the actual and dummy array need not have the same shape or even the same rank. The size of the dummy array, however, must not exceed the size of the actual argument. Therefore, a subscript in the last dimension of an assumed-size array may extend from the lower bound to a value that does not cause the reference to go beyond the storage associated with the actual argument.

Because the last dimension of an assumed-size array has no upper bound, the dimension has no extent and the array consequently has no shape. The name of an assumed-size array therefore cannot appear in contexts in which a shape is required, such as a function result or a whole array reference.

The following example, assumed_size.f90, illustrates two assumed-size arrays: `x` (declared in `subr`) and `i_array` (declared in `func`):

assumed_size.f90

```
PROGRAM main
  REAL :: a(2,3)  ! an explicit-shape array, represented by the
                  !   vector [10, 10]
  k = 0
  DO i = 1, 3
    DO j = 1, 2
      k = k + 1
      a(j, i) = k
    END DO
  END DO

  PRINT *, 'main: a =', a
  CALL subr (a)
END PROGRAM main

SUBROUTINE subr(x)
  REAL :: x(2,*)  ! an assumed-size array; the subscript for the
                  !   last dimension may take any value 1 - 3

! PRINT *, x      ! ILLEGAL, whole array reference not allowed

  PRINT *, ‘main: x(2, 2) = ‘, x(2, 2)

  PRINT *, 'returned by func: ', func(x), ', the value in x(2,3)'
END SUBROUTINE subr

REAL FUNCTION func(y)
  REAL :: y(0:*)  ! an assumed-size array; the subscript may
                  !   take any value 0 - 5

  func = y(5)
END FUNCTION func
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 assumed_size.f90
$ a.out
 main: a = 1.0 2.0 3.0 4.0 5.0 6.0
 main: x(2, 2) =  4.0
 returned by func:  6.0 , the value in x(2,3)
```

An assumed-size array is a FORTRAN 77 feature that has been superseded by the assumed-shape array; see “Assumed-shape arrays” on page 56.

Array sections

An **array section** is a selected portion of another array (the *parent*) that is itself an array, even if it consists of only one element, or possibly none. An array section can appear wherever an array name is allowed.

The syntax for specifying an array section is:

array-name (*section-subscript-list*)[(*substring-range*)]

where:

section-subscript-list
: is a comma-separated list of *section-subscript*.

section-subscript
: is one of:

- *subscript*
- *subscript-triplet*
- *vector-subscript*

subscript
: is a scalar integer expression.

subscript-triplet
: takes the form:
[*subscript*] : [*subscript*] [: *stride*]
where *stride* is a scalar integer expression.

vector-subscript
: is a rank-one integer array expression.

substring-range
: specifies a character substring, as described in "Character substrings" on page 36. If *substring-range* is specified, *array-name* must be of type character.

Section-subscript-list must specify *section-subscript* for each dimension of the parent array. The rank of the array section is the number of *subscript-triplets* and *vector -subscripts* that appear in the *section-subscript-list*. Because an array section is also an array, at least one *subscript-triplet* or *vector-subscript* must be specified.

The following sections provide more information about *subscript-triplet* and *vector-subscript*.

Subscript triplet

A **subscript triplet** selects elements from the parent array to form another array. It specifies a lower bound, an upper bound, and a stride for any dimension of the parent array. Elements are selected in a regular manner from a dimension. The stride can, for example, select every second element.

All three components of a subscript triplet are optional. If a bound is omitted, it is taken from the parent array. However, an upper bound must be specified if a subscript triplet is used in the last dimension of an assumed-sized array.

A bound in a subscript triplet need not be within the declared bounds for that dimension of the parent array if all the elements selected are within its declared bounds. If the stride is omitted, the default is to increment by one.

The stride must not be zero. If it is positive, the subscripts range from the lower bound up to and including the upper bound, in steps of stride. When the difference between the upper bound and lower bound is not a multiple of the stride, the last subscript value selected by the subscript triplet is the largest integer value that is not greater than the upper bound. The array expression `a(1: 9: 3)` selects subscripts 1, 4, and 7 from `a`.

Strides may also be negative. A negative stride selects elements from the parent array starting at the lower bound and proceeds backwards through the parent array in steps of the stride down the last value that is greater than the upper bound. For example, the expression `a(9:1:- 3)` selects the subscripts 9, 6, and 3 in that order from `a`.

If the section bounds are such that no elements are selected in a dimension (for example, the section `a(2:1)`), the section has zero-size.

The following example shows subscript triplet notation assigning the same value to a regular pattern of array elements.

```
INTEGER, DIMENSION(3,6) :: x,y,z   ! declare 3 3x6 arrays

! initialize the arrays, using whole-array assignments.
x = 0; y = 0; z = 0

! assign to elements of x, y, and z, using subscript triplets
x(3,2:4:1)  = 1
y(2,2:6:2)  = 2
z(1:2,3:6)  = 3

! The arrays x, y, and z now have the following values:
!       x               y                z
!  0 0 0 0 0 0     0 0 0 0 0 0      0 0 3 3 3 3
!  0 0 0 0 0 0     0 2 0 2 0 2      0 0 3 3 3 3
!  0 1 1 1 0 0     0 0 0 0 0 0      0 0 0 0 0 0
```

In the following example of an array substring, the variable `dates(5:10)` is an array section that includes elements 5 through to 10 of the parent array dates, and the variable `dates(5:10)(8:11)` is also an array section of the array dates but only contains the last 4 character positions of the elements 5 through to 10.

```
CHARACTER(11) :: dates(20)
dates(5:10)(8:11) = "1776"
```

Vector subscripts

A **vector subscript** is any expression that results in a rank-one array with integer value. The values of the array select the corresponding elements of the parent array for a given dimension. Vector subscripts can describe an irregular pattern and may be useful for indirect array addressing. For example, if `v` represents a rank-one integer array initialized with the values 4, 3, 1, 7, then the array section `a(v)` is a rank-one array composed of the array elements `a(4)`, `a(3)`, `a(1)`, and `a(7)`—in that order.

Vector subscripts are commonly specified using array constructors, which are described in the next section. For example, the expressions `a(v)` and `a((/ 4, 3, 1, 7/))` reference the same section of the array `a`.

Vector subscripts may not appear:

- On the right hand side of a pointer assignment statement.
- In an I/O statement as an internal file.
- As an actual argument that is associated with a dummy argument declared with `INTENT(OUT)` or `INTENT(INOUT)` or with no `INTENT`.

A vector subscript may specify the same element more than once. When a vector subscript of this form specifies an array section, the array section is known as a *many-one array section*. An example of a *many-one array section* is:

```
a( (/ 4, 3, 4, 7/) )
```

where element 4 has been selected twice. A many-one array section may not appear in either an input list or on the left-hand side of an assignment statement.

The following example, vector_sub.f90, illustrates an array section using a section subscript list.

vector_sub.f90

```
PROGRAM main
  ! m is a rank-1 array that has been
  ! initialized with the values of an array constructor

  INTEGER, DIMENSION(4) :: m = (/ 2, 3, 8, 1/)

  INTEGER :: i

  ! initialize a (a rank-1 array) with the values
  ! 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 11.0
  REAL, DIMENSION(10) :: a = (/ (i*1.1, i=1,10) /)

  ! b is an uninitialized 4x2 array
  REAL, DIMENSION(4,2) :: b

  ! print a section of a, using a vector subscipt
  PRINT *,a(m)

  ! assign the values 5.5, 11.0, 6.6, and 5.5 to the first column
  ! b; this is an example of a many-one array
  b(:,1) = a( (/ 5, 10, 6, 5/) )

  ! the vector subscript MIN(m,4) represents a rank-1 array with
  ! the values 2, 3, 4, 1; the second column of b is assigned
  ! the values 11.0, 6.6, 5.5, 5.5
  b(:,2) = b(MIN(m,4),1)

  ! increment a(2), a(3), a(8), and a(1) by 20.0
```

```
  a(m) = a(m) + 20.0

  ! print the new values in a
  PRINT *,a
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 vector_sub.f90
$ a.out
 2.2 3.3 8.8 1.1
 21.1 22.2 23.3 4.4 5.5 6.6 7.7 28.8 9.9 11.0
```

Array-valued structure component references

A structure component reference can specify an array or a scalar. If, for example, the parent in the reference is declared as an array and likewise one of the components is declared as an array, this makes possible an *array-valued structure component reference*. Conceptually, an array-valued structure component reference is similar to a reference to an array section (see “Array sections” on page 63).

Consider the following code:

```
TYPE student_data
  CHARACTER(25) :: name
  INTEGER       :: average, test(4)
END TYPE student_data

TYPE course_data
  CHARACTER(25) :: course_title
  INTEGER       :: course_num, class_size
  TYPE(student_data) :: student(10)
END TYPE course_data
TYPE (course_data) :: course(5)
```

These statements prepare a database for maintaining course information for 50 students—10 students per course. The information about the students is held in `student`—an array of derived type. Likewise, the information about the five courses is held in `course`, which is also an array of derived type and which has `student` as one of its components. The following statement assigns a test score to a one student in one course, using a structure component reference:

```
course(5)%student(7)%test(4) = 95
```

The reference is scalar-valued: `95` is assigned to a single element, `test(4)` of `student(7)` of `course(5)`.

However, it is also possible to reference more than one element in a structure component reference. The following statement assigns the same score to one test taken by all students in one course:

```
course(4)%student%test(3) = 60
```

The structure component reference is array-valued because thirty elements are assigned with the one reference. The reference is to a section of the array `course`, rather than to the entire array.

The next statement also makes an array-valued structure component reference to initialize all the tests of one student in one course:

```
course(3)%student(3)%test = 0
```

The next statement uses a subscript triplet in an array-valued structure component reference to assign the same score to one test of three students in one course:

```
course(2)%student(1:3)%test(4) = 82
```

It would be convenient if we could initialize all tests of all students in all courses to 0. But the Standard does not allow structure component references in which more than one of the parts specifies a rank greater than 0. In other words, the following is not legal:

```
course%student%test = 0  ! ILLEGAL
```

The following example, array_val_ref.f90, contains the code examples listed in this section:

array_val_ref.f90

```
PROGRAM main
! illustrates array-valued structure component references

  ! define a derived type that will be used to declare an
  !   object of this type as a component of another derived type
  TYPE student_data
    CHARACTER(25) :: name
    INTEGER       :: average, test(4)
  END TYPE student_data

  TYPE course_data
    CHARACTER(25) :: course_title
    INTEGER       :: course_num, class_size
    TYPE(student_data) :: student(10)   ! an array of derived
  ! type
  END TYPE course_data

  TYPE (course_data) :: course(5)       ! an array of derived
  ! type

  ! scalar-valued structure component reference
  course(5)%student(7)%test(4) = 95
  PRINT *, course(5)%student(7)%test(4)

  ! array-valued structure component reference
  course(4)%student%test(3) = 60
  PRINT *, course(4)%student%test(3)

  ! array-valued structure component reference
  course(3)%student(3)%test = 0
  PRINT *, course(3)%student(3)%test
```

```
  ! array-valued structure component reference, using
  ! a subscript triplet to reference a section of the
  ! array component student
  course(2)%student(1:3)%test(4) = 82
  PRINT *, course(2)%student(1:3)%test(4)

  ! the following commented-out statement is illegal:
  ! only one part (of the combined components and
  ! parent) in a structure component reference
  ! may have a rank greater than 0.
  ! course%student%test = 0

END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 array_val_ref.f90
$ a.out
 95
 60 60 60 60 60 60 60 60 60 60
 0 0 0 0
 82 82 82
```

Array constructors

An **array constructor** is used to assign values to an array. The generated values are supplied from a list of scalar values, arrays of any rank, and implied `DO` specifications. An array constructor may appear in any context in which a rank-one array expression is allowed. An array with a rank greater than one may be constructed by using the `RESHAPE` intrinsic function. The type of an array constructor is taken from the values in the list, which must all have the same type and type parameters (including character length). The extent is taken from the number of values specified.

The syntax of an array constructor is:

`(/` *ac-value-list* `/)`

where *ac-value-list* is a comma-separated list of one or more *ac-values*. Each *ac-value* may be any of the following:

- Scalar expressions, for example:

  ```
  (/ 1.2, 0.0, 2.3 /)
  ```

- An array expression, for example:

  ```
  (/ x(0:5) /)
  ```

  where the values in `x(0)` through `x(5)` become the values of the array constructor. If the array the value list has a rank greater than one, the values are generated in column-major order, as explained in "Array fundamentals" on page 52.

- An implied-`DO` specification, taking the form:

  `(`*ac-value-list*`,` *do-var* `=` *expr1*`,` *expr2* `[,` *expr3*`])`

  where *do-var* is the name of a scalar integer variable, *expr1* is the initial value, *expr2* is the final value, and *expr2* is the stride (the default is 1). For example:

  ```
  (/ i, i = 1, 10 )
  ```

When used to initialize an array in a type declaration or in an assignment statement, all elements in the array must be initialized. For example, the following is illegal:

```
INTEGER :: i(10) = (/ 1, 2, 3 /)  ! ILLEGAL:  too few
                                  !   initializers
```

If no values are supplied, the array constructor is zero-sized. For example, the size of the following array constructor:

```
(/ ( i, i=10,n) /)
```

depends on the value of the variable n; if the value of the variable is less than 10, then the constructor contains no values.

If the list contains only constant values, the array constructor may initialize a named constant or a type declaration statement. An array constructor may not initialize variables in a DATA statement, which may contain scalar constants only.

As an extension, HP Fortran 90 allows the use of [and] in place of (/ and /).

The following are examples of array constructors:

```
! array x is assigned three real values.
x = (/19.3, 24.1, 28.6/)

! One vector, consisting of 16 integer values, is assigned to j
j = (/4, 10, k(1:5), 2 + l, (m(n), n = -7,-2),16, 1/)

! assign 5 values
a = (/(base(k), k=1,5)/)

! The named constant t is a rank-one array initialized with
!   the values 36.0 and 37.0
REAL,DIMENSION(2):: t
PARAMETER (t=(/ 36.0, 37.0/))

! the array constructor is reshaped as  1 3 5 7
!                                       2 4 6 8
! and is then assigned to z
z=RESHAPE((/1,2,3,4,5,6,7,8/), (/2,4/) )

! an array constructor is used for the second component of
! the structure constructor
alaska = site(”NOME”,(/-63,4/))

diagonal = (/ (b(i,i), i=1,n) /)
hilbert = RESHAPE( (/ ((1.0/(i+j), i=1,n), j=1,n) /), (/ n,n /) )
ident = RESHAPE ( (/ (1, (0, i=1,n), j=1,n-1), 1 /), (/ n,n /) )
```

As shown in last three examples, an array constructor with implied-DO loops and the RESHAPE function permit construction of arrays that cannot otherwise be expressed conveniently with alternative notations.

Array expressions

Array operations areperformed in parallel. That is, an operation is performed on each element independently and in any order. The practical effect of this is that, because an assignment statement may have the same array on both the left and right-hand sides, the right-hand side is fully evaluated before any assignment takes place. This means that in some cases the compiler may create temporary space to hold intermediate results of the computation.

A scalar may appear in an array expression. If the scalar is used in an expression containing whole array references—for example

```
a = b + 2.0   ! a and b are conformable arrays of type real
```

then the effect is as if the scalar were evaluated and then broadcast to form a conformable array of elements, each having the value of the scalar. Thus, a scalar used in an array context is conformable with the array or arrays involved.

Zero-sized arrays may also appear in an array expression. Although they have no elements, they do have a shape and must therefore follow the rule of conformable arrays. Because scalars are conformable with any array, they may therefore appear in an operation involving a zero-sized array.

The following illustrates valid and invalid array expressions.

```
SUBROUTINE foo(a,b,c)

! a is an assumed-shape array with rank-one
REAL :: a(:)

! b is a pointer to a rank-two array
REAL, POINTER :: b(:,:)

! c is an assumed-size array
REAL :: c(*)

! d is an allocatable array; its shape can only be defined in an
! ALLOCATE statement
REAL, ALLOCATABLE :: d(:)

! create the array d with the same size as a; a and d have
! the same shape and are therefore conformable
ALLOCATE(d(SIZE(a)))

! copy the array a into d
d = a
```

```
! sets each element of the array associated with b to 0.0;
! the effect is as if the scalar were broadcast into a
! temporary array, with the same shape as b; b is then assigned
! to theleft-hand side
b = 0.0

! corresponding elements of a and d are added together and then
! stored back into the corresponding array element of d
d = a + d

! conceptually the operand SQRT(d) is evaluated into an
! intermediate array with the same shape as d; each element of
! the intermediate array will be added to the corresponding
! element of a and stored into the corresponding element of d
d = a + SQRT(d)

DEALLOCATE(d)

! examples of illegal uses of arrays:

! ILLEGAL - c is an assumed-size array and so has no shape;
! an assumed-size array may not be used as a whole array
! operand(except in an  argument list)
a = c

! ILLEGAL - the arrays a and b do not have the same shape and are
! therefore not conformable
a = a + b

! ILLEGAL - d was previously deallocated and must not be
! referenced subsequently
a = a + d

END SUBROUTINE foo
```

Array-valued functions

A function may be array-valued; that is, its return value may evaluate to an array of values rather than to a scalar. Array-valued functions may appear in any array expression except:

- In an input list
- On the left side of an assignment statement (unless returning the result from within a function)

Array-valued functions may also be used in an array expression wherever a scalar function reference is allowed but must be conformable—that is, the function result must have the same shape as the expression.

The following sections describe intrinsic functions and user-defined functions that are array-valued.

Intrinsic functions

Elemental procedures and transformation procedures have particular relevance to array expressions. Elemental procedures—for example, `SQRT` and `SIN`—are specified for scalar arguments, but with an array argument they return an array-valued result with the same shape as the argument. Each element of the result is as if the function were applied to each corresponding element of the argument.

A transformational procedure—for example, `RESHAPE`, `SUM`, and `MATMUL`—generally has one or more array arguments that the procedure operates on as a whole, and usually returns an array-valued result whose elements may depend not only on the corresponding elements of the arguments but also on the values of other elements of the arguments.

User-defined functions

User-defined functions can return either a scalar-valued result or an array-valued result. A scalar function can appear in an array expression; its effect is to broadcast its value throughout a conformable array. A reference to a user-defined array-valued function must obey the rules for functions in general, and must also conform to the shape of the expression in which it appears.

User-defined functions are described in “External procedures” on page 128.

The following code segment illustrates two array-valued functions, `genrand` (user-defined) and `RESHAPE` (intrinsic):

```
PROGRAM main

! The following interface block describes the characteristics of
! the function genrand; the function inputs a single integer
! scalar and returns a real array of rank-one with an extent
! equal to the value of its argument
INTERFACE
    FUNCTION genrand(n)
    INTEGER:: n
    REAL, DIMENSION (n)::genrand
    END FUNCTION genrand
END INTERFACE

REAL :: a(100)
REAL :: b(10,10)

! set array a to the result returned by the function genrand;
! note that the left and right hand side are conformable
a = genrand(SIZE(a))

! add each element of a to the corresponding element of the
! result returned by genrand, forming an intermediate rank-one
! result that is passed into the intrinsic function RESHAPE.
! This intrinsic transforms its argument into a 10 by 10 array.
! Again, the left and right hand side are conformable.
b = RESHAPE(a  + genrand(100),(/ 10, 10 /))
  .
  .
  .
END PROGRAM main
```

Array inquiry intrinsics

Table 9 lists and briefly describes the inquiry intrinsic functions that return the properties of an array. For a full description of these intrinsics, see Chapter 11, “Intrinsic procedures,” on page 475.

Table 9 **Array inquiry intrinsic functions**

Intrinsic	Description
ALLOCATED	Returns the allocation status of an allocatable array; see “Allocatable arrays” on page 59.
ASSOCIATED	Returns the association status of an array pointer; see “Pointer association status” on page 48.
LBOUND	Returns either the lower bound of a specified dimension or the lower bounds of the array as a whole.
SHAPE	Returns the shape of the array as a rank-one integer array.
SIZE	Returns the size of the array or the extent of a particular dimension.
UBOUND	Returns the upper bound of a specified dimension or the upper bounds of the array as a whole.

5 Expressions and assignment

This chapter describes expressions and assignment. More specifically, it covers the following topics:

- Expressions, including their components:
  - Operands
  - Operators
  - Special forms of expression
- Assignment, including the following topics:
  - Assignment statement
  - Pointer assignment
  - Masked array assignment

NOTE This chapter discusses intrinsic operators and assignment only. For information about user-defined operators and assignment, see “Defined operators” on page 155 and “Defined assignment” on page 157.

Expressions

An expression is the specification of data and, possibly, a set of operations that enable the computer to evaluate the expression and produce a value. Because an expression results in a value, it has a type, kind, and shape. If an expression is of the character type, it also has a length parameter.

The general form of an expression is:

[*operand1*] *operator* *operand2*

operand1, *operand2*
: are data objects or expressions that evaluate to data. They may be array-valued or scalar-valued.

operator
: is either an intrinsic or defined operator. If *operator* is unary, *operand1* must not be specified.

The following sections describe operands, operators, and expressions in more detail.

Operands

An operand may be any of the following:

- A constant or a variable, such as `1.0`, `'ab'`, or `a`
- An array element or an array section, such as `a(1,3)` or `a(1,2:3)`
- A character substring or a structure component, such as `ch(1:3)` or `employee%name`
- An array constructor, such as `(/1.0,2.0/)`
- A structure constructor, such as `employee(8, "Wilson", 123876)`
- A function reference, such as `SQRT(x)`
- An expression in parentheses, such as `(b + SIN(y)**2)`

Any variable or function reference used as an operand in an expression must have been previously defined. Likewise, any pointer must have been previously associated with a target. If an operand has the `POINTER` attribute, the target associated with it is the operand.

When an operand is a whole array reference, the complete array is referenced. An assumed-size array variable cannot be an operand. An array section of an assumed-size array can be an operand if the extent of the last dimension of the section is defined by the use of a subscript, a section subscript with an extent for the upper bound, or a vector subscript. (Assumed-size arrays are discussed in “Assumed-size arrays” on page 61, and array sections in “Array sections” on page 63.)

If two operands in an expression are arrays, they must have the same shape. If one operand is a scalar, it is treated as if it were an array of the same shape as the other operand, in which all elements have the value of the scalar. The result of the operation is an array in which each element is the result of applying the operator repeatedly to corresponding elements of the two operands.

The rules governing how the use of operands in an expression vary, depending on the type of expression. For example, some operands that may appear on the right-hand side of an assignment statement but not in an initialization expression. See “Special forms of expression” on page 89 for detailed information about the different forms of an expression and the restrictions that those forms impose on operands.

Operators

HP Fortran 90 recognizes the following types of operators:

- Arithmetic operators
- Relational operators
- Concatenation operator
- Logical operators
- Bitwise operators
- Defined operators

All of these except the last are intrinsic operators—that is, the operations they perform are defined by HP Fortran 90. Intrinsic operators are described in the following sections. Defined operators are those that the programmer defines—or *overloads*, if the operator already has already been defined—using the `INTERFACE` statement. Defined operators and overloading are discussed in “Defined operators” on page 155.

Arithmetic operators

The arithmetic operators are:

- Additive operators (`+` and `-`). These can be used either as unary operators or binary operators.
- Multiplicative operators (`/`, `*`, and `**`). These are binary.

Two operands joined by a binary operator can be of different numeric types or different kind type parameters. The type of the result is:

- If the type and kind type parameters of the operands are the same, the result has the same type and kind type parameter.
- If the type of the operands is the same but the kind type parameters differ, the result has the same type and the larger kind type parameter.
- If either operand is of type complex, the result is of type complex.
- If either operand is of type real and the other operand is not of type complex, the result is of type real.

Except for a value raised to an integer power, each operand that differs in type or kind type parameter from that of the result is converted to a value with the type and kind type of the result before the operation is performed.

Logical and integer operands can be combined with arithmetic operators. The logical operand is treated as an integer of the same kind type parameter, and the result of the operation is of type integer. If the operands have different kind type parameters, the shorter is considered to be extended as a signed integer. For information about logical values, see “Logical operators” on page 84.

The arithmetic operators behave as expected, with the following qualifications:

- The division of an integer by an integer is defined to be the integer closest to the true result that is between zero and the true result.
- Exponentiation of an integer to a negative integer—`i1**i2`, where `i2` is negative—is interpreted as `1/(i1**(-i2))`, where the division is interpreted as described for division of one integer by another.
- If `x1` and `x2` are real and `x1` is negative, then `x1**x2` could be an invalid expression, as the result could be complex. Note, however, that `CMPLX(x1)**x2` is valid; the result is the principal value.

The following are HP extensions to the Fortran 90 Standard:

- The exponentiation operator may be followed by a signed entity, as in the following example:

  ```
  i ** -j
  ```

  The Fortran 90 Standard does not allow adjacent operators.

- Operands of logical and integer types may be combined with the arithmetic operators. The logical variable is treated as an integer of equivalent size, and the result of the operation is an integer value. When different lengths of operands are involved, the shorter is considered extended as a signed integer. The following is an example:

  ```
  LOGICAL(1) :: boolean1 = -4
  LOGICAL(4) :: boolean4 = 2**16 + 27
  INTEGER(1) :: flag1
  INTEGER(4) :: flag4

  flag4 = boolean4 - boolean1    !set flag4 to 2**16 + 31

  ! a relational operator with a logical operand
  IF (boolean4 > 65536) THEN
      flag1 = -(boolean4/65536)  !set flag1 to -1
  ENDIF
  ```

Relational operators

The relational operators are `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`, `==`, `/=`, `>`, `>=`, `<`, and `<=`. All relational operators are binary. The letter forms of the relational operators have the same meaning as the symbol forms. Thus, `.EQ.` is a synonym for `==`, `.NE.` is a synonym for `/=`, and so on.

If the operands in a relational operation are numerical expressions with different type or kind type parameters, the operands are converted to the type and kind type parameters that the sum of the operands would have and are then compared; see "Arithmetic operators" on page 82 for information about the result of mixed arithmetic expressions.

If the operands are character expressions, the shorter operand is blank-padded to the length of the other prior to the comparison. The comparison starts at the first character and proceeds until a character differs or equality is confirmed. See Appendix C for the collating sequence.

Concatenation operator

The concatenation operator is `//`. It is binary.

In a concatenation operation, each operand of the concatenation operator must be of type character and have the same kind type parameter. The character length parameter of the result is the sum of the character length parameters of the operands.

Logical operators

The logical operator are `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`, `.XOR.`, and `.NOT.`. The `.NOT.` operator is unary; the others are binary. The `.XOR.` is an HP extension having the same meaning as the `.NEQV.` operator.

As an HP extension, the operands of a logical expression may be of type integer. Functions returning integers may appear in logical expressions, and functions returning logicals may appear in integer expressions.

If the operands of a logical operation have different kind type parameters, the operand with the smaller parameter is converted to a value with the larger parameter before the operation is performed. The result has the larger kind type parameter.

Table 10 shows the behavior of the logical operators for the different permutations of operand values. Note that the `.XOR.` operator is a synonym for the `.NEQV.` operator and behaves similarly.

Table 10 Logical operators

opnd1	*opnd2*	.AND.	.OR.	.EQV.	.NEQV.	.NOT. *opnd1*
.TRUE.	.TRUE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.

Bitwise operators

As an extension to the Standard, HP Fortran 90 allows logical operators to be used as bitwise operators on integer operands. The logical operations are bitwise; that is, they are performed for each bit of the binary representations of the integers. When the operands are of different lengths, the shorter is considered to be extended to the length of the other operand as if it were a signed integer, and the result has the length of the longer operand.

When logical operators are used on integer operands, any nonzero value is considered `.TRUE.`, and a zero value is considered `.FALSE.` .

In general, an actual argument of type integer may not be used in a reference to a procedure when the corresponding dummy argument is of type logical, nor may an actual argument of type logical be used when the dummy argument is of type integer. As an HP extension, logical and integer arguments may be used interchangeably in calls to bit manipulation intrinsics. See Chapter 11, “Intrinsic procedures,” on page 475 for information about the bit manipulation intrinsics.

The following example shows the use of the `.AND.` operator to perform a bitwise AND operation:

```
INTEGER i, j

i = 5
j = 3
PRINT *, i .AND. j

! Output from the PRINT statement:  1
```

The next example shows the use of logical operators to perform bit-masking operations.

```
INTEGER(2) mask2
INTEGER(4) mask4
DATA mask2/ -4 /
DATA mask4/Z"ccc2"/

mask4 = mask4 .NEQV. mask2      !set mask4 to Z"ffff333e"

mask2 = .NOT. mask4             !set mask2 to Z"ccc1"
```

The next example makes a standard-conforming reference to a bit manipulation intrinsic:

```
INTEGER :: mask = 65535
LOGICAL :: is_even = .FALSE.
IF (IAND(mask,1) /= 0) is_even = .TRUE.
```

HP Fortran 90 allows the following nonstandard version of the preceding example:

```
LOGICAL :: mask = z"ffff"

INTEGER :: is_even = .FALSE.
IF (IAND(mask,1)) is_even = .TRUE.
```

Operator precedence

When an expression expands to

operand1 operator1 operand2 operator2 operand3 ...

each operator is assigned a precedence. The defined order of evaluation is that any subexpressions containing an operator with higher precedence than the adjacent operators is evaluated first. Where operators are of equal precedence, evaluation is from left to right. The exception to this rule is the exponentiation operator (**), which is evaluated from right to left.

Any expression or subexpression may be enclosed in parentheses. These expressions are always evaluated first, using the rules explained above. This usage of parentheses is therefore equivalent to normal mathematical usage.

Table 11 lists the precedence of the operators, and Table 12 gives example expressions that illustrate operator precedence.

Table 11 **Operator precedence**

Precedence	Operators
Highest	User defined unary operators
.	`**`
.	`* /`
.	Unary + Unary -
.	`+ -`
.	`//`
.	`.EQ. .NE. .LT. .LE. .GT. .GE.` `== /= < <= > >=`
.	`.NOT.`
.	`.AND.`
.	`.OR.`
.	`.EQV. .NEQV. .XOR.`
Lowest	User-defined binary operators

Table 12 **Examples of operator precedence**

Expression	How evaluated	Explanation
`a+b*c`	`a + (b*c)`	* has a higher precedence than +.
`a/b*c`	`(a/b)*c`	/ and * have the same precedence, and evaluation is left to right.
`a**b**c`	`a**(b**c)`	** evaluates right to left.
`a.AND.b.AND .c.OR.d`	`((a.AND.b).AND.c) .OR.d)`	Logical operators evaluate left to right.

The Standard allows the compiler to generate code that evaluates an expression by any sequence that produces a result mathematically equivalent to the sequence implied by the statement. This laxity permits code optimization, including (for example) the reordering of expressions and the promotion of common subexpressions.

Because the order of evaluation is not defined by the Standard, a function reference within an expression may not modify any of the other operands within the same expression. For example, `fun(x)+x` is indeterminate if the reference to `fun` modifies the value of the argument `x`.

Special forms of expression

Certain language constructs allow only restricted forms of expressions. For example, the value specified for a named constant in a `PARAMETER` statement may be defined by an expression, but it must be possible to evaluate the expression at compile-time. This means that the expression must not contain any operands that depend on program execution for their value. To take another example, a bound of a dummy array argument may be specified as an expression, but it must be possible to evaluate this expression on entry to the subprogram.

There are special restrictions imposed on operands and operators that may appear in an expression, depending on whether the expression is one of the following:

- Constant expressions
- Initialization expressions
- Specification expressions

The following sections describe the special forms of expression.

Constant expressions

A **constant expression** is either a constant or an expression containing only intrinsic operators and constant operands. This restriction also applies to any clearly defined part of a constant—for example, a substring with constant start and end points, or an array or structure constructor. A constant expression may include references to intrinsic functions that can be evaluated at compile-time. A constant expression may appear in any context in which any expression may appear.

The following are examples of constant expressions:

```
123                    ! an integer literal

"Hello " // " World"   ! a character constant expression

3.0_single             ! a real literal constant where single is
                       ! a named integer constant

coord(0.0,infinity)    ! a structure constructor in which
                       ! "infinity" is a named constant

(/ SQRT(x), x, x*x /)  ! an array constructor in which x is a
                       ! named real constant

x*x + 2*x*y + y*y      ! a constant numeric expression where x
                       ! and y are named constants

SUM(iterations,DIM=1)  ! reference to a transformational
                       !  intrinsic where iterations is an
                       !  array-valued named constant

SHAPE(matrix)          ! a reference to an inquiry intrinsic in
                       ! which "matrix" is an array with
                        ! constant bounds
```

Initialization expressions

An **initialization expression** is a more specialized form of constant expression that can appear as the initial value in a declaration statement. Initialization expressions have these additional restrictions:

- Exponentiation is only allowed if the second operand is an integer.
- Any subexpression within the expression must itself be an initialization expression.
- All arguments to intrinsic function references must be initialization expressions.
- Only the following transformational intrinsic functions may be referenced:
  - REPEAT
  - RESHAPE
  - SELECTED_INT_KIND
  - SELECTED_REAL_KIND
  - TRANSFER

  - TRIM

- Any inquiry intrinsic that is referenced may interrogate a property of an entity (such as bounds or kind type parameter) only if the property is a constant.
- Any elemental intrinsic functions must have integer or character arguments and an integer or character result.

Initialization expressions are required for the following:

- Values of named constants. Any entity declared with the `PARAMETER` attribute must be initialized with an initialization expression.
- Kind parameter in a type specification statement.
- The `KIND` dummy argument of a type conversion intrinsic function.
- Initial values in type declaration statements.
- Expressions in structure constructors in `DATA` statements.
- Case values in `CASE` statements.
- Subscript expressions or substring ranges in `EQUIVALENCE` statements.

The following entities may not be initialized:

- Dummy arguments
- Function results
- Allocatable arrays
- Pointers
- External names
- Intrinsic names
- Automatic objects

The following are examples of initialization expressions:

```
-456                   ! an integer literal

("Hello "// "World")   ! a character constant expression

pi * r ** 2            ! a constant numeric expression, where
                       !   pi and r are named constants

ABS(i * j)             ! reference to an elemental intrinsic,
                       !  where i and j are named integer
                       !  constants

SELECTED_REAL_KIND(7)   ! reference to a transformational
intrinsic
```

The following are illegal initialization expressions:

```
x ** 2.5               ! the power operand is not an integer

LOG(10.0)              ! the intrinsic function is neither
                       !  integer nor character type

SUM( (/ i, 2 /) )       ! reference to a prohibited function
```

For information about initializing arrays with an array constructor, see “Array constructors” on page 71.

Specification expressions

A **specification expression** has a scalar value, is of type integer, and can be evaluated on entry to the scoping unit in which it appears. A specification expression may appear (for example) as a bound in an array declaration or as the length in a CHARACTER type declaration.

An operand in a specification expression is one of the following:

- A literal or named constant or part of a constants.
- A variable that is available by argument, host, or use association or is in common.
- An array constructor or structure constructor where each element or component is also a specification expression or is a variable in an implied-DO loop appearing in the array constructor.
- A dummy argument having neither the OPTIONAL attribute nor the INTENT(OUT) attribute.
- An argument to an intrinsic function.

- A reference to an elemental intrinsic function that returns an integer result.
- A reference to any of the following transformational intrinsic functions:
  - REPEAT
  - RESHAPE
  - SELECTED_INT_KIND
  - SELECTED_REAL_KIND
  - TRANSFER
  - TRIM
- Any inquiry intrinsic except ALLOCATED, ASSOCIATED, and PRESENT. Other inquiry intrinsics may be referenced so long as the property interrogated is not defined by either a pointer assignment or ALLOCATE statement. Furthermore, an inquiry intrinsic may not interrogate the following properties of an assumed size array:
  - Upper bound of the last dimension
  - Extent of the last dimension
  - Size of the array
  - Shape of the array

The differences between specification expressions and initialization expressions are summarized in Table 13.

Table 13 Initialization and specification expressions

Initialization expression	Specification expression
Can be either scalar or array-valued.	Must be scalar-valued.
Can be of any type.	Must be of type integer.
Must be a constant expression.	Can reference variables by host, argument, or use storage association; can reference variables in common.
Except for ALLOCATED, ASSOCIATED, and PRESENT, can reference inquiry intrinsics to interrogate a property of an entity, provided that the property is constant.	Can reference inquiry intrinsic functions, except for ALLOCATED, ASSOCIATED, and PRESENT. The arguments must be specification expressions or variables whose bounds or type parameters inquired about are not assumed, are not defined by the ALLOCATE statement, or are not defined by pointer assignment.

The following are examples of specification expressions:

```
789                    ! an integer literal constant

MAX(m+n,0)             ! m and n are integer dummy arguments

LEN(c)                 ! c is a character variable accessible via
                       ! host association

SELECTED_INT_KIND(5)   ! reference to a transformational
                       ! intrinsic

UBOUND(arr,DIM=n)      ! reference to an array inquiry
                       ! intrinsic in which arr is an array
                       ! accessible via USE and n is a
                        ! variable in common
```

Assignment

An assignment operation defines a variable by giving it a value. In HP Fortran 90, there are four types of assignment:

- Intrinsic assignment (also known as the *assignment statement*)
- Pointer assignment
- Masked-array assignment (also known as the `WHERE` construct)
- Defined assignment

The following sections describe the first three assignment types. The last—defined assignment—is defined by the programmer, using the `INTERFACE` statement. For information about defined assignment, see “Defined assignment” on page 157.

Assignment statement

An assignment statement gives the value of an expression to a variable. It has the following syntax:

variable = *expression*

variable may be any nonpointer variable or a pointer variable that is associated with a target. (If *variable* is a pointer, *expression* is assigned to the target.) The valid combinations of types for *variable* and *expression* are given in Table 14. The intrinsic functions that document the conversions are described in Chapter 11.

Table 14 **Conversion of *variable*=*expression***

Variable type	Expression type	Conversion
Integer	Integer, real, or complex	`INT(`*expression*`, KIND(`*variable*`))`
Real	Integer, real, or complex	`REAL(`*expression*`, KIND(`*variable*`))`
Character	Character (same kind parameters)	`CMPLX(`*expression*`, KIND(`*variable*`))`

Variable type	Expression type	Conversion
Logical	Logical	Truncate *expression* if its length is greater than that of *variable*; otherwise, pad value assigned to *variable*, with blanks.
Logical	Logical	`LOGICAL(`*expression*`, KIND(`*variable*`))`
Derived type	Same derived type	None

As described in “Bitwise operators” on page 85, HP Fortran 90 allows integer and logical operands to be used interchangeably. HP Fortran 90 also allows logical expressions to be assigned to integer variables and integer expressions to logical variables. As shown in Table 14, a logical expression may also be assigned to real or complex variables, and similarly, a real or complex expression may be assigned to a logical variable.

If *variable* is a scalar, *expression* must be scalar. If *variable* is an array or an array section, *expression* must be either an array-valued expression of the same shape or a scalar. If *variable* is an array or an array section, and *expression* is a scalar, the value of *expression* is assigned to all elements of *variable*. If *variable* and *expression* are both arrays, the assignment is carried out element by element with no implied ordering.

The *expression* is evaluated completely before the assignment is started. For example, the following code segment:

```
CHARACTER (LEN=4):: c
c(1:4) = 'abcd'
c(2:4) = c(1:3)
```

sets `c(2:4)` to `"abc"`, not to `"aaa"`, which might result from a left-to-right character-by-character assignment.

The following examples illustrate assignments of different data types:

```
! declarations of the variables used in the assignment statements
!   to follow
integer icnt
type circle
   real radius
   real xreal y
end type
type (circle) circle1, circle2
real area, pi
logical boolx, booly, pixel(10,10)
integer a(10,5)
integer, dimension (10,10):: matrix1, matrix2
character*3 initials
character*10 surname
character*20 name

icnt = icnt + 1    ! integer assignment

circle1 = circle2  ! derived-type assignment

area = pi * circle%radius**2    ! real assignment

pixel(x,y) = boolx .AND. booly  ! assigns a logical expression to
                                ! an element of the logical
                                ! array pixel

a(:,1:2) = 0  ! first two columns of a are set to zero

maxtrix1 = maxtrix2  ! each element of maxtrix2 is assigned to
                     ! the corresponding element of maxtrix1

name = initials // surname  ! character assignment using the
                            ! concatenation operator
```

Pointer assignment

Pointer assignment establishes an association between a pointer and a target. Once the association is established, if the pointer is referenced on the left-hand side of an assignment statement, it is the target to which the assignment is made. And if the pointer is referenced in an expression, the target is taken as the operand in the expression.

The syntax of a pointer assignment is:

pointer-object => *target-expression*

pointer-object

is a variable with the `POINTER` attribute.

target-expression

is one of the following:

- A variable with the TARGET or POINTER attribute
- A function reference or defined operation that returns a pointer result

The type, kind, and rank of *pointer-object* and *target-expression* must be the same. If *target-expression* is an array, it cannot be an assumed-size array or an array section with a vector subscript. For information about assumed-size arrays, see “Assumed-size arrays” on page 61. For information about array sections with vector subscripts, see “Vector subscripts” on page 65.

If *target-expression* is a pointer already associated with a target, then *pointer-object* becomes associated with the target of *target-expression*. If *target-expression* is a pointer that is disassociated or undefined, then *pointer-object* inherits the disassociated or undefined status of *target-expression*. For information about pointer status, see “Pointer association status” on page 48.

The following example, ptr_assign.f90, illustrates association of scalar and array pointers with scalar and array targets:

ptr_assign.f90

```
PROGRAM main
  INTEGER, POINTER :: p1, p2, p3(:)  ! declare three pointers, p3
                                     ! is a deferred-shape array
  INTEGER, TARGET :: t1 = 99, t2(5) = (/ 1, 2, 3, 4, 5 /)

  ! p1, p2 and p3 are currently undefined.
  p1 => t1       ! p1 is associated with t1.
  PRINT *, 'contents of t1 referenced through p1:', p1

  p2 => p1       ! p2 is associated with t1.
                 ! p1 remains associated with t1.
  PRINT *, 'contents of t1 referenced through p1 through p2:', p2

  p1 => t2(1)    ! p1 is associated with t2(1).
              ! p2 remains associated with t1.

  PRINT *, 'contents of t2(1) referenced through p1:', p1

  p3 => t2       ! p3 is associated with t2.
  PRINT *, &
    'contents of t2 referenced through the array pointer p3:', p3

  p1 => p3(2)    ! p1 is associated with t2(2).
  PRINT *, &
```

```
    'contents of t2(2) referenced through p3 through p1:', p1

  NULLIFY(p1)    ! p1 is disassociated.
  IF (.NOT. ASSOCIATED(p1)) PRINT *, "p1 is disassociated."

  p2 => p1       ! Now p2 is also disassociated.
  IF (.NOT. ASSOCIATED(p2)) PRINT *,  &
    "p2 is disassociated by pointer assignment."
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 ptr_assign.f90
$ a.out
 contents of t1 referenced through p1: 99
 contents of t1 referenced through p1 through p2: 99
 contents of t2(1) referenced through p1: 1
 contents of t2 referenced through the array pointer p3: 1 2 3
 4 5
 contents of t2(2) referenced through p3 through p1: 2
 p1 is disassociated.
 p2 is disassociated by pointer assignment.
```

Masked array assignment

In a masked array assignment, a logical expression—called a *mask*—controls the selection of array elements for assignment. Masked array assignment is implemented by the `WHERE` statement and the `WHERE` construct. The syntax of the `WHERE` statement is:

`WHERE` (*array-logical-expression*) *array* = *array-expression*

where *array-logical-expression*, *array*, and *array-expression* must all be conformable. The *array-logical-expression* (the mask) is evaluated for each element and the outcome (`.TRUE.` or `.FALSE.`) determines whether an assignment is made to the corresponding element of *array*.

The syntax of the `WHERE` construct is:

```
WHERE ( array-logical-expression )
       array = array-expression
      [array = array-expression] ...
[ELSEWHERE
       array = array-expression
      [array = expression] ... ]
END WHERE
```

The WHERE construct is similar to the WHERE statement, but more general in that several *array* = *array-expression* statements can be controlled by one *array-logical-expression*. In addition, an optional ELSEWHERE part of the construct assigns array elements whose corresponding *array-logical-expression* elements evaluate to .FALSE..

When a WHERE construct is executed, *array-logical-expression* is evaluated just once and therefore any subsequent assignment in a WHERE block (the block following the WHERE statement) or ELSEWHERE block to an entity of *array-logical-expression* has no effect on the masking. Thereafter, successive assignments in the WHERE block are evaluated in sequence as if they were specified in a WHERE statement, as follows:

WHERE (*array-logical-expression*) *array* = *array-expression*

Each assignment in the ELSEWHERE is executed as if it were:

WHERE (.NOT.*array-logical-expression*) *array* = *array-expression*

For example, the following WHERE construct:

```
WHERE (a > b)
  a = b
  b = 0
ELSEWHERE
  b = a
  a = 0
END WHERE
```

is evaluated as if it was specified as:

```
mask = a > b
WHERE (mask) a = b
WHERE (mask) b = 0
WHERE (.NOT.mask) b = a
WHERE (.NOT.mask) a = 0
```

Only assignment statements may appear in a WHERE block or an ELSEWHERE block. Within a WHERE construct, only the WHERE statement may be the target of a branch.

The form of a WHERE construct is similar to that of an IF construct, but with this important difference: no more than one block of an IF construct may be executed, but in a WHERE construct at least one (and possibly both) of the WHERE and ELSEWHERE blocks will be executed. In a WHERE construct, this difference has the effect that results in a WHERE block may feed into, and hence affect, variables in the ELSEWHERE block. Notice, however, that results generated in an ELSEWHERE block cannot feed back into variables in the WHERE block.

The following example score2grade.f90 illustrates the use of a masked assignment to find the letter-grade equivalent for each test score in the array `test_score`. To do the same operation without the benefit of masked array assignment would require a `DO` loop iterating over the array either in an `IF-ELSE-IF` construct or in a `CASE` construct, testing and assigning to each element at a time.

score2grade.f90

```
PROGRAM main
  ! illustrates the use of the WHERE statement in masked array
  ! assignment
  !
  ! use an array constructor to initialize the array that holds
  ! the numerical scores
  INTEGER, DIMENSION(10) :: test_score = &
    (/75,87,99,63,75,51,79,85,93,80/)
  ! array to hold the equivalent letter grades (A, B, C, etc.)
  CHARACTER, DIMENSION(10) :: letter_grade
  ! because the array arguments are declared in the procedure
  ! as assumed-shape arrays, the procedure's interface must
  ! be explicit
  !
  INTERFACE
    SUBROUTINE convert(num, letter)
      INTEGER :: num(:)
      CHARACTER :: letter(:)
    END SUBROUTINE convert
  END INTERFACE

  PRINT *, 'Numerical score:', test_score
  CALL convert(test_score, letter_grade)
  PRINT '(A,10A3)', ' Letter grade:   ', letter_grade
END PROGRAM main

SUBROUTINE convert(num, letter)
  ! declare the dummy arguments as assumed-shape arrays
  INTEGER :: num(:)
  CHARACTER :: letter(:)

  ! use the WHERE statements to figure the letter grade
  ! equivalents
  WHERE (num >= 90) letter = 'A'
  WHERE (num >= 80 .AND. num < 90) letter = 'B'
  WHERE (num >= 70 .AND. num < 80) letter = 'C'
  WHERE (num >= 60 .AND. num < 70) letter = 'D'
  WHERE (num < 60) letter = 'F'
END SUBROUTINE convert
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 score2grade.f90
$ a.out
 Numerical score: 75 87 99 63 75 51 79 85 93 80
 Letter grade:     C  B  A  D  C  F  C  B  A  B
```

The next example is a subroutine that uses the WHERE construct to replace each positive element of array a by its square root. The remaining elements calculate the complex square roots of their values, which are then stored in the corresponding elements of the complex array ca. In the ELSEWHERE part of the construct, the assignment to array a should not appear before the assignment to array ca; otherwise, all of ca will be set to zero.

```
SUBROUTINE find_sqrt(a, ca)
  REAL :: a(:)
  COMPLEX :: ca(:)

  WHERE (a > 0.0)
    ca = CMPLX(0.0)
    a  = SQRT(a)
  ELSEWHERE
    ca = SQRT(CMPLX(a))
    a  = 0.0
  END WHERE
END SUBROUTINE find_sqrt
```

6 Execution control

The normal flow of execution in a Fortran 90 program is sequential. Statements execute in the order of their appearance in the program. However, you can alter this flow. The following topics, described in this chapter, describe how to achieve this:

- Control constructs and statement blocks
- Flow control statements

For a full description of each Fortran 90 control statement, see Chapter 10, “HP Fortran 90 statements,” on page 241. For information about the `WHERE` construct, see “Masked array assignment” on page 99.

Control constructs and statement blocks

A control construct consists of a statement block whose execution logic is defined by one of the following control statements:

- CASE statement
- DO statement
- IF statement

A statement block is a sequence of statements delimited by a control statements and its corresponding terminal statement. A statement block consists of zero or more statements and can include nested control constructs. However, any nested construct must have its beginning and end within the same statement block.

Although the Standard forbids transferring control *into* a statement block except by means of its control statement, HP Fortran 90 allows it. The Standard does permit the transferring control *out of* a statement block. For example, the following IF construct contains a GO TO statement that legally transfers control to a label that is defined outside the IF construct:

```
  IF (var > 1) THEN
    var1 = 1
  ELSE
    GO TO 2
  END IF
    ...
2 var1 = var2
```

The next logical IF statement is nonstandard (but permitted by HP Fortran 90) because it would transfer control into the DO construct:

```
  IF (.NOT.done) GO TO 4  ! nonstandard!
    ...
  DO i = 1, 100
    sum = b + c
4   b = b + 1
  END DO
```

The following sections describe the operations performed by the three control constructs.

CASE construct

The CASE construct selects (at most) one out of a number of statement blocks for execution.

Syntax

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

Notes on syntax

case-selector is one of the following:

- *case-value*
- *low* :
- : *high*
- *low* : *high*

case-selectors must be mutually exclusive and must agree in type with *case-expr*.

case-expr must evaluate to a scalar value and must be an integer, logical, or character type.

If *construct-name* is given in the SELECT CASE statement, the same name may appear after any CASE statement within the construct, and must appear in the END CASE statement. The construct name cannot be used as a name for any other entity within the program unit.

CASE constructs can be nested. Construct names can then be useful in avoiding confusion.

Although the Standard forbids branching to any statement in a CASE construct other than the initial SELECT CASE statement from outside the construct, HP Fortran 90 allows it. The Standard allows branching to the END SELECT statement from within the construct.

For additional information about *case-selector*, see “CASE” on page 265.

Execution logic

The execution sequence of the CASE construct is as follows:

1. *case-expr* is evaluated.
2. The resulting value is compared to each *case-selector*.
3. If a match is found, the corresponding *statement-block* executes.
4. If no match is found but a CASE DEFAULT statement is present, its *statement-block* executes.
5. If no match is found and there is no CASE DEFAULT statement, execution of the CASE construct terminates without any block executing.
6. The normal flow of execution resumes with the first executable statement following the END SELECT statement, unless a statement in *statement-block* transfers control.

Example

The following CASE construct prints an error message according to the value of ios_err:

```
INTEGER :: ios_err
...
SELECT CASE (ios_err)
CASE (:900)
  PRINT *, "Unknown error"
CASE (913)
  PRINT *, "Out of free space"
CASE (963:971)
  PRINT *, "Format error"
CASE (1100:)
  PRINT *, "ISAM error"
CASE DEFAULT
  PRINT *, "Miscellaneous Error"
END SELECT
```

DO construct

The DO construct repeatedly executes a statement block. The syntax of the DO statement provides two ways to specify the number of times the statement block executes:

- By specifying a loop count.
- By testing a logical expression as a condition for executing each iteration.

You can also omit all control logic from the DO statement, in effect creating an infinite loop. The following sections describe the three variations of the DO construct.

You can use the CYCLE and EXIT statements to alter the execution logic of the DO construct. For information about these statements, see “Flow control statements” on page 112.

Counter-controlled DO loop

A counter-controlled DO loop uses an index variable to determine the number of times the loop executes.

Syntax

```
[ construct-name : ] DO index = init, limit [ , step ]
   statement-block
END DO [ construct-name ]
```

HP Fortran 90 also supports the older, FORTRAN 77-style syntax of the DO loop:

```
DO label index = init, limit [ , step ]
   statement-sequence
label terminal-statement
```

A third form, combining elements of the other two, is also supported:

```
[construct-name :] DO label index = init, limit [, step]
```

For a full description of the DO loop syntax—including a list of legal *terminal-statements*—see “DO” on page 297.

Execution logic

The following execution steps apply to all three syntactic forms, except as noted:

1. The loop becomes active, and *index* is set to *init*.
2. The iteration count is determined by the following expression:

   `MAX( INT ( `*limit*` - `*init*` + `*step*` ) / `*step*`, 0 )`

   step is optional, with the default value of 1. It may not be 0.

   Note that the iteration count is 0 if either of the following conditions is true:

   - *step* (if present) is a positive number and *init* is greater than *limit*.
   - *step* is a negative number and *init* is less than *limit*.
3. If the iteration count is 0, the construct becomes inactive and the normal flow of execution resumes with the first executable statement following the `END DO` or terminal statement.
4. *statement-block* executes. (In the case of the old-style syntactic form, both *statement-sequence* and *terminal-statement* execute.)
5. The iteration count is decremented by 1, and *index* is incremented by *step*, or by 1 if *step* is not specified.
6. Go to Step 3.

NOTE

To ensure compatibility with older versions of Fortran, you can use the `+onetrip` compile-line option to ensure that, when a counter-controlled `DO` loop is encountered during program execution, the body of the loop executes at least once.

Examples

This example uses nested `DO` loops to sort an array into ascending order:

```
INTEGER :: scores(100)

DO i = 1, 99
  DO j = i+1, 100
    IF (scores(i) > scores(j)) THEN
      temp = scores(i)
      scores(i) = scores(j)
      scores(j) = temp
    END IF
  END DO
END DO
```

The following example uses the older syntactic form. Note that, unlike the newer form, old-style nested DO loops can share the same terminal statement:

```
   DO 10 i = 1, 99
     DO 10 j = i+1, 100
       if (scores(i) <= scores(j)) GO TO 10
       temp = scores(i)
       scores(i) = scores(j)
       scores(j) = temp
10 CONTINUE
```

Conditional DO loop

A conditional DO loop uses the WHILE syntax to test a logical expression as a condition for executing the next iteration.

Syntax

[*construct-name* :] DO WHILE (*logical-expression*)
    statement-block
END DO [*construct-name*]

Fortran 90 also supports the older syntax of the DO WHILE loop:

DO *label* WHILE (*logical-expression*)
    statement-sequence
label terminal-statement

Execution logic

1 The loop becomes active.

2 The *logical-expression* is evaluated. If the result of the evaluation is false, the loop becomes inactive, and the normal flow of execution resumes with the first executable statement following the END DO statement, or in the old DO-loop syntax, the terminal statement.

3 *statement-block* executes. (In the case of the old-style syntactic form, both *statement-sequence* and *terminal-statement* execute.)

4 Go to Step 2.

Example

```
! Compute the number of years it takes to double the value of an
! investment earning 4% interest per annum
REAL :: money, invest, interest
INTEGER :: years

money = 1000
invest = money
interest = .04
years = 0
DO WHILE (money < 2*invest) ! doubled our money?
  years = years + 1
```

```
  money = money + (interest * money)
END DO
PRINT *, "Years =", years
```

Infinite DO loop

The DO statement for the infinite DO loop contains no loop control logic. It executes a statement block for an indefinite number of iterations, until it is terminated explicitly by a statement within the block; for example, a RETURN or EXIT statement.

Syntax

```
[ construct-name :] DO
 statement-block
END DO [ construct-name ]
```

Execution logic

The execution sequence of an infinite DO loop is as follows:

1. The loop becomes active.
2. *statement-block* executes.
3. Go to Step 2.

Example

```
! Compute the average of input values; press 0 to exit
INTEGER :: i, sum, n

sum = 0
n = 0
average:  DO
  PRINT *, 'Enter a new number or 0 to quit'
  READ *, i
  IF (i == 0) EXIT
  sum = sum + i
  n = n + 1
END DO average
PRINT *, 'The average is ', sum/n
```

IF construct

The IF construct selects between alternate paths of execution. The executing path is determined by testing logical expressions. At most, one statement block within the IF construct executes.

Syntax

```
[construct-name :] IF (logical-expression1) THEN
  statement-block1
[ELSE IF (logical-expression2) THEN [construct-name]
  statement-block2 ]
   .
   .
   .
[ELSE [construct-name]
  statement-block3]
END IF [construct-name]
```

Execution logic

1 *logical-expression1* is evaluated. If it is true, *statement-block1* executes.

2 If *logical-expression1* evaluates to false and ELSE IF statements are present, the *logical-expression* for each ELSE IF statement is evaluated. The first expression to evaluate to true causes the associated *statement-block* to execute.

3 If all expressions evaluate to false and the ELSE statement is present, its *statement-block* executes. If the ELSE statement is not present, no statement block within the construct executes.

4 The normal flow of execution resumes with the first executable statement following the END IF statement.

Example

```
! Compare two integer values
IF ( num1 < num2 ) THEN
    PRINT *, "num1 is smaller than num2."
ELSE IF ( num1 > num2 ) THEN
    PRINT *, "num1 is greater than num2."
ELSE
    PRINT *, "The numbers are equal"
END IF
```

Flow control statements

Flow control statements alter the normal flow of program execution or the execution logic of a control construct. For example, the `GO TO` statement can be used to transfer control to another statement within a program unit, and the `EXIT` statement can terminate execution of a `DO` construct.

This section describes the operations performed by the following flow control statements:

- `CONTINUE` statement
- `CYCLE` statement
- `EXIT` statement
- Assigned `GO TO` statement
- Computed `GO TO` statement
- Unconditional `GO TO` statement
- Arithmetic `IF` statement
- Logical `IF` statement
- `PAUSE` statement
- `STOP` statement

For additional information about these statements, see Chapter 10, “HP Fortran 90 statements,” on page 241.

CONTINUE statement

The CONTINUE statement has no effect on program execution. It is generally used to mark a place for a statement label, especially when it occurs as the terminal statement of a FORTRAN 77-style DO loop.

Syntax

```
CONTINUE
```

Execution logic

No action occurs.

Example

```
! find the 50th triangular number
   triangular_num = 0
   DO 10 i = 1, 50
       triangular_num = triangular_num + i
10 CONTINUE
   PRINT *, triangular_num
```

CYCLE statement

The CYCLE statement interrupts execution of the current iteration of a DO loop.

Syntax

CYCLE [*do-construct-name*]

Execution logic

1. The current iteration of the enclosing DO loop terminates. Any statements following the CYCLE statement do not execute.
2. If *do-construct-name* is specified, the iteration count for the named DO loop decrements. If *do-construct-name* is not specified, the iteration count for the immediately enclosing DO loop decrements.
3. If the iteration count is nonzero, execution resumes at the start of the statement block in the named (or enclosing) DO loop. If it is zero, the relevant DO loop becomes inactive.

Example

```
LOGICAL :: even
INTEGER :: number

loop: DO i = 1, 10
   PRINT *, "Enter an integer: "
   READ *, number
    IF (number == 0) THEN
      PRINT *, "Must be nonzero."
      CYCLE loop
    END IF
    even = (MOD(number, 2) == 0)
    IF (even) THEN
      PRINT *, "Even"
    ELSE
      PRINT *, "Odd"
    END IF
END DO loop
```

EXIT statement

The EXIT statement terminates a DO loop. If it specifies the name of a DO loop within a nest of DO loops, the EXIT statement terminates all loops by which it is enclosed, up to and including the named DO loop.

Syntax

EXIT [*do-construct-name*]

Execution logic

If *do-construct-name* is specified, execution terminates for all DO loops that are within range, up to and including the DO loop with that name. If no name is specified, execution terminates for the immediately enclosing DO loop.

Example

```
DO
  PRINT *, "Enter a nonzero integer: "
  READ *, number
  IF (number == 0) THEN
    PRINT *, "Bye"
    EXIT
  END IF
  even_odd = MOD(number, 2)
  IF (even_odd == 0) THEN
    PRINT *, "Even"
  ELSE
    PRINT *, "Odd"
  END IF
END DO
```

Assigned GO TO statement

The assigned `GO TO` statement transfers control to the statement whose statement label was assigned to an integer variable by an `ASSIGN` statement.

Syntax

`GO TO` *integer-variable* `[ , (` *label-list* `) ]`

If *label-list* is present, then the label previously assigned to *integer-variable* must be in the list.

Execution logic

Control transfers to the executable statement at *integer-variable*.

Example

```
INTEGER int_label
   .
   .
   .
   ASSIGN 20 TO int_label
   .
   .
   .
GOTO int_label
.
.
.
20 ...
```

Computed GO TO statement

The computed `GO TO` statement transfers control to one of several labeled statements, as determined by the value of an arithmetic expression.

Syntax

`GO TO (` *label-list* `) [ , ]` *integer-expression*

Execution logic

1 *integer-expression* is evaluated.

2 The resulting integer value (the index) specifies the ordinal position of the label that is selected from *label-list*.

3 Control transfers to the executable statement with the selected label. If the value of the index is less than 1 or greater than the number of labels in *label-list*, the computed `GO TO` statement has no effect, and control passes to the next executable statement in the program.

Example

```
  DO
    PRINT *, "Enter a number 1-3: "
    READ *, k
    GO TO (20, 30, 40) k
    PRINT *, "Number out of range."
    EXIT
20  i = 20
    GO TO 100
30  i = 30
    GO TO 100
40  i = 40
100 print *, i
  END DO
```

Unconditional GO TO statement

The unconditional `GO TO` statement transfers control to the statement with the specified label.

Syntax

`GO TO` *label*

Execution logic

Control transfers to the statement at *label*.

Example

Older, “dusty-deck” Fortran programs often combine the `GO TO` statement with the logical `IF` statement to form a kind of leap-frog logic, as in the following:

```
   IF ( num1 /= num2) GO TO 10
   PRINT *, ”num1 and num2 are equal.”
   GO TO 30
10 IF ( num1 > num2 ) GO TO 20
   PRINT *, ”num1 is smaller than num2.”
   GO TO 30
20 PRINT *, ”num1 is greater than num2.”
30 CONTINUE
```

Arithmetic IF statement

The arithmetic `IF` transfers control to one of three labeled statements, as determined by the value of an arithmetic expression.

Syntax

`IF (` *arithmetic-expression* `)` *label1*, *label2*, *label3*

Execution logic

1 *arithmetic-expression* is evaluated.

2 If the resulting value is negative, control transfers to the statement at *label1*.

3 If the resulting value is 0, control transfers to the statement at *label2*.

4 If the resulting value is positive, control transfers to the statement at *label3*.

Example

As shown in this example, two or more labels in the label list can be the same.

```
i = MOD(total, 3) + 1
IF ( i ) 10, 20, 10
```

Logical IF statement

The logical IF statement executes a single statement, conditional upon the value of a logical expression. The statement it executes must not be:

- A statement used to begin a construct
- Any END statement
- Any IF statement

Syntax

IF (*logical-expression*) *executable-statement*

Execution logic

1. *logical-expression* is evaluated.
2. If it evaluates to true, *executable-statement* executes.
3. The normal flow of execution resumes with the first executable statement following the IF statement. (If *executable-statement* is an unconditional GO TO statement, control resumes with the statement specified by the GO TO statement.)

Example

```
LOGICAL :: finished
.
.
.
IF ( finished ) PRINT *, "Done."
```

PAUSE statement

The PAUSE statement causes a temporary break in program execution.

Syntax

PAUSE [*pause-code*]

where *pause-code* is a character constant or a list of up to 5 digits.

Execution logic

1. Execution of the program is suspended, and the following message is written to standard output:

   `To resume execution, type 'go'.`

   If *pause-code* is specified, the following message is written:

   `To resume execution, type 'go'.`

   PAUSE *pause-code*

2 The normal flow of execution resumes after the user types the word `go` followed by **RETURN**. If the user enters anything other than `go`, program execution terminates.

If the standard input device is other than a terminal, the message is:

```
To resume execution, execute a kill -15 pid
command.
```

pid is the unique process identification number of the suspended program. The `kill` command can be issued at any terminal at which the user is logged in.

Example

```
PAUSE 999
```

STOP statement

The `STOP` statement terminates program execution.

Syntax

```
STOP [ stop-code ]
```

where *stop-code* is a character constant, a named constant, or a list of up to 5 digits.

Execution logic

Program terminates execution. If *stop-code* is specified, the following is written to standard output:

```
STOP stop-code
```

Example

```
STOP "Program has stopped executing."
```

7 Program units and procedures

This chapter describes the internal structure of each type of program unit, how it is used, and how information is communicated between program units and shared by them. This includes the following topics:

- Terminology and concepts
- Main program
- External procedures
- Internal procedures
- Statement functions
- Arguments
- Procedure interface
- Modules
- Block data program unit

For detailed information about individual statements that can be used to build program units and procedures, see Chapter 10, “HP Fortran 90 statements,” on page 241.

Terminology and concepts

The following sections define the terms and explain the concepts that are mentioned throughout this chapter.

Program units

A program consists of the following program units:

- Main program unit
- External procedure, which can be either a subroutine or a function
- Module program unit
- Block data program unit

A complete executable program contains one (and only one) main program unit and zero or more other program units, each of which is separately compilable. A program unit is an ordered set of constructs, statements, comments, and `INCLUDE` lines. The heading statement identifies the kind of program unit; it is optional in a main program unit only. An `END` statement marks the end of a program unit.

The only executable program units are the **main program** and **external procedures**. Program execution begins with the first executable statement in the main program and ends (typically) with the last. During execution, if the main program references an external procedure, control passes to the procedure, which executes and returns control to the main program. An executing procedure can also reference other procedures or even reference itself recursively.

The main program unit is described in “Main program” on page 125, and external procedures are described in “External procedures” on page 128.

The nonexecutable program units are:

- The **module program unit**, which contains data declarations, user-defined type definitions, procedure interfaces, common block declarations, namelist group declarations, and subprogram definitions used by other program units. Modules are described in “Modules” on page 161.

- The **block data program unit**, which specifies initial values for variables in named common blocks. Block data program units are described in “Block data program unit” on page 169.

Procedures

A procedure is a **subroutine** or **function** that contains a sequence of statements and that may be invoked during program execution. Depending on where and how it is used, a procedure can be one of the following:

- **Intrinsic procedures** are defined by the language and are available for use without any declaration or definition. Intrinsic procedures implement common computations that are important to scientific and engineering applications. Intrinsic procedures are described in detail in Chapter 11, “Intrinsic procedures,” on page 475.
- An **external procedure** is a separately compilable program unit whose name and any additional entry points have global scope. External procedures are described in “External procedures” on page 128.
- An **internal procedure** has more limited accessibility than an external procedure. It can appear only within a main program unit or an external procedure and cannot be accessed outside of its hosting program unit. Internal procedures are described in “Internal procedures” on page 135.
- A **module procedure** can be defined only within a module program unit and can be accessed only by use association. Module procedures are described in “Modules” on page 161.

Scope

All defined Fortran entities have a *scope* within which their properties are known. For example, a label used within a subprogram cannot be referenced directly from outside the subprogram; the subprogram is the **scoping unit** of the label. A variable declared within a subprogram has a scope that is the subprogram. A common block name can be used in any program unit, and it refers to the same entity—that is, the name has global scope. At the other extreme, the index variable used within an implied-DO loop in a DATA statement or array constructor has a scope consisting only of the implied-DO loop construct itself.

Association

If the concept of scope limits the accessibility of entities, then the concept of association permits different entities to become accessible to each other in the same or different scope. The different types of association are:

- **Argument association** is the association that is established between actual arguments and dummy arguments during a procedure reference. For more information, see “Argument association” on page 139.
- **Host association** applies to nested scoping units, where the outer scoping unit (for example, an external procedure) plays host to the inner scoping unit (for example, an internal procedure). Host association allows the host and its nested scoping units to share data. For information about internal procedures, see “Internal procedures” on page 135.
- **Pointer association** is the association between a pointer and its target that is established by a pointer assignment statement. For more information, see “Pointer association status” on page 48 and “Pointer assignment” on page 97.
- **Sequence association** is the association that is established between dummy and actual arguments when they are arrays of different rank. For more information, see “Array dummy argument” on page 140.
- **Storage association** is the association of different objects with the same storage area and is established by the `EQUIVALENCE` and `COMMON` statements. For more information about storage association, refer to the descriptions of the `EQUIVALENCE` and `COMMON` statements in Chapter 10, “HP Fortran 90 statements,” on page 241. Derived-type objects that include the `SEQUENCE` statement in their definition can also be storage associated; see “Sequence derived type” on page 41.
- **Use association** allows different program units access to module entities by means of the `USE` statement. For more information about modules and the `USE` statement, see “Modules” on page 161.

Main program

A main program is a program unit. There must be exactly one main program in an executable program. Execution always begins with the main program.

The main program can determine the overall design and structure of the complete program and often performs various computations by referencing procedures. A program may consist of the main program alone, in which case all the program logic is contained within it.

A main program has the form:

```
[PROGRAM program-name]
   [specification-part]
   [execution-part]
   [internal-procedure-part]
END [PROGRAM [program-name]]
```

program-name

is the name of the program. *program-name* can appear on the `END PROGRAM` statement only if it also appears on the `PROGRAM` statement; the name must be the same in both places.

specification-part

is zero or more of the statements listed in Table 15 as well as any of the following:

- Type declaration statement
- Derived-type definition
- Interface block
- Statement function
- Cray-style pointer statement (HP extension)
- Structure definition (HP extension)
- Record declaration (HP extension)

execution-part

is zero or more of the statements or constructs listed in Table 16 as well as any of the following:

- Assignment statement
- Pointer assignment statement

internal-procedure-part

takes the form:

```
CONTAINS
    [internal-procedure]...
```

where *internal-procedure* is one or more internal procedures; see “Internal procedures” on page 135.

Table 15 Specification statements

ALLOCATABLE	FORMAT	POINTER
COMMON	IMPLICIT	SAVE
DATA	INTRINSIC	STATIC
DIMENSION	NAMELIST	USE
EQUIVALENCE	OPTIONAL	VIRTUAL
EXTERNAL	PARAMETER	VOLATILE

Table 16 Executable statements

ACCEPT	ELSE	ON
ALLOCATE	ELSE IF	OPEN
ASSIGN	ELSEWHERE	PAUSE
BACKSPACE	ENCODE	PRINT
CALL	END	READ
CASE construct	ENDFILE	REWIND
CLOSE	EXIT	STOP
CONTINUE	FORMAT	TYPE (I/O)
CYCLE	GO TO	WHERE
DEALLOCATE	IF	WHERE construct
DECODE	IF construct	WRITE
DO	INQUIRE	
DO construct	NULLIFY	

The only required component of a main program unit is the END statement. The following is therefore a valid, compilable program:

```
END
```

External procedures

External procedures are implemented as either functions or subroutines. The major difference between the two is that a function subprogram returns a value and can therefore appear as an operand in an expression.

The following sections describe both types of external procedures, including the following topics:

- Procedure definition
- Procedure reference
- Returning from a procedure call
- Alternate entry points

For detailed information about any of the statements associated with procedures (for example, `SUBROUTINE` and `FUNCTION`), refer to Chapter 10, "HP Fortran 90 statements," on page 241.

Procedure definition

The definition of an external procedure takes the form:

external-procedure-statement
    [*specification-part*]
    [*execution-part*]
    [*internal-procedure-part*]
end-external-procedure-statement

external-procedure-statement

takes one of the following forms, depending on whether the procedure is a subroutine or function

- `[RECURSIVE] SUBROUTINE` *name* `&`
  `[([`*dummy-arg-list*`])]`
- `[RECURSIVE][`*type-spec*`] FUNCTION` *name* `&`
  `([`*dummy-arg-list*`]) [RESULT (`*result-name*`)]`

where *name* is the name of the procedure; *type-spec* is the type of the function's result value; and *dummy-arg-list* is a comma-separated list of dummy arguments, as described in

"Arguments" on page 139. The `SUBROUTINE` and `FUNCTION` statements are fully described in Chapter 10, "HP Fortran 90 statements," on page 241.

specification-part

is zero or more of the statements listed in Table 15 as well as the `AUTOMATIC` statement.

execution-part

is zero or more of the statements listed in Table 16 as well as the following statements:

- `ENTRY` statement
- `RETURN` statement

internal-procedure-part

takes the form:

```
CONTAINS
    [internal-procedure]...internal-procedure
```

is the definition of an internal procedure; see "Internal procedures" on page 135.

end-external-procedure-statement

takes one of the following forms, depending on whether the procedure is a subroutine or function:

- `END [SUBROUTINE [`*subroutine-name*`]]`
- `END [FUNCTION [`*function-name*`]]`

Procedure reference

A procedure reference—also known as a *procedure call*—occurs when a procedure name is specified in an executable statement, which causes the named procedure to execute. The following sections describe references to subroutines and functions, and recursive references—when a procedure directly or indirectly calls itself.

Referencing a subroutine

A reference to an external subroutine occurs in a `CALL` statement, which specifies either the subroutine name or one of its entry point names. The syntax of the `CALL` statement is:

`CALL` *subroutine-name* `[([`*actual-argument-list*`])]`

actual-argument-list

is a comma-separated list of the actual arguments that take the form:

`[`*keyword* `=]` *actual-argument*

keyword

is the name of a dummy argument that appears in the `SUBROUTINE` statement. For more information about *keyword*, see “Keyword option” on page 144.

actual-argument

is one of:

- Expression, including a variable name
- Procedure name
- Alternate return

For detailed information about arguments, see “Arguments” on page 139.

alternate-return

is one of:

- **label*
- &*label*

label must be a branch target in the same scoping unit as the `CALL` statement. The ampersand prefix (&) is an HP extension and is permitted in fixed source form only. For information about alternate returns, see "Returning from a procedure reference" on page 132.

For information about referencing a subroutine that implements a defined assignment, see "Defined assignment" on page 157.

Referencing a function

An external function subprogram is referenced either by its name or by one of its entry point names. The syntax of a function reference is:

name ([*actual-argument-list*])

where *name* is the function name or the name of one of its entry points (see "Alternate entry points" on page 134). *actual-argument-list* has the same as it does in a subroutine reference (see "Procedure reference" on page 130), except that it may not include an alternate return.

For information about referencing a function that implements a defined operator, see "Defined operators" on page 155.

Recursive reference

A procedure that directly or indirectly invokes itself is *recursive*. Such a procedure must have the word `RECURSIVE` added to the `FUNCTION` or `SUBROUTINE` statement.

If a function calls itself directly, both `RECURSIVE` and a `RESULT` clause must be specified in the `FUNCTION` statement, making its interface explicit.

The following is a recursive function:

```
RECURSIVE FUNCTION factorial (n) RESULT(r)
  INTEGER :: n, r
  IF (n.ne.0) THEN
    r = n*factorial(n-1)
  ELSE
    r = 1
  ENDIF
END FUNCTION factorial
```

Both internal and external procedures can be recursive.

Returning from a procedure reference

When the `END` statement of a subprogram is encountered, control returns to the calling program unit. The `RETURN` statement can be used to the same effect at any point within a procedure. The syntax of the `RETURN` statement is:

`RETURN` [*alt-return-arg*]

where *alt-return-arg* is a scalar integer expression that evaluates to the position of one of an alternate-return argument in the subroutine argument list. *alt-return-arg* is not permitted with `RETURN` statements appearing in functions.

By default, when control returns from a subroutine call, the next statement to execute is the first executable statement following the `CALL` statement. However, by specifying alternate returns as actual arguments in the subroutine call, the programmer can return control to other statements. The alternate returns are labels prefixed with an asterisk (*). Each label is inserted in the list of actual arguments in the position that corresponds to a placeholder—a simple asterisk (*)—in the dummy argument list. For example, if the subroutine `subr` has the following list of dummy arguments:

```
SUBROUTINE subr(x, y, z, *, *)
```

then the actual arguments must include two labels for alternate returns, as in the following call:

```
CALL subr(a, b, c, *10, *20)
```

As a compatibility extension, HP Fortran 90 allows the ampersand (&) as a prefix character instead of the asterisk, but only in fixed source form. Alternate returns cannot be optional, and the associated actual argument cannot have keywords. For detailed information about the syntax of the alternate return argument, refer to the descriptions of the `CALL` and `RETURN` statements in Chapter 10, “HP Fortran 90 statements,” on page 241.

The following example, alt_return.f90, illustrates the alternate return mechanism. The referenced subroutine, `subr`, selects one of two alternate return arguments based on the value of the first argument, `where_to`.

alt_return.f90

```
PROGRAM main
    ! illustrates alternate return arguments

    INTEGER :: por  ! point of return

    por = -1  ! interpreted by arithmetic IF
    CALL subr(por, *10, *15)  ! executes first
    PRINT *, 'Default returning point'
    por = 0
    CALL subr(por, *10, *15)  ! executes second
    GOTO 20  ! control should never reach here
10  PRINT *, 'Line 10 in main'
    por = 1
    CALL subr(por, *10, *15)  ! executes third
    GOTO 20  ! control should never reach here
15  PRINT *, 'Line 15 in main'
20  CONTINUE

END PROGRAM main

SUBROUTINE subr(where_to, *, *)
! Argument list includes placeholders for two alternate returns;
! the third argument, where_to, is used to select a return
! argument

    INTEGER :: where_to

    ! use arithmetic IF to select a return
    IF (where_to) 25, 30, 35  ! labels to transfer control
    PRINT *, 'Should never print'
25  PRINT *, 'Line 25 in subr'
    RETURN    ! default returning point
30  PRINT *, 'Line 30 in subr'
    RETURN 1  ! select the first return argument
35  PRINT *, 'Line 35 in subr'
    RETURN 2  ! select the second return argument

END SUBROUTINE subr
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 alt_return.f90
$ a.out
 Line 25 in subr
 Default returning point
 Line 30 in subr
 Line 10 in main
 Line 35 in subr
 Line 15 in main
```

Alternate entry points

When a procedure is referenced, execution normally begins with the first executable statement in the procedure. Using the ENTRY statement, however, the programmer can define alternate entry points into the procedure and associate a name with each entry point. Each ENTRY statement within a procedure defines a procedure entry, which can be referenced by name as a separate, additional procedure.

The syntax for the ENTRY statement is:

ENTRY *entry-name* ([*dummy-arg-list*])[RESULT (*result-name*)]

Refer to “ENTRY” on page 319 for a full description of the ENTRY statement.

Internal procedures

An internal procedure is similar to an external procedure except that:

- It must be defined within a hosting program unit—a main, external, or module program unit—following the CONTAINS statement.
- It can be referenced by the host only.
- It can access other entities by host association within the host.
- It cannot have an ENTRY statement.
- It cannot be passed as an argument.
- It cannot contain an internal procedure.

The syntax of an internal procedure definition is the same as for an external procedure (see "Procedure definition" on page 128), except that it has no internal procedure part. The reference to an internal procedure is the same as for an external procedure; see "Procedure reference" on page 130.

The following example, int_func.f90, declares and references an internal function. Note that both the external procedure and the internal procedure have an assumed-shape array as a dummy argument, which requires the procedure to have an explicit interface (see "Procedure interface" on page 151). External procedures must be declared in an interface block to make their interface explicit; the interface of internal procedures is explicit by default.

int_func.f90

```
PROGRAM main

  ! declare and initialize an array to pass to an external
  ! procedure
  REAL, DIMENSION(3) :: values = (/2.0, 5.0, 7.0/)

  ! Because the dummy argument to print_avg is an assumed-shape
  ! array (see the definition of print_avg below), the
  ! procedure interface of print_avg must
  ! be made explicit within the calling program unit.

  INTERFACE
    SUBROUTINE print_avg(x)
      REAL :: x(:)
    END SUBROUTINE print_avg
```

```
  END INTERFACE

  CALL print_avg(values)
END PROGRAM main

! print_avg is an external subprogram
SUBROUTINE print_avg(x)
  REAL :: x(:)  ! an assumed-shape array

  ! reference the internal function get_avg
  PRINT *, get_avg(x)

  CONTAINS  ! start of internal procedure part
    REAL FUNCTION get_avg(a)  ! get_avg is an internal procedure
      ! The interface of an internal procedure is explicit within
      ! the hosting unit, so this function may declare a as an
      ! assumed-shape array.
      REAL a(:)  ! an assumed-shape array

      ! references to the SUM and SIZE intrinsics
      get_avg = SUM(a) / SIZE(a)
    END FUNCTION get_avg

END SUBROUTINE print_avg
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 int_func.f90
$ a.out
 4.66667
```

Statement functions

If an evaluation of a function with a scalar value can be expressed in just one Fortran assignment statement, such a definition can be included in the specification part of a main program unit or subprogram. This definition is known as a *statement function*. It is local to the scope in which it is defined. The syntax is:

function-name (*dummy-argument-list*) = *scalar-expression*

All dummy arguments must be scalars. All entities used in *scalar-expression* must have been declared earlier in the specification part. A statement function can reference another statement function that has already been declared. The name cannot be passed as a procedure-name argument. A statement function has an explicit interface.

The following example, stmt_func.f90, is the same as the one listed in “Internal procedures” on page 135 except that it implements `get_avg` as a statement function rather than as an internal function. As noted in the comments to the program, the elements of the array `x` are passed to the statement function as separate arguments because dummy arguments of a statement function must be scalars.

stmt_func.f90

```
PROGRAM main

  ! declare and initialize an array to pass to an external
  ! procedure
  REAL, DIMENSION(3) :: values = (/2.0, 5.0, 7.0/)

  ! Because the dummy argument to print_avg is an assumed-shape
  ! array (see the definition of print_avg below), the
  ! procedure interface of print_avg must be made
  ! explicit within the calling program unit.

  INTERFACE
    SUBROUTINE print_avg(x)
      REAL :: x(:)
    END SUBROUTINE print_avg
  END INTERFACE

  CALL print_avg(values)
END PROGRAM main

! print_avg is an external subprogram
SUBROUTINE print_avg(x)
  REAL :: x(:)  ! an assumed-shape array
```

```
  ! Define the statement function get_avg.
  ! Note that the dummy arguments must be scalar, so in order
  ! to find the average of the elements of the array, we must
  ! pass each element as a separate argument
  get_avg(x1, x2, x3) = (x1 + x2 + x3) / 3

  ! reference the statement function get_avg
  PRINT *, get_avg(x(1), x(2), x(3))

END SUBROUTINE print_avg
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 stmt_func.f90
$ a.out
 4.66667
```

Arguments

Arguments data to be passed during a procedure call. Arguments are of two sorts: dummy arguments and actual arguments. **Dummy arguments** are specified in the argument list in a procedure definition. They define the number, type, kind, and rank of the **actual arguments**. The actual arguments are the arguments that appear in the procedure reference and are the actual entities to be used by the referenced procedure, even though they are known by the dummy argument names.

This section covers the following topics related to arguments:

- Argument association
- Keyword option
- Duplicated association
- `INTENT` attribute
- `%REF` and `%VAL`

Argument association

Argument association is the linkage of actual argument to dummy argument that initially occurs when a procedure having arguments is invoked. During the execution of the referenced procedure, the dummy arguments are effectively aliases for the actual arguments. After control returns to the program unit making the reference, the dummy arguments and actual arguments are no longer associated, and the actual arguments may no longer be referenced by the dummy argument names.

The principle of argument association is positional: the first item in the list of actual arguments is associated with the first item in the list of dummy arguments, and so on with the remaining arguments in each list. However, the programmer can use the keyword option to override this positional correspondence; see “Keyword option” on page 144.

Dummy and actual arguments must agree in kind, type, and rank. The corresponding dummy and actual arguments must both be scalars or both arrays; if they are both arrays, they must have the same dimensionality. Likewise, if an actual argument is an expression or a reference to a function, it must match the type and kind of the dummy argument.

The following sections provide more detailed information about these types of dummy arguments:

- Scalars
- Arrays
- Derived types
- Pointers
- Procedure names

Scalar dummy argument

If the dummy argument is a scalar, the corresponding actual argument must be a scalar or a scalar expression, of the same kind and type. If the dummy argument is a character variable and has assumed length, it inherits the length of the actual argument. Otherwise, the length of the actual argument must be at least that of the dummy argument, and only the characters within the range of the dummy argument can be accessed by the subprogram. Lengths may differ for default character types only.

Array dummy argument

If the dummy argument is an assumed-shape array, the corresponding actual argument must match in kind, type, and rank; the dummy argument takes its shape from the actual argument, resulting in an element-by-element association between the actual and dummy arguments.

If the dummy argument is an explicit-shape or assumed-size array, the kind and type of the actual argument must match but the rank need not. The elements are **sequence associated**—that is, the actual and dummy arguments are each considered to be a linear sequence of elements in storage without regard to rank or shape, and corresponding elements in each sequence are associated with each other in array element order.

A consequence of sequence association is that the overall size of the actual argument must be at least that of the dummy argument, and only elements within the overall size of the dummy argument can be accessed by referenced procedure.

For example, if an actual argument has this declaration:

```
REAL a(0:3,0:2)
```

and the corresponding dummy argument has this declaration:

```
REAL d(2,3,2)
```

then the correspondence between elements of the actual and dummy arguments is as follows:

```
Dummy   <=> Actual
-------------------
d(1,1,1) <=> a(0,0)
d(2,1,1) <=> a(1,0)
d(1,2,1) <=> a(2,0)
          ...
d(2,3,2) <=> a(3,2)
```

When an actual argument and the associated dummy argument are default character arrays, they may be of unequal character length. If this is the case, then the first character of the dummy and actual arguments are matched, and the successive characters—rather than array elements—are matched.

The next example illustrates character sequence association. Assuming this declaration of the actual argument:

```
CHARACTER*2 a(3,4)
```

and this declaration of the corresponding dummy argument:

```
CHARACTER*4 d(2,3)
```

then the correspondence between elements of the actual and dummy arguments is as follows:

```
Dummy   <=>  Actual
-------------------
d(1,1)  <=>  a(1,1)//a(2,1)
d(2,1)  <=>  a(3,1)//a(1,2)
         ...
d(2,3)  <=>  a(2,4)//a(3,4)
```

An actual argument may be an array section, but associating an array section with any other but an assumed-shape dummy argument may cause a copy of the array section to be generated and is likely to result in a degradation in performance.

For information about the different types of arrays, see “Array declarations” on page 54.

Derived-type dummy argument

When passing a derived-type object, the corresponding dummy and actual arguments of derived types are assumed to be of the same derived type. Unless the interface of the referenced procedure is explicit within the program unit that makes the reference, the compiler does not perform any type-checking. It is the programmer's responsibility to ensure that the types of the dummy argument and the actual argument are the same, such as by doing either of the following:

- Replicating the definition of the derived type in both subprograms
- Placing the definition in a module and making the definition available to both subprograms by use association

For information about explicit interface, see “Procedure interface” on page 151. For information modules and use association, see “Modules” on page 161.

Pointer dummy argument

If the dummy argument has the `POINTER` attribute, the actual argument must also have the `POINTER` attribute. Furthermore, they must match in kind, type, and rank. If the dummy argument does not have the `POINTER` attribute but the actual argument is a pointer, the argument association behaves as if the pointer actual argument were replaced by its target at the time of the procedure reference.

Procedure dummy argument

If a dummy argument is a procedure, the actual argument must be the name of an appropriate subprogram, and its name must have been declared as `EXTERNAL` in the calling unit or defined in an interface block (see “Procedure interface” on page 151). Internal procedures, statement functions, and generic names may *not* be passed as actual arguments.

If the actual argument is an intrinsic procedure, the appropriate specific name must be used in the reference. It must have the `INTRINSIC` attribute.

The following example, intrinsic_arg.f90, declares the intrinsics `QSIN` and `QCOS` with the `INTRINSIC` attribute so that they can be passed as arguments to the user-defined subroutine `call_int_arg`. Note that the dummy argument, `trig_func`, is declared in the subroutine with the

EXTERNAL attribute to indicate that it is a dummy procedure. This declaration does not conflict with the declaration of the actual arguments in the main program unit because each occurs in different scoping units.

intrinsic_arg.f90

```
PROGRAM main
  ! declare the intrinsics QSIN and QCOS with the INTRINSIC
  ! attribute to allow them to be passed as arguments
  REAL(16), INTRINSIC :: QSIN, QCOS

  CALL call_int_arg(QSIN)
  CALL call_int_arg(QCOS)
END PROGRAM main

SUBROUTINE  call_int_arg(trig_func)
! trig_func is an intrinsic function--see the declarations
! of the actual arguments in the main program.  trig_func
! is declared here as EXTERNAL to indicate that it is a
! dummy procedure.

  REAL(16), EXTERNAL :: trig_func
  REAL(16), PARAMETER :: pi=3.1415926
  INTEGER :: i

  DO i = 0, 360, 45
    ! Convert degrees to radians (i*pi/180) and call the
    !   intrinsic procedure passed as trig_func.
    WRITE(6, 100) i,” degrees “, trig_func(i*pi/180)
  END DO
100 FORMAT (I4, A9, F12.8)
END SUBROUTINE call_int_arg
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 intrinsic_arg.f90
$ a.out
   0 degrees   0.00000000
  45 degrees   0.70710675
  90 degrees   1.00000000
 135 degrees   0.70710686
 180 degrees   0.00000015
 225 degrees  -0.70710665
 270 degrees  -1.00000000
 315 degrees  -0.70710697
 360 degrees  -0.00000030
   0 degrees   1.00000000
  45 degrees   0.70710681
  90 degrees   0.00000008
 135 degrees  -0.70710670
 180 degrees  -1.00000000
 225 degrees  -0.70710691
 270 degrees  -0.00000023
```

```
 315 degrees   0.70710659
 360 degrees   1.00000000
```

See "HP Fortran 90 statements" on page 241 for information about the EXTERNAL and INTRINSIC statements. Intrinsic procedures are fully described in Chapter 11, "Intrinsic procedures," on page 475.

Keyword option

The **keyword option** allows the programmer to specify actual arguments in a procedure reference independently of the position of the dummy arguments. Using the keyword option, the programmer explicitly pairs an actual argument with its dummy argument, as shown by the syntax:

dummy-argument = *actual-argument*

If the keyword option is used for an argument, it must be followed by other arguments with the keyword option. If all arguments in the argument list use the keyword option, the actual arguments may appear in any order.

As an example of how to use the keyword option, consider the SUM intrinsic function. As described in "SUM(ARRAY, DIM, MASK)" on page 577, this intrinsic has three arguments: array, dim, and mask, in that order; dim and mask are optional arguments. The following are therefore valid references to SUM:

```
SUM(a,2)
SUM(a,mask=a.gt.0)
SUM(dim=2,array=a)
```

The following is an invalid reference—the mask keyword must be specified:

```
SUM(a,dim=2,a.gt.0)     ! ILLEGAL, mask keyword missing
```

Optional arguments

An actual argument may be omitted from the argument list of a procedure reference if its corresponding dummy argument is optional. A dummy argument is optional if it is declared with the OPTIONAL attribute and appears at the end of the argument list. The procedure reference may also omit trailing arguments with the OPTIONAL attribute. Otherwise, keywords must be provided to maintain an identifiable correspondence (see “Keyword option” on page 144). Only procedures with an explicit interface may have optional arguments.

The following example, optional_arg.f90, references an internal function that declares one of its dummy arguments with the OPTIONAL attribute. (Internal functions have an explicit interface, making them eligible for optional arguments; see “Internal procedures” on page 135.) The function uses the PRESENT intrinsic to test whether or not the optional argument is present. If the intrinsic returns .TRUE. (an actual argument is associated with the optional dummy argument), the function returns the sum of the two arguments; otherwise, it returns the required argument incremented by 1.

optional_arg.f90

```
PROGRAM main
! illustrates the optional argument feature

  INTEGER :: arg1 = 10, arg2 = 20

  PRINT *, add_or_inc(arg1)  ! omit optional argument
  PRINT *, add_or_inc(arg1, arg2)

  CONTAINS  ! internal procedure with explicit interface
  INTEGER FUNCTION add_or_inc(i1, i2)
  ! return the sum of both arguments if the second argument
  ! (declared as optional) is present; otherwise, return the
  ! first argument incremented by 1

    INTEGER :: i1
    INTEGER, OPTIONAL :: i2  ! optional argument

    ! use PRESENT intrinsic to see if i2 has an actual
    ! argument associated with it
    IF (PRESENT(i2)) THEN
      add_or_inc = i1 + i2  ! add both arguments
    ELSE
      add_or_inc = i1 + 1   ! increment required argument
    END IF
  END FUNCTION add_or_inc
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 optional_arg.f90
$ a.out
 11
 30
```

For information about the syntax, rules and restrictions governing the OPTIONAL statement and attribute, see “OPTIONAL (statement and attribute)” on page 387. For information about the PRESENT intrinsic see “PRESENT(A)” on page 559.

Duplicated association

If a procedure reference would cause a data object to be associated with two or more dummy arguments, the object must not be redefined within the referenced procedure. Consider the following example:

```
PROGRAM p
   CALL s (a,a)
CONTAINS
   SUBROUTINE s (c,d)
     c = 22.01        ! ILLEGAL definition of one of the dummy
                      ! arguments associated with data object a
    ...
  END SUBROUTINE

END PROGRAM
```

Both dummy arguments, c and d, are associated with the actual argument a. The procedure includes an assignment to c, the effect of which is to redefine a. This attempt to redefine a is invalid. This rule actual arguments that are overlapping sections of the same array.

Similarly, if a data object is available to a procedure through both argument association and either use, host, or storage association, then the data object must be defined and referenced only through the dummy argument.

In the following code, the data object `a` is available to the subroutine as a consequence of argument association and host association. The direct reference to `a` in the subroutine is illegal.

```
PROGRAM p
   CALL s (a,b)
CONTAINS
   SUBROUTINE s (c,d)
     c = 22.01    ! valid definition of a through the dummy
                  ! argument
     d = 3.0*a    ! direct reference to a is ILLEGAL
     ...
  END SUBROUTINE
END PROGRAM
```

INTENT attribute

To enable additional compile-time checking of arguments and to avoid possibly unwanted side effects, the `INTENT` attribute can be declared for each dummy argument, which may be specified as `INTENT(IN)`, `INTENT(OUT)` or `INTENT(INOUT)`.

The values that may be specified for the `INTENT` attribute have the following significance:

- `IN` is used if the argument is not to be modified within the subprogram.
- `OUT` implies that the actual argument must not be used within the subprogram before it is assigned a value.
- `INOUT` (the form `IN OUT` is also permitted) implies that the actual argument must be defined on entry and is definable within the subprogram.

See "INTENT (statement and attribute)" on page 358 for more information about the `INTENT` attribute.

%VAL and %REF built-in functions

By default, HP Fortran 90 passes noncharacter arguments by reference. Instead of passing the value of the actual argument to the referenced procedure, Fortran passes its address, with which the name of the dummy argument becomes associated—as explained in "Argument association" on page 139. When HP Fortran 90 passes character arguments, it includes a hidden length parameter along with the address of the actual argument.

However, it is possible to change the way arguments are passed by using the `%VAL` and `%REF` built-in functions, which HP Fortran 90 provides as extensions:

- `%VAL(`*arg*`)` specifies that the value of *arg*—rather than its address—is to be passed to the referenced procedure. *arg* can be a constant variable, an array element, or a derived-type component.

- `%REF(`*arg*`)` specifies that the address of *arg* is to be passed to the referenced procedure. Because this is how HP Fortran 90 normally passes all noncharacter arguments, `%REF` is useful only when *arg* is of type character. The effect of using `%REF` with a character argument is to suppress the hidden length parameter.

These built-in functions are typically used to pass arguments from Fortran to a procedure written in another language, such as a C function. The following example illustrates this use. The program consists of a Fortran 90 main program unit and a C function. The main program calls the C function, passing 4 arguments: an integer constant, a real variable, a character variable, and an integer expression. The main program uses the built-in functions to change Fortran's argument-passing conventions to conform to C. C expects all arguments except the string—Fortran's character variable—to be passed by value. It expects the string to be passed by reference, without the hidden length parameter.

pass_args.f90

```
PROGRAM main
  REAL :: x = 3.4
  INTEGER :: i1 = 5, i2 = 7
  ! C expects strings to be null-terminated, so use the
  ! concatenation operator to append a null character.
  CHARACTER(LEN=5) :: str = "Hi!"//CHAR(0)

  ! Pass 4 arguments--a constant, a variable, a character
  ! variable, and an expression--to a function written in C.
  ! Use HP Fortran 90's built-in functions to change the
  ! argument-passing conventions to conform to C.
  CALL get_args(%VAL(20), %VAL(x), %REF(str), %VAL(i1+i2))
END PROGRAM main
```

get_args.c

```
#include <stdio.h>

/* accept 4 arguments from a Fortran 90 program, which are
 * passed as C expects them to be passed
 */
void get_args(int i1, float x, char *s, int i2)
{
  /* display argument values */
  printf("First argument:   %i\n", i1);
  printf("Second argument:  %f\n", x);
  printf("Third argument:   %s\n", s);
  printf("Fourth argument:  %i\n", i2);
}
```

Here are the command lines to compile and link both files, and to execute the program, along with the output from a sample run:

```
$ cc -Aa -c get_args.c
$ f90 pass_args.f90 get_args.o
$ a.out
First argument:   20
Second argument:  3.400000
Third argument:   Hi!
Fourth argument:  12
```

For additional information about multi-language programming, refer to the *HP Fortran 90 Programmer's Guide*. The built-in functions can also be used with the `ALIAS` directive, where they have a slightly different syntax.

Procedure interface

A **procedure interface** is the information specified in a procedure reference, including the name of the procedure, the arguments, and (if the procedure is a function) the result. If the interface is **explicit**, all of the characteristics of the arguments and the result—type, kind, attributes, and number—are defined within the scope of the reference. If the interface is **implicit**, the compiler may be able to make sufficient assumptions about the interface to permit the procedure reference.

All procedure interfaces are implicit except for the following:

- Intrinsic procedure
- Internal procedure
- Module procedure
- Recursive function that specifies a result clause
- External procedure whose interface is declared in an interface block

An explicit interface is required when:

- The procedure reference uses the keyword form of an actual argument.
- The procedure has `OPTIONAL` arguments.
- Any dummy argument is an assumed-shape array or a pointer.
- The result of a function is array-valued or a pointer.
- The procedure is a character function, the length of which is determined dynamically.
- The procedure reference is to a generic name.
- The procedure reference implements a user-defined operator or assignment.
- The procedure has the same name as an intrinsic procedure, but you want it to have precedence over the intrinsic; see “Availability of intrinsics” on page 476.

- You want the compiler to perform argument-checking at compile-time.

The following sections describe the interface block and its use for creating:

- Generic procedures
- Defined operators
- Defined assignment

Interface blocks

An **interface block** is used to provide an explicit interface for external procedures or to define a generic procedure. An interface block may appear in any program unit, except a block data program unit. It is specified in the specification part of the program unit.

The syntax for an interface block is:

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

generic-spec

is one of:

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

If *generic-spec* is omitted, then the `MODULE PROCEDURE` statement must also be omitted.

generic-name

is the name of the generic procedure that is referenced in the subprogram containing the interface block.

operator

is a unary or binary operator—intrinsic or user-defined—of the form:

.*letter*[*letter*]... .

interface-body

is:

function-statement
[*specification-part*]
end-function-statement

or

subroutine-statement
[*specification-part*]
end-subroutine-statement

module-procedure-name-list

is a comma-separated list of names of module procedures that have *generic-spec* as a generic interface. Each module-procedure name must be accessible either by use association or—if this interface block is in a module that defines the module procedure—by host association.

If the `MODULE PROCEDURE` statement is present, then *generic-spec* must also be present.

The following example, proc_interface.f90, uses an interface block in the main program unit to provide an explicit interface for the function `avg`.

proc_interface.f90

```
! Define an external function avg with one assumed-shape dummy
! argument.  Note that the definition of the function must
! lexically precede its declaration in the interface block.
REAL FUNCTION avg(a)
  REAL a(:)
  avg = SUM(a)/SIZE(a)
END FUNCTION avg

PROGRAM main
  REAL,DIMENSION(3) :: x
  INTERFACE
    REAL FUNCTION avg(a)
      REAL, INTENT(IN) :: a(:)
    END FUNCTION avg
  END INTERFACE
  x=(/2.0, 4.0, 7.0/)
  PRINT *, avg(x)
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 proc_interface.f90
$ a.out
 4.33333
```

Generic procedures

The Fortran 90 concept of **generic procedures** extends the FORTRAN 77 concept of generic intrinsics to allow user-defined generic procedures. A procedure is generic if its name—a *generic name*—is associated with a set of specific procedures. Referencing the generic name allows actual arguments to differ in type, kind, and rank. The differences in the arguments determine which specific procedure is invoked.

A generic procedure is defined in an interface block that specifies its name and the interfaces of the specific procedures; see “Interface blocks” on page 152. The specific procedures within the interface block must all be subroutines or all functions. The interface for each procedure must differ from the others in one or more of the following ways:

- The number of dummy arguments must differ.
- Arguments that occupy the same position in the dummy argument lists must differ in type, kind, or rank.
- The name of a dummy argument must differ from the names of the other dummy arguments in the argument lists of the other procedures, or all dummy arguments with the same name must differ in type, kind, or rank.

There may be more than one interface block with the same generic name, but the specific procedures whose interfaces appear in all such interface blocks must be distinguishable by the above criteria.

The `MODULE PROCEDURE` statement can be used to extend the list of specific procedures to include procedures that are otherwise accessible to the program unit containing the interface block. The `MODULE PROCEDURE` statement specifies only the procedure names; the procedure interfaces are already explicit. The `MODULE PROCEDURE` statement may appear only in an interface block that has a generic specification. Furthermore, the interface block must be contained either in the same module that contains the definitions of the named procedures or in a program unit in which the procedures are accessible through use association.

The following example assumes that two subroutines have been coded for solving linear equations: `rlineq` for when the coefficients are real, and `zlineq` for when the coefficients are complex. A generic name, `lineq`, is declared in the `INTERFACE` statement, enabling it to be used for referencing either of the specific procedures, depending on whether the arguments are real or complex:

```
INTERFACE lineq
   SUBROUTINE rlineq(ra,rb,rx)
      REAL,DIMENSION(:,:) :: ra
      REAL,DIMENSION(:) :: rb,rx
   END SUBROUTINE rlineq
   SUBROUTINE zlineq(za,zb,zx)
      COMPLEX,DIMENSION(:,:) :: za
      COMPLEX,DIMENSION(:) :: zb,zx
   END SUBROUTINE zlineq
END INTERFACE lineq
```

Defined operators

The `OPERATOR` clause can be used with the `INTERFACE` statement either to define a new user-defined operator or to extend—or *overload*—the behavior of an already defined or intrinsic operator. This second use is similar to defining a generic procedure (see “Generic procedures” on page 154). The re-defined operator becomes associated with a *generic operator*.

When the `OPERATOR` clause is present in the `INTERFACE` statement, the specific procedures within the interface block must all be functions. The functions can implement the operator for operands of different types, kinds, and ranks. These functions are restricted to one or two mandatory arguments, depending on whether the defined operator is unary or binary. The functions return the result of an expression of the form:

[*operand*] *operator operand*

Each dummy argument of the functions listed in the interface block must have the `INTENT(IN)` attribute. If operator is intrinsic, each specified function must take the same number of arguments as the intrinsic operator has operands. Furthermore, the arguments must be distinguishable from those normally associated with the intrinsic operation. However, argument keywords must not be used when the argument is specified as an operand to a defined operator.

If a user-defined operator is referenced by its generic name, the reference must resolve to a unique, specific function name. The selection of the function is accomplished by matching the number, type, kind, and rank of the operand with the dummy argument lists of the functions specified in the interface block. As with generic name references (see “Generic procedures” on page 154), exactly one procedure must match the properties of the operands, and the matching function is selected and invoked.

The following program, def_op.f90, illustrates a defined operation. The operation, `.inrect.`, compares two derived-type operands. The one operand holds the `x` and `y` co-ordinates of a point on a graph, and the other holds the set of co-ordinates defining a rectangle. If the point is inside the rectangle, the operation evaluates to `.TRUE.`. The module in which the operation is defined also contains the definitions of the types of the operands.

As noted in the comments, when a module is defined in the same file as any USE statements that reference the module, the definition must lexically precede the USE statements. For information about modules and the USE statement, see “Modules” on page 161.

def_op.f90

```
! Note that, if a module definition and any USE statements that
! reference the definition are in the same file, then the
! definition must lexically precede the USE statements.
MODULE coord_op_def
  ! Defines a logical operation for comparing two derived-type
  ! operands, as well as the derived types

  ! Define a derived type for the co-ordinates of a point
  ! in a graph
  TYPE coord_pt
    INTEGER :: x, y
  END TYPE coord_pt

  ! define a derived type for the co-ordinates of a rectangle
  TYPE rect_coords
    TYPE(coord_pt) :: p1, p2
  END TYPE rect_coords

  ! Interface block to define the logical operator .inrect.
  ! Evaluates to .TRUE. if the point operand lies inside
  ! the rectangle operand
  INTERFACE OPERATOR (.inrect.)
    MODULE PROCEDURE cmp_coords
  END INTERFACE

CONTAINS
  LOGICAL FUNCTION cmp_coords(pt, rect)
  ! returns .TRUE. if pt is inside rect

    ! arguments
    TYPE (coord_pt), INTENT (IN) :: pt
    TYPE (rect_coords), INTENT (IN) :: rect
```

```
  cmp_coords = .FALSE.    ! initialization
  IF (pt%x >= rect%p1%x .AND. pt%x < rect%p2%x     &
  .AND. pt%y >= rect%p1%y .AND. pt%y < rect%p2%y)  &
    cmp_coords = .TRUE.   ! pt is inside rect

  END FUNCTION cmp_coords
END MODULE coord_op_def

PROGRAM main
  ! make the defined operation and the derived-type definitions
  ! of the operands accessible to this program unit
  USE coord_op_def

  ! specify a value for the rectangle co-ordinates
  TYPE (rect_coords) :: rectangle =  &
     rect_coords(coord_pt(3, 5), coord_pt(7, 10))
  TYPE (coord_pt) :: point  ! user will specify value for this

  PRINT *, 'Enter two co-ordinates (integers) in a graph:'
  READ *, point

  ! perform defined operation
  IF (point .inrect. rectangle) THEN
    PRINT *, 'The point lies inside the rectangle.'
  ELSE
    PRINT *, 'The point lies outside the rectangle.'
  END IF
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 def_op.f90
$ a.out
 Enter two co-ordinates (integers) in a graph:
4,8
 The point lies inside the rectangle.
```

Defined assignment

The `ASSIGNMENT` clause can be used with the `INTERFACE` statement to specify one or more subroutines that extend—or *overload*—the assignment operator. Each subroutine must have exactly two arguments. The first argument can have either the `INTENT(OUT)` or the `INTENT(INOUT)` attribute; the second argument must have the `INTENT(IN)` attribute. The first argument corresponds to the variable on the left-hand side of an assignment statement, and the second to the expression on the right-hand side.

Similarly to generic names and defined operators, there can be more than one defined assignment, but each occurrence of the assignment statement must resolve to a unique, specific subroutine. The subroutine whose dummy arguments match the left-hand and right-hand sides of the assignment statement in kind, type, and rank is selected and invoked from the list of subroutines specified in the defined-assignment interface block.

The following example, def_assign.f90, illustrates defined assignment. The assignment consists of performing an elementary statistical analysis of the data on the right-hand operand and storing the results in the left-hand operand. As noted in the comments, when a module is defined in the same file as any USE statements that references the module, the definition must lexically precede the USE statements. For information about modules and the USE statement, see “Modules” on page 161.

def_assign.f90

```
! Note that, if a module definition and any USE statements that
! reference the definition are in the same file, then the
! definition must lexically precede the USE statements.
MODULE def_assign_stats
  ! Defines the derived-type operands and extends the assignment
  ! operator to perform a statistical analysis of the data in
  ! raw_data

  ! input data
  TYPE raw_data
    REAL :: x(100)  ! values to be averaged
    INTEGER :: n    ! number of values assigned to x
  END TYPE raw_data

  ! output data
  TYPE stats_data
    REAL :: sum, max, min, avg  ! statistical results
  END TYPE stats_data

  ! interface block to extend the assignment operator
  INTERFACE ASSIGNMENT (=)
    MODULE PROCEDURE do_stats
  END INTERFACE

CONTAINS
  SUBROUTINE do_stats(lside, rside)
  ! define the operations that are performed when
  ! rside is assigned (=) to lside

    TYPE (raw_data), INTENT (IN) :: rside
    TYPE (stats_data), INTENT (OUT) :: lside

    ! use a structure constructor for initialization
    lside = stats_data(0, 0, 9999999.9, 0)

    ! find the sum, max, and min
    DO i = 1, rside%n
      lside%sum = lside%sum + rside%x(i)
```

```
      IF (lside%max < rside%x(i)) lside%max = rside%x(i)
      IF (lside%min > rside%x(i)) lside%min = rside%x(i)
    END DO

    lside%avg = lside%sum / rside%n  ! the average
  END SUBROUTINE do_stats
END MODULE def_assign_stats

PROGRAM main
  ! Make the defined assignment and the definitions of the
  ! derived-type operands in the assignment accessible to
  ! this program unit
  USE def_assign_stats

  TYPE (raw_data) :: user_data     ! right-hand side of
  ! assignment
  TYPE (stats_data) :: user_stats  ! left-hand side of assignment

  CALL get_data(user_data)  ! collect user data
  user_stats = user_data    ! defined assignment statement

  PRINT *, 'Maximum =', user_stats%max
 PRINT *, 'Minimum =', user_stats%min
  PRINT *, 'Sum =', user_stats%sum
  PRINT *, 'Average =', user_stats%avg
END PROGRAM main

SUBROUTINE get_data(data)
  ! this subroutine stores user-input values and the number
  ! of values stored in data

  ! make the definition of raw_data accessible
  USE def_assign_stats
  TYPE (raw_data) :: data    ! the argument
  REAL :: val
  INTEGER :: i

  ! get user input
  DO i = 1, 100
    PRINT *, 'Enter a positive real (negative to quit):'
    READ *, val
    IF (val < 0.0) EXIT  ! negative, so leave
    data%x(i) = val
    data%n = i  ! count of values so far
  END DO
END SUBROUTINE get_data
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 def_assign.f90
$ a.out
 Enter a positive real (negative to quit):
25.5
 Enter a positive real (negative to quit):
35.5
 Enter a positive real (negative to quit):
45.5

 Enter a positive real (negative to quit):
-1
 Maximum = 45.5
 Minimum = 25.5
 Sum = 106.5
 Average = 35.5
```

Modules

A module is a nonexecutable program unit that contains—usually related—definitions and declarations that may be accessed by use association. Typically, modules are used for:

- Defining and declaring derived types
- Defining and declaring global data areas
- Defining operators
- Creating subprogram libraries

The definitions within a module are made accessible to other program units through **use association**. The program unit that requires access to the module must have a USE statement at the head of its specification part, and the statement must specify the name of the module.

The following sections describe the module program unit and the USE statement. The last section gives an example program that uses a module.

NOTE Compiling programs that contain modules requires care to ensure that each module is compiled before the program unit that uses it. For detailed information about compiling programs that contain modules, refer to the *HP Fortran 90 Programmer's Guide*.

Module program unit

The syntax of a module program unit is:

```
MODULE module-name
    [specification-part]
    [module-procedure-part]
END [MODULE [module-name]]
```

where:

module-name

is the name of the module.

specification-part

is zero or more of the statements listed in Table 15 with the exception of the FORMAT statement. Also, *specification-part* must not contain statement function definitions or automatic objects. (Specifying the SAVE attribute within a module is unnecessary in HP Fortran 90, as entities declared within a module retain their values by default.)

Each entity declared in *specification-part* and each of the procedure defined in *module-procedure-part* has either the PUBLIC or PRIVATE attribute. By default, all entities have the PUBLIC attribute and are thereby accessible by use association. Entities having the PRIVATE attribute are accessible from within the module only.

The PUBLIC and PRIVATE attributes and statements are fully described in Chapter 10, “HP Fortran 90 statements,” on page 241.

module-procedure-part

is:

CONTAINS
module-procedure[*module-procedure*...]

module-procedure

is either a function or subroutine. module-procedure has the same structure as an external function or subroutine except that the END statement of module-procedure must include the SUBROUTINE or FUNCTION keyword, as appropriate; for an external procedure this is optional. For information about external subroutines, see “External procedures” on page 128.

Note the following about module procedures:

- They have an explicit interface within the using program unit. It is not necessary to create an interface block for a module procedure.
- They can also contain internal procedures.
- They can be passed as an actual argument.

The following may be contained in a module and be made accessible by use association:

- Declared variables
- Named constants
- Derived-type definitions
- Procedure interfaces
- Module procedures
- Generic names
- Namelist groups

USE statement

The `USE` statement provides access to module entities within the *using program unit*—that is, the program unit in which the statement is specified. The `USE` statement specifies the name of the module that the program unit wants to access. The information in the specified module is made accessible to the program unit by *use association*. The `USE` statement must appear at the head of the specification part of a program unit.

The `USE` statement can take either of two forms:

- `USE` *module-name*`[,` *rename-list]*
- `USE` *module-name*`, ONLY :` *access-list*

where:

rename-list

is a comma separated list of:

local-name `=>` *module-entity-name*

module-entity-name

is the name of a module entity.

local-name

is the name by which *module-entity-name* will be accessed within the using program unit.

access-list

is a comma-separated list of:

[*local-name* =>] *module-entity-name*

As shown in the syntax description, the USE statement provides a renaming feature that allows module entities to be renamed within a using program unit. The association between *local-name* and *module-entity-name* is conceptually similar to argument association: the one name is an alias for the other, and the association between the two is in effect only within the using program unit.

The renaming feature can be used to resolve name conflicts when more than one module contains an entity with the same name. Consider a program unit that has access by use association to two modules: mod_defs1 mod_defs2. The names of the entities in mod_defs1 are a, b, and c; and the names of the entities in mod_defs2 are b, c, and d. The following USE statements will avoid name conflicts within the using program unit:

```
USE mod_defs1
USE mod_defs2, b => local_b, c => local_c
```

The ONLY clause provides an additional level of control over access to module entities. As described in "Module program unit" on page 161, the PRIVATE and PUBLIC attributes control access to module entities in *all* using program units. The ONLY clause controls access within a specific program unit.

For example, consider a module named mod_defs that contains the entities ent_x, ent_y, and ent_z. If a program unit contains the following USE statement:

```
USE mod_defs, ONLY : ent_x, entry += local_y
```

it has access to ent_x and ent_y only. Furthermore, it must access ent_y by the name local_y.

A program unit may have more than one USE statement specifying the same module:

- If one of the USE statements is without the ONLY clause, then all module entities with the PUBLIC attribute are accessible. Furthermore, all *local-names* from the *rename-lists* and *access-lists* are interpreted as a single concatenated *rename-list*.
- If all of the USE statements have the ONLY clause, all of the *access-lists* are interpreted as a single concatenated *access-list*.

For more information, see “USE” on page 461.

Program example

The following example program consists of three files:

- `main.f90`
- `precision.f90`
- `lin_eq_slv.f90`

The file `main.f90` is the driver that has access to entities in two modules—`precision` and `linear_equation_solver`—by use association. The modules are the other two files.

The purpose of `precision` is to communicate a kind type parameter to the other program units in the program, for the sake of precision portability. The second module—`linear_equation_solver`—contains three module procedures, the first of which, `solve_linear_equations`, uses the other two; `solve_linear_equations` is itself invoked by the main program.

Stated algebraically, the equations that `main.f90` provides as input for solution are:

```
2x + 3y + 4z = 20
3x + 4y + 5z = 26
4x + 5y - 6z = -4
```

main.f90

```
PROGRAM main
  ! use the two modules defined in precision.f90 and
  ! lin_eq_slv.f90
  USE precision
  USE linear_equation_solver
  IMPLICIT NONE
  ! the matrix a contains the coefficients to solve; b holds
  ! the constants on the right-hand side of the equation;
  ! the solution goes in x
  REAL (adequate) :: a(3,3), b(3), x(3)
  INTEGER :: i, j

  ! set by solve_linear_equations to indicate whether or not
  ! a solution was possible
  LOGICAL :: error

  ! initialize the matrix
  DO i = 1,3
```

```
    DO j = 1,3
      a(i,j) = i+j
    END DO
  END DO
  a(3,3) = -a(3,3)

  ! initialize the vector of constants
  b = (/ 20, 26, -4 /)
  CALL solve_linear_equations (a, x, b, error)

  IF (error) THEN
    PRINT *, 'Cannot solve.'
  ELSE
    PRINT *, 'The solution:', x
  END IF
END PROGRAM main
```

precision.f90

```
MODULE precision
  ! The named constant adequate is a kind number of a real
  ! representation with at least 10 digits of precision and 99
  ! digits range that normally results in 64-bit arithmetic.
  ! This constant ensures the same level of precision
  ! regardless of whether the program
  ! of whether the program is compiled on a 32-bit or 64-bit
  ! single-precision machine.
  INTEGER, PARAMETER :: adequate = SELECTED_REAL_KIND(10,99)
END MODULE precision
```

lin_eq_slv.f90

```
MODULE linear_equation_solver
  USE precision
  IMPLICIT NONE
  PRIVATE adequate  ! to avoid a "double definition" of adequate
                    ! in program units that also use precision

  ! forbid outside access to these two module procedures
  PRIVATE :: factor, back_substitution

  CONTAINS  ! module procedures defined here
  SUBROUTINE solve_linear_equations (a, x, b, error)
    ! solve the system of linear equations ax = b; set error to
    ! true if the extents of a, x, and b are incompatible or
    ! a zero pivot is found
    REAL (adequate), DIMENSION (:, :), INTENT (IN) :: a
    REAL (adequate), DIMENSION (:), INTENT (OUT) :: x
    REAL (adequate), DIMENSION (:), INTENT (IN) :: b
    LOGICAL, INTENT (OUT) :: error
    REAL (adequate), DIMENSION (SIZE (b), SIZE (b) + 1) :: m
    INTEGER :: n
    n = SIZE (b)
    ! check for compatible extents
    error = SIZE(a, DIM=1) /= n .OR. SIZE(a, DIM=2) /= n &
           .OR. SIZE(x).LT. n
```

```
  IF (error) THEN
    x = 0.0
    RETURN
  END IF

  ! append the right-hand side of the equation to m
  m (1:n, 1:n) = a
  m (1:n, n+1) = b
  ! factor m and perform forward substitution in the last
  ! column of m
  CALL factor (m, error)
  IF (error) THEN
    x = 0.0
    RETURN
  END IF
  ! perform back substitution to obtain the solution
  CALL back_substitution (m, x)
END SUBROUTINE solve_linear_equations

SUBROUTINE factor (m, error)
  ! Factor m in place into a lower and upper triangular
  ! matrix using partial pivoting
  ! Set error to true if a pivot element is zero; Perform
  ! forward substitution with the lower triangle on the
  ! right-hand side m(:,n+1)
  REAL (adequate), DIMENSION (:, :), INTENT (INOUT) :: m
  LOGICAL, INTENT (OUT) :: error
  INTEGER, DIMENSION (1) :: max_loc
  REAL (adequate), DIMENSION (SIZE (m, DIM=2)) :: temp_row
  INTEGER :: n, k
  INTRINSIC MAXLOC, SIZE, SPREAD, ABS

  n = SIZE (m, DIM=1)
  triang_loop: DO k = 1, n
    max_loc = MAXLOC (ABS (m (k:n, k)))
    temp_row (k:n+1) = m (k, k:n+1)
    m (k, k:n+1) = m (k-1+max_loc(1), k:n+1)
    m (k-1+max_loc(1), k:n+1) = temp_row (k:n+1)
    IF (m (k, k) == 0) THEN
      error = .TRUE.
      EXIT triang_loop
    ELSE
      m (k, k:n+1) = m (k, k:n+1) / m (k, k)
      m (k+1:n, k+1:n+1) = m (k+1:n, k+1:n+1) -  &
                           SPREAD (m (k, k+1:n+1), 1, n-k) *  &
                           SPREAD (m (k+1:n, k), 2, n-k+1)
    END IF
  END DO triang_loop
END SUBROUTINE factor

SUBROUTINE back_substitution (m, x)
  ! Perform back substitution on the upper triangle to compute
  ! the solution
  REAL (adequate), DIMENSION (:, :), INTENT (IN) :: m
  REAL (adequate), DIMENSION (:), INTENT (OUT) :: x
  INTEGER :: n, k
```

```
    INTRINSIC SIZE, SUM

    n = SIZE (m, DIM=1)
    DO k = n, 1, -1
      x (k) = m (k, n+1) - SUM (m (k, k+1:n) * x (k+1:n))
    END DO
  END SUBROUTINE back_substitution
END MODULE linear_equation_solver
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 precision.f90 lin_eq_slv.f90 main.f90
$ a.out
 The solution: 1.0 2.0 3.0
```

The order in which the files appear on the `f90` command line is significant: files that contain modules must be compiled before files containing the program units that use the modules. For more information about compiling programs that use modules, see the *HP Fortran 90 Programmer's Guide*

Block data program unit

A block data program unit initializes data values in common blocks. The syntax of a block data program unit is:

```
BLOCK DATA [block-data-name]
   [specification-part]
END [BLOCK DATA [block-data-name]]
```

block-data-name

is the name of the block data program unit. Note that the name is optional. If omitted, no more than one unnamed block data program unit may appear in an executable program.

specification-part

is zero or more of the following:

- Type declaration statement
- USE statement
- IMPLICIT statement
- COMMON statement
- DATA statement
- EQUIVALENCE statement
- Derived-type definition
- The following attribute-specification statements:
  - DIMENSION
  - INTRINSIC
  - PARAMETER
  - POINTER
  - SAVE
  - TARGET

If a USE statement appears in a block data program unit, it makes only the named constants accessible to the program unit.

The block data program unit can initialize more than one common block. However, a common block can be initialized in only one block data program unit. It is not necessary to initialize every object within the common block, but the common block must be completely specified.

As an extension, HP Fortran 90 allows the initialization of objects in blank—or unnamed—common. The following example illustrates this extension:

```
BLOCK DATA blank
  COMMON//aa(3),ab(5)  ! an unnamed common block
  DATA aa/3*1.0/
  DATA ab/1.0,2.0,3*4.0/
END BLOCK DATA blank
```

8 I/O and file handling

This chapter describes input/output and file handling as supported by HP Fortran 90. This includes the following topics:

- Records
- Files
- Connecting a file to a unit
- File access methods
- Nonadvancing I/O
- I/O statements
- Syntax of I/O statements
- ASA carriage control
- Example programs

Records

The record is the basic unit of Fortran 90 I/O operations. It consists of either characters or binary values, depending upon whether the record is formatted or unformatted. The following sections describe both formatted and unformatted records, plus the special case of the end-of-file record.

Note that nonadvancing I/O makes it possible to read and write partial records. For more information, see “Nonadvancing I/O” on page 187.

Formatted records

A formatted record consists of characters that have been edited during list-directed or namelist-directed I/O, or by a format specification during a data transfer. (For information about format specifications, see “Format specification” on page 207.) The length of a formatted record is measured in characters; there is no predefined maximum limit to the length of a formatted record.

Unformatted records

An unformatted record consists of binary values in machine-representable format. The length of an unformatted record is measured in bytes. Unformatted records cannot be processed by list-directed or namelist-directed I/O statements or by I/O statements that use format specifications to edit data.

End-of-file record

The end-of-file record is a special case: it contains no data and is the last record of a sequential file. The end-of-file record is written:

- By the `ENDFILE` statement
- When the file is closed—either explicitly by the `CLOSE` statement or implicitly when the program terminates—immediately following a write operation
- When a `BACKSPACE` statement executes after a write operation, before the file is backspaced

If the end-of-file record is encountered during the execution of the READ statement, the program execution will abort unless the READ statement includes the END= specifier, the IOSTAT= specifier, or both. For information about these specifiers, see the description of the READ statement in Chapter 10, “HP Fortran 90 statements,” on page 241.

Files

A file is a collection of data, organized as a sequence of logical records. Records in a file must be either all formatted or all unformatted, except for the end-of-file record.

The following sections describe the two types of files, external files and internal files.

External files

An external file is stored on disk, magnetic tape, or some other peripheral device. External files can be accessed sequentially or directly as described in “File access methods” on page 178.

Scratch files

A scratch file is a special type of external file. It is an unnamed, temporary file that exists only while it is open—that is, it exists no longer than the life of the program. HP Fortran 90 uses the *tempnam*(3S) system routine to name the scratch file. The name becomes unavailable through the file system immediately after it is created, and it cannot be seen by the *ls*(1) command and cannot be opened by any other process.

To create a scratch file, you must include the STATUS='SCRATCH' specifier in the OPEN statement, as in the following:

```
OPEN (25, STATUS='SCRATCH')
```

In all other respects, a scratch file behaves like other external files. For an example of a program that uses a scratch file, see “File access” on page 202.

Internal files

An internal file is stored in a variable where it exists for the life of the variable. Its main use is to enable programs to transfer data internally between a machine representation and a character format, using edit descriptors to make the conversions. (For more information about edit descriptors, see “Edit descriptors” on page 208.)

An internal file can be one of the following:

- A character variable
- A character array
- A character array element
- A character substring
- An integer or real array (HP Fortran 90 extension)
- Any of the above that is either a field of a structure or a component of a derived type

Note, however, that a section of a character array with a vector subscript cannot be used as an internal file.

Accessing records in an internal file is analogous to accessing them in a formatted sequential file; see “Formatted I/O” on page 178. For an example program that uses an internal file, see “Internal file” on page 198.

Connecting a file to a unit

Before a program can perform any I/O operations on an external file, it must establish a logical connection between the file and a unit number. Once the connection is established, the program can reference the file by specifying the associated unit number (a nonnegative integer expression). In the following example, the OPEN statement connects unit number 1 to the file my_data, allowing the WRITE statement to write the values in total_acct and balance to my_data:

```
OPEN (UNIT=1, FILE='my_data')
WRITE (1, '(F8.2)') total_acct, balance
```

The following sections describe three types of unit numbers:

- Those that are explicitly connected by means of the OPEN statement
- Preconnected unit numbers
- Automatically opened unit numbers

Connecting to an external file

Typically, the connection between an external file and a unit number is established by the OPEN statement. When the program is finished using the file, the connection is terminated by the CLOSE statement. Once the connection is terminated, the unit number can be assigned to a different file by means of another OPEN statement. Similarly, a file whose connection was broken by a CLOSE statement can be reconnected to the same unit number or to a different unit number.

A unit cannot be connected to more than one file at a time.

The following code establishes a connection between unit 9 and the external file first_file, which is to be by default opened for sequential access. When the program is finished with the file, the CLOSE statement terminates the connection, making the unit number available for connection to other files. Following the CLOSE statement, the program connects unit 9 to a different external file, new_file:

```
! connect unit 9 to first_file
  OPEN (9, FILE='first_file')
  ...
! process file
  ...
```

```
! terminate connection
  CLOSE (9)
! connect same unit number to new_file
  OPEN (9, FILE='new_file')
  ...
! process file
  ...
! terminate connection
  CLOSE (9)
```

Performing I/O on internal files

An internal file is not connected to a unit number and therefore does not require an `OPEN` statement. It is referenced as a character variable. In the following example, the `WRITE` statement transfers the data from `char_var` to the internal file `int_file`, using list-directed formatting. Because `int_file` is declared to be 80 characters long, it is assumed that the length of `char_var` will be no more than 80 characters.

```
CHARACTER(LEN=80) :: int_file
...
WRITE (FILE=int_file, FMT=*) char_var
```

For information about internal files, see “Internal files” on page 174.

Preconnected unit numbers

Unit numbers 5, 6, and 7 are preconnected; that is, they do not have to be explicitly opened and are connected to system-defined files, as follows:

- Unit 5 is connected to standard input—by default, the keyboard of the machine on which the program is running.
- Unit 6 is connected to standard output—by default, the terminal/ display of the machine on which the program is running.
- Unit 7 is connected to standard error—by default, the terminal/ display of the machine on which the program is running.

Each predefined logical unit is automatically opened when a Fortran 90 program begins executing and remains open for the duration of the program. This means, for example, that standard output can be used by a `PRINT` statement without prior execution of an `OPEN` statement. Attempting to `CLOSE` a preconnected logical unit has no effect.

A preconnected unit number can be reused with an `OPEN` statement that assigns it to a new file. Once a preconnected unit number is connected to a new file, however, it cannot be reconnected to its original designation.

You can use the HP-UX input/output redirection (< and >) and piping (|) operators to redirect from standard input, standard output, or standard error to a file of your own choosing.

Automatically opened unit numbers

Unit numbers that have not been associated with a file by an `OPEN` statement can be automatically opened using the `READ` or `WRITE` statement. When a file is automatically opened, a string is created of the form:

`ftn`*XX*

where *XX* is replaced by the unit number in the range 01 to 99.

If you have made an environment variable assignment of the form `ftn`*XX* = *path*, the file named in *path* is opened. Otherwise, the file whose name is `ftn`*XX* is opened in the current directory. If the file does not exist, it is created.

The following program

```
PROGRAM Auto
WRITE (11,'(A)') 'Hello, world!'
END
```

writes the string

```
Hello, world!
```

to the file `ftn11`.

If this program is compiled to `a.out` and is run as follows (using `/bin/sh` or `/bin/ksh`)

```
ftn11=datafile
export ftn11
a.out
```

the output string is written to the file `datafile` instead of `ftn11`.

Automatically opened files are always opened as sequential files. Other characteristics of an automatically opened file, such as record length and format, are determined by the data transfer statement that creates the file. If the statement does not specify formatted, list-directed, or namelist-directed I/O, the file is created as an unformatted file.

File access methods

HP Fortran 90 allows both sequential access and direct access. You specify the access method with the `OPEN` statement when you connect the file to a unit number. The following example opens the file `new_data` for direct access:

```
OPEN(40, ACCESS='DIRECT', RECL=128, FILE='new_data')
```

If you do not specify an access method, the file is opened for sequential access.

The following sections describe both sequential and direct methods.

Sequential access

Records in a file opened for sequential access can be accessed only in the order in which they were written to the file. A sequential file may consist of either formatted or unformatted records. If the records are formatted, you can use list-directed, namelist-directed, and formatted I/O statements to operate on them. If the records are unformatted, you must use unformatted I/O statements only. The last record of a sequential file is the end-of-file record.

The following sections describe the types of I/O that can be used with sequential files, namely:

- Formatted I/O
- List-directed I/O
- Namelist-directed I/O
- Unformatted I/O

Formatted I/O

Formatted I/O uses format specifications to define the appearance of data input to or output from the program, producing ASCII records that are formatted for display. (Format specifications are described in detail in “Format specification” on page 207.) Data is transferred and converted, as necessary, between binary values and character format. You cannot perform formatted I/O on a file that has been connected for unformatted I/O; see “Unformatted I/O” on page 185.

Formatted I/O can be performed only by data transfer statements that include a format specification. The format specification can be defined in the statement itself or in a `FORMAT` statement referenced by the statement.

For an example of a program that accesses a formatted file, see “File access” on page 202.

List-directed I/O

List-directed I/O is similar to formatted I/O in that data undergoes a format conversion when it is transferred but without the use of a format specification to control formatting. Instead, data is formatted according to its data type. List-directed I/O is typically used when reading from standard input and writing to standard output.

List-directed I/O uses the asterisk (`*`) as a format identifier instead of a list of edit descriptors, as in the following `READ` statement, which reads three floating-point values from standard input:

```
READ *, A, B, C
```

List-directed I/O can be performed only on internal files and on formatted, sequential external files. It works identically for both file types.

Input

Input data for list-directed input consists of values separated by one or more blanks, a slash, or a comma preceded or followed by any number of blanks. (No values may follow the slash.) An end-of-record also acts as a separator except within a character constant. Leading blanks in the first record read are not considered to be part of a value separator unless followed by a slash or comma.

Input values can be any of the values listed in Table 17. A blank is indicated by the symbol *b*.

Table 17 **Input values for list-directed I/O**

Value	Meaning
z	A null value, indicated by two successive separators with zero or more intervening blanks (for example, ,*b*/).
c	A literal constant with no embedded blanks. It must be readable by an `I`, `F`, `A`, or `L` edit descriptor. Binary, octal, and hexadecimal data are illegal.
*r**c*	Equivalent to *r* (an integer) successive occurrences of *c* in the input record. For example, `5*0.0` is equivalent to `0.0 0.0 0.0 0.0 0.0`.
*r**z*	Equivalent to *r* successive occurrences of *z*.

Reading always starts at the beginning of a new record. Records are read until the list is satisfied, unless a slash in the input record is encountered. The effect of the slash is to terminate the `READ` statement after the assignment of the previous value; any remaining data in the current record is ignored.

Table 18 outlines the rules for the format of list-directed input data.

Table 18 **Format of list-directed input data**

Data type	Input format rules
Integer	Conforms to the same rules as integer constants.
Real and double precision	Any valid form for real and double precision. In addition, the exponent can be indicated by a signed integer constant (the `Q`, `D`, or `E` can be omitted), and the decimal point can be omitted for those values with no fractional part.

Data type	Input format rules
Complex and double complex	Two integer, real, or double precision constants, separated by a comma and enclosed in parentheses. The first number is the real part of the complex or double complex number, and the second number is the imaginary part. Each of the numbers can be preceded or followed by blanks or the end of a record.
Logical	Consists of a field of characters, the first nonblank character of which must be a `T` for true or an `F` for false (excluding the optional leading decimal point). Integer constants may also appear.
Character	Same form as character constants. Delimiting with single or double quotation marks is needed only if the constant contains any separators; delimiters are discarded upon input. Character constants can be continued from one record to the next. The end-of-record does not cause a blank or any other character to become part of the constant. If the length of the character constant is greater than or equal to the length, *len*, of the list item, only the leftmost *len* characters of the constant are transferred. If the length of the constant is less than *len*, the constant is left-justified in the list item with trailing blanks.

Output

The format of list-directed output is determined by the type and value of the data in the output list and by the value of the `DELIM=` specifier in the `OPEN` statement. For information about the `DELIM=` specifier, see the description of the `OPEN` statement in Chapter 10, “HP Fortran 90 statements,” on page 241.

Table 19 summarizes the rules governing the display of each data type.

Table 19 **Format of list-directed output data**

Data type	Output format rules
Integer	Output as an integer constant.
Real and Double Precision	Output with or without an exponent, depending on the magnitude. Also, output with field width and decimal places appropriate to maintain the precision of the data as closely as possible.
Complex	Output as two numeric values separated by commas and enclosed in parentheses.
Logical	If the value of the list element is `.TRUE.`, then `T` is output. Otherwise, `F` is output.
Character	Output using the A*len* format descriptor, where *len* is the length of the character expression (adjusted for doubling). If `DELIM='NONE'` (the default), no single (') or double (”) quotation marks are doubled, and the records may not be suitable list-directed input. If the value specified by `DELIM=` is not `'NONE'`, only the specified delimiter is doubled. Character strings are output without delimiters, making them also unsuitable for list-directed input.

With the exception of character values, all output values are preceded by exactly one blank. A blank character is also inserted at the start of each record to provide ASA carriage control if the file is to be printed; see “ASA carriage control” on page 197 for a description of this. For example, the following statement:

```
PRINT *, 'Hello, world!'
```

outputs the line (where *b* indicates a blank):

b`Hello,`*b*`world!`

If the length of the values of the output items is greater than 79 characters, the current record is written and a new record started.

Slashes, as value separators, and null values are not output by list-directed `WRITE` statements.

Namelist-directed I/O

Namelist-directed I/O enables you to transfer a group of variables by referencing the name of the group, using the NML= specifier in the data transfer statement. The NAMELIST statement specifies the variables in the group and gives the group a name.

Like list-directed I/O, namelist-directed I/O does not use a format specification when formatting data but uses default formats, as determined by the data types.

In the following example, the NAMELIST statement defines the group name_group, which consists of the variables i, j, and c. The READ statement reads a record from the file connected to unit number 27 into name_group. The PRINT statement then writes the data from the variables in name_group to standard output. (As an extension, HP Fortran 90 allows this use of the PRINT statement in namelist I/O.)

```
INTEGER :: i, j
CHARACTER(LEN=10) :: c
NAMELIST /name_group/ i, j, c
...
READ (UNIT=27,NML=name_group)
PRINT name_group
```

Each namelist-directed output record begins with a blank character to provide for ASA carriage control if the records are to be printed (see “ASA carriage control” on page 197).

Namelist-directed I/O can be performed only on formatted, sequential external files.

The following program illustrates namelist-directed I/O:

```
PROGRAM namelist
INTEGER, DIMENSION(4) :: ivar
CHARACTER(LEN=3), DIMENSION(3,2) :: cvar
LOGICAL :: lvar
REAL :: rvar
NAMELIST /nl/ ivar, cvar, lvar, rvar
READ (*,nl)
PRINT nl
END PROGRAM namelist
```

If the input data is:

```
&nl
ivar = 4,3,2,1
lvar=toodles
cvar=,,'QRS',2*,2*'XXX'
rvar=5.75E25, cvar(3,2)(1:2)='AB'
/
```

then the output will be:

```
b&NLbIVAR   = 4 3 2 1bCVAR   ='', 'QRS', '',
'', 'XXX', 'ABX'bLVAR   = TbRVAR   =
5.75000E+25b/
```

The following sections describe the format of namelist-directed input and output. See "NAMELIST" on page 373 for detailed information about the `NAMELIST` statement.

Input

A namelist-directed input record takes the following form:

1. An ampersand character (&) immediately followed by a namelist group name. The group name must have been previously defined by a `NAMELIST` statement.

   As an extension, the dollar sign ($) can be substituted for the ampersand.

2. A sequence of name-value pairs and value separators. A name-value pair consists of the name of a variable in the namelist group, the equals sign (=), and a value having the same format as for list-directed input (*z*, *c*, *r***c*, and *r**). A name-value pair can appear in any order in the sequence or can be omitted.

   A value separator may be one of the following:

- Blanks
- Tabs
- Newlines
- Any of the above with a single comma

3. A terminating slash (`/`). As an extension, (`$END`) can be substituted for the slash.

Names of character type may be qualified by substring range expressions and array names by subscript/array section expressions. If the name in a name-value pair is that of an array, the number of the values following the equals sign must be separated by value separators and must not exceed the number of elements in the array. If there are fewer values than elements, null values are supplied for the unfilled elements.

Namelist-directed input values are formatted according to the same rules as for list-directed input data; see Table 18.

Output

The output record for namelist-directed I/O has the same form as the input record, but with these exceptions:

- The namelist group name is always in uppercase.
- Logical values are either `T` or `F`.
- As in list-directed output, character values are output without delimiters by default, making them unsuitable for namelist-directed input. However, you can use the `DELIM=` specifier in the `OPEN` statement to specify the single or double quotation mark as the delimiter to use for character constants.
- Only character and complex values may be split between two records.

Unformatted I/O

Unformatted I/O does not perform format conversion on data it transfers. Instead, data is kept in its internal, machine-representable format. You cannot perform unformatted I/O on files that have been connected for formatted I/O (see “Formatted I/O” on page 178).

Unformatted I/O is more efficient than formatted, list-directed, or namelist-directed I/O because the transfer occurs without the conversion overhead. However, because unformatted I/O transfers data in internal format, it is not portable.

Direct access

When performing I/O on a direct-access file, records can be read or written in any order. The records in a direct-access file are all of the same length.

Reading and writing records is accomplished by `READ` and `WRITE` statements containing the `REC=` specifier. Each record is identified by a record number that is a positive integer. For example, the first record is record number 1; the second, number 2; and so on. If `REC=` is not specified:

- The `READ` statement inputs from the current record, and the file pointer moves to the next record.
- The `WRITE` statement outputs to the record at the position of the file pointer, and the file pointer is advanced to the next record.

As an extension, HP Fortran 90 allows sequential I/O statements to access a file connected for direct access.

Once established, a record number of a specific record cannot be changed or deleted, although the record may be rewritten. A direct-access file does not contain an end-of-file record as an integral part of the file with a specific record number. Therefore, when accessing a file with a direct-access read or write statement, the `END=` specifier is not valid and is not allowed.

Direct-access files support both formatted and unformatted record types. Both formatted and unformatted I/O work exactly as they do for sequential files. However, you cannot perform list-directed, namelist-directed, or nonadvancing I/O on direct-access files.

For an example program that uses direct access, see “File access” on page 202.

Nonadvancing I/O

By default, a data transfer leaves the file positioned after the last record read or written. This type of I/O is called advancing. Fortran 90 also allows nonadvancing I/O, which positions the file just after the last character read or written, without advancing to the next record. It is character-oriented and can be used only with external files opened for sequential access. It cannot be used with list-directed or namelist-directed I/O.

To use nonadvancing I/O, you must specify `ADVANCE='NO'` in the `READ` or `WRITE` statement. The example program in “File access” on page 202 uses nonadvancing I/O in the first `WRITE` statement, which is reproduced here:

```
WRITE (6, FMT='(A)', ADVANCE='NO') &
       ' Enter number to insert in list:  '
```

The effect of nonadvancing I/O on the `WRITE` statement is to suppress the newline character that is normally output at the end of a record. This is the desired effect in the example program: by using a nonadvancing `WRITE` statement, the user input to the `READ` statement stays on the same line as the prompt.

You can get the same effect with the newline ($) edit descriptor, an HP Fortran 90 extension that also suppresses the carriage-return/linefeed sequence at the end of a record; see “Newline ($) edit descriptor” on page 211.

For an example program that illustrates nonadvancing I/O in a `READ` statement, see “Nonadvancing I/O” on page 199. For more information about nonadvancing I/O and the `ADVANCE=` specifier, see the `READ` and `WRITE` statements in Chapter 10.

I/O statements

HP Fortran 90 supports three types of I/O statements:

- Data transfer statements (see Table 20)
- File positioning statements (see Table 21)
- Auxiliary statements (see Table 22)

For detailed information about all I/O statements, refer to Chapter 10, “HP Fortran 90 statements,” on page 241.

Table 20 Data transfer statements

Statement	Use
`ACCEPT`	Inputs data from the preconnected default input device (standard input) (extension).
`DECODE`	Inputs data from an internal file (extension).
`ENCODE`	Outputs data to an internal file (extension).
`PRINT`	Outputs data to the preconnected default output device file (standard output)
`READ`	Inputs data from a connected or automatically opened unit.
`TYPE`	Synonym for the `PRINT` statement (extension).
`WRITE`	Outputs data to a connected or automatically opened unit.

NOTE

Although the `DECODE` and `ENCODE` statements are available as compatibility extensions for use with internal files, they are nonportable and are provided for compatibility with older versions of Fortran. To keep your programs standard-conforming and portable, you should use the `READ` and `WRITE` statements with both external and internal files.

`ACCEPT` and `TYPE` are also available as compatibility extensions for reading from standard input and writing to standard output. However, if you wish your program to be portable, you should use the `READ` and `PRINT` statements instead of the `ACCEPT` and `TYPE` statements.

Table 21 **File positioning statements**

Statement	Use
`BACKSPACE`	Moves the file pointer of the connected sequential file to the start of the previous record.
`ENDFILE`	Writes an end-of-file record as the next record of the sequential file.
`REWIND`	Moves the file pointer of the connected file to the initial point of the file.

Table 22 **Auxiliary statements**

Statement	Use
`CLOSE`	Disconnects a unit from a file.
`INQUIRE`	Requests information about a file or unit.
`OPEN`	Connects an existing file to a unit, creates a file and connects it to a unit, or changes certain specifiers of a connection between a file and a unit.

Syntax of I/O statements

The general syntactic form of file-positioning and auxiliary statements is:

statement-name (*io-specifier-list*)

where

statement-name	is one of the statements listed in Table 21 or Table 22.
io-specifier-list	is a comma-separated list of I/O specifiers that control the statement's operation.

The general form of a data-transfer statement is:

statement-name (*io-specifier-list*) *data-list*

where

statement-name	is one of the statements listed in Table 20.
io-specifier-list	is a comma-separated list of I/O specifiers that control the data transfer.
data-list	is a comma-separated list of data items.

The following sections describe the I/O specifiers and the form of *data-list*. For detailed information about the syntax of individual I/O statements, see Chapter 10, "HP Fortran 90 statements," on page 241.

I/O specifiers

I/O specifiers provide I/O statements with additional information about a file or a data transfer operation. They can also be used (especially with the `INQUIRE` statement) to return information about a file. Table 23 lists all I/O specifiers supported by HP Fortran 90 and identifies the statements in which each can appear. Note that the `ACCEPT`, `DECODE`, `ENCODE`, and `TYPE` statements are not listed in the table as they are nonstandard. All I/O specifiers and statements are fully described in Chapter 10, "HP Fortran 90 statements," on page 241. Each I/O specifier is described under the I/O statement in which it may appear.

Table 23 **I/O statements and specifiers**

I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
ACCESS=				✓	✓				
ACTION=				✓	✓				
ADVANCE=							✓		✓
BLANK=				✓	✓				
DELIM=				✓	✓				
DIRECT=				✓					
END=							✓		
EOR=							✓		
ERR=	✓	✓	✓	✓	✓		✓	✓	✓
EXIST=				✓					
FILE=				✓	✓				
FMT=							✓		✓
FORM=				✓	✓				
FORMATTED=				✓					
IOLENGTH=				✓					
IOSTAT=	✓	✓	✓	✓	✓		✓	✓	✓
NAME=				✓					
NAMED=				✓					
NEXTREC=				✓					
NML=							✓		✓
NUMBER=				✓					

I/O Specifiers	BACKSPACE	CLOSE	ENDFILE	INQUIRE	OPEN	PRINT	READ	REWIND	WRITE
OPENED=				✓					
PAD=				✓	✓				
POSITION=				✓	✓				
READ=				✓					
READWRITE=				✓					
REC=							✓		✓
RECL=				✓	✓				
SEQUENTIAL=				✓					
SIZE=							✓		
STATUS=		✓			✓				
UNFORMATTED=				✓					
UNIT=	✓	✓	✓	✓	✓		✓	✓	✓
WRITE=				✓					

I/O data list

The I/O data list can be used with any data transfer statement except namelist I/O; see “Namelist-directed I/O” on page 183 for a description of this. The general form of the I/O data list is:

item1[, *item2*...]

where *item* is a either a simple data element or an implied-DO loop.

The following sections describe simple data elements and the implied-DO loop.

Simple data elements

In a read operation, the simple data element specifies a variable, which can include:

- A scalar
- An array
- An array element or section
- A character substring
- A structure
- A component of a structure
- A record
- A field of a record
- A pointer

In a write operation, the simple data element can include any variable that is valid for a read operation, plus most expressions. Note that, if the expression includes a function reference, the function must not itself perform I/O.

The output list in the following `PRINT` statement contains two simple list elements, a variable named `radius` and an expression formed from `radius`:

```
99 FORMAT('Radius = ', F10.2, 'Area = ', F10.2)
   PRINT 99, radius, 3.14159*radius**2
```

The next `READ` statement contains three simple elements: a character substring (`name(1:10)`), a variable (`id`), and an array name (`scores`):

```
88 FORMAT(A10,I9,10I5)
   READ(5, 88) name(1:10), id, scores
```

If an array name is used as a simple data element in the I/O list of a `WRITE` statement, then every element in the array will be displayed. If a format specification is also used, then the format will be reused if necessary to display every element. For example, the following code

```
   INTEGER :: i(10) = (/1,2,3,4,5,6,7,8,9,10/)
88 FORMAT('   N1:',I5, ' N2:',I5, ' N3:',I5)
   PRINT 88, i
```

will output the following:

```
N1:    1 N2:    2 N3:    3
N1:    4 N2:    5 N3:    6
N1:    7 N2:    8 N3:    9
N1:   10 N2:
```

The following restrictions apply to the use of arrays in input and output:

- Sections of character arrays that specify vector-valued subscripts cannot be used as internal files.
- An assumed-size array cannot be referenced as a whole array in an input or output list.

The following restrictions apply to the use of structures and records in input and output:

- All components of the structure or fields of the record must be accessible within the scoping unit that contains the data transfer statement.
- Every component of the structure or field of the record is written.
- A structure in an I/O list must not contain a pointer that is an ultimate component—that is, the last component in a variable reference. In the expression *a%b%c*, *a* and *b* can be pointers, but not *c*.

Implied-DO loop

An implied-`DO` loop consists of a list of data elements to be read, written, or initialized, and a set of indexing parameters. The syntax of an implied-`DO` loop in an I/O statement is:

(*list* , *index* = *init* , *limit* [, *step*])

where

list
: is an I/O list, which can contain other implied-`DO` loops.

index
: is an integer variable that controls the number of times the elements in *list* are read or written. The use of real variables is supported but obsolescent.

init

is an expression that is the initial value assigned to *index* at the start of the implied-DO loop.

limit

is an expression that is the termination value for *index*.

step

is an expression by which *index* is incremented or decremented after each execution of the DO loop. *step* can be positive or negative. Its default value is 1.

Inner loops can use the indexes of outer loops.

The implied-DO loop acts like a DO construct. The range of the implied-DO loop is the list of elements to be input or output. The implied-DO loop can transfer a list of data elements that are valid for a write operation. *index* is assigned the value of *init* at the start of the loop. Execution continues in the same manner as for DO loops (see “DO construct” on page 107).

The implied-DO loop is generally used to transmit arrays and array elements, as in the following:

```
INTEGER :: b(10)
PRINT *, (b(i), i = 1,10)
```

If b has been initialized with the values 1 through 10 in order, the PRINT statement will produce the following output:

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

If an nonsubscripted array name occurs in the list, the entire array is transmitted at each iteration. For example:

```
REAL :: x(3)
PRINT *, (x, i=1, 2)
```

If x has been initialized to be [1 2 3], the output will be:

```
 1.0 2.0 3.0 1.0 2.0 3.0
```

The list can contain expressions that use the index value. For example:

```
REAL :: x(10) = (/.1, .2, .3, .4, .5, .6,  .7, .8, .9, 1 /)
PRINT *, (i*2, x(i*2), i = 1, 5)
```

print the numbers

```
2 .2 4 .4 6 .6 8 .8 10 1
```

Implied-DO loops can also be nested. The form of a nested implied-DO loop in an I/O statement is:

(((*list*, *index1* = *init1*, *limit1*, *step1*), *index2* = *init2*, *limit2*, *step2*) ... *indexN* = *initN*, *limitN*, *stepN*)

Nested implied-DO loops follow the same rules as do other nested DO loops. For example, given the following statements:

```
REAL :: a(2,2)

a(1,1) = 1
a(2,1) = 2
a(1,2) = 3
a(2,2) = 4

WRITE(6,*)((a(i,j),i=1,2),j=1,2)
```

the output will be:

```
 1.0 2.0 3.0 4.0
```

The first, or nested DO loop, is completed once for each execution of the outer loop.

ASA carriage control

The program *asa*(1) processes the output of a Fortran 90 program that uses ASA carriage control characters so that it can be properly handled by many printers.

The syntax of `asa` is:

asa [*file-names*]

where *file-names* is a list of file names to be output with carriage control characters interpreted according to ASA rules.

Table 24 describes the ASA carriage-control characters.

Table 24 **ASA carriage-control characters**

Character	Meaning
blank	Advance one line.
`0`	Advance two lines.
`1`	Advance to top of next page.
+	Do not advance; overstrike previous line.

`asa` reads input from *file-names* or from standard input if *file-names* is not specified. The first character of each line is interpreted as a control character. Lines beginning with any character other than those listed in Table 24 are interpreted as if they began with a blank, and an appropriate diagnostic appears on standard error. The first character of each line is not printed. The `asa` program interprets input lines and sends its output to standard output. Each input file begins on a new page.

To properly view the output of programs that use `asa` carriage control characters, `asa` should be used as a filter. For example, the following example pipes the output of `fortran_asa`, an executable HP Fortran 90 program that outputs lines with ASA carriage control characters, through the `asa` filter to the line printer command, `lp`:

```
fortran_asa | asa | lp
```

Example programs

This section gives example programs that illustrate I/O and file-handling features of HP Fortran 90.

Internal file

The following example, int_file.f90, illustrates how internal files can use edit descriptors internally. The comments within the program explain in detail what the program does.

int_file.f90

```
! The main program is a driver for the function roundoff, which
! truncates and rounds a floating-point number to a requested
! number of decimal places.  The main program prompts for two
! numbers, a double-precision number and an integer.  These are
! passed to the function roundoff as arguments.  The
! double-precision argument (x) is the value to be rounded, and
! the integer (n) represents the number of decimal places for
! rounding.  The function converts both arguments to character
! format, storing them in separate internal files.  The function
! uses the F edit descriptor (to which n in character format has
! been appended) to round x.  This rounded value is finally
! converted back from a character string to a double-precision
! number, which the function returns.
PROGRAM main
    REAL (KIND=8) :: x, y, roundoff

  ! Use nonadvancing I/O to suppress the newline and keep the
  !   prompt on the same line as the input.
  WRITE (6, '(X, A)', ADVANCE='NO') 'Enter a real number:  '
  READ (5, '(F14.0)') x
  WRITE (6, '(A)') 'How many significant digits (1 - 9) to the'
  WRITE (6,'(X, A)',ADVANCE='NO') 'right of the decimal point?  '

  ! Don't enter a number greater than you input into x!
  READ (5, '(I1)') n
  y = roundoff(x, n)
  PRINT *, y
END PROGRAM main

  ! This function truncates and rounds x to the number of decimal
  ! places specified by n.  The function performs no error
  ! checking on either argument.
  REAL (KIND=8) FUNCTION roundoff(x, n)
  INTEGER :: n
  REAL (KIND=8) :: x
  CHARACTER (LEN=14) :: dp_val
  CHARACTER :: dec_digits
```

```
  ! Use an edit descriptor to convert the value of n to a
  !   character; write the result to the internal file
  !   dec_digits.
  WRITE (dec_digits, '(I1)') n

  ! Concatenate dec_digits to the string 'F14.'.  The complete
  ! string forms an edit descriptor that will convert the
  ! binary value of x to a formatted value of x to a
  ! formatted character string that formats the
  ! value.  The character represents the requested level of
  ! precision.  The formatted number is stored in the internal
  ! file dp_val.
  WRITE (dp_val, '(F14.'//dec_digits//')') x

  ! Re-convert the formatted record in dp_val to a binary
  ! value that the function will return.
  READ (dp_val, '(F14.0)') roundoff

END FUNCTION roundoff
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 int_file.f90
$ a.out
 Enter a real number:  3.1415927
How many significant digits (1 - 9) to the
 right of the decimal point?  3
 3.142
```

Nonadvancing I/O

The following program reads a formatted sequential file as a set of records divided into an arbitrary number of fields. The program uses nonadvancing I/O to read and process each field. The comments explain what the program does. Included with the is a listing of the data file, `grades`, read by the program.

nonadvance.f90

```
! This program uses nonadvancing I/O to read a series of
! sequential-file records, character by character.  Each
! record is divided into fields.  The first field is the name
! of a student and is 20 characters log.  Each of the
! remaining fields s a numeric test score and is 3
! i characters long.  The name score fields.  The program
! reads the name field, then reads each score field
! until it encounters end-of-record.  When the
! program encounters end-of-record, it starts a new record.
! When it encounters and end-of-file,
! the program is done.  For the sake of simplicity, the
! program does no error-checking.
```

```
PROGRAM main
  INTEGER :: grade, count, sum, average
  CHARACTER(LEN=20) name

  OPEN(20, FILE='grades')
  WRITE (6, 10) "Name", "Average"
  WRITE (6, *) "--------------------------"
  DO  ! read and process each record
    sum = 0
    count = 0
    ! Read the first field of each record, using nonadvancing
    ! I/O so as not to advance beyond that field.  The END=
    ! specifier causes the program to exit the loop and branch
    ! to the statement at 999 when it detects end-of-file.
    READ(20, "(A20)", ADVANCE='NO', END=999) name

    ! Read each of the score fields of the record, using
    ! nonadvancing I/O to avoid advancing to the next record
    ! after each read.  The EOR= specifier causes the program
    ! to break out of the loop and resume
    ! execution at the statement labeled 99.
    DO  ! inner loop to read scores
    ! read a score and convert it to integer
      READ(20, "(I3)", ADVANCE='NO', EOR=99) grade

      count = count + 1
      sum = sum + grade
    END DO

    ! calculate average
99  average = sum/count
    WRITE(6, 20) name, average  ! write student name and average
  END DO

10 FORMAT (X, A, T21, A)
20 FORMAT (X, A, I3)
999 CLOSE(20)
END PROGRAM main
```

grades

```
Sandra Delford       79 85 81 72100100
Joan Arunsoelton      8 64 77 79
Herman Pritchard    100 92 87 65  0
Felicity Holmes      97 78 58 75 88 73
Anita Jayson         93 85 90 95 68 72 93
Phil Atley            9 27 35 49
Harriet Myrle        84 78 93 95 97 92 84 93
Pete Hartley         67 54 58 71 93 58
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 nonadvance.f90
$ a.out
 Name                  Average
 -------------------------
 Sandra Delford          86
 Joan Arunsoelton        57
 Herman Pritchard        68
 Felicity Holmes         78
 Anita Jayson            85
 Phil Atley              30
 Harriet Myrle           89
 Pete Hartley            66
```

File access

The following example, file_access.f90, illustrates both sequential and direct access on external files. The file opened for direct access is a scratch file. The comments explain what the program does.

file_access.f90

```
! This program uses an external file and a scratch file to
! insert a number into a list of numerically sorted numbers.
! The sorted list is held in a external file. The program uses
! the scratch file as a temporary holding place. The program
! uses direct access method with the scratch file.
PROGRAM main
  REAL :: number_to_insert, number_in_list
  INTEGER :: rec_num, ios1, ios2, i

  ! Initialize counter.
  rec_num = 0

  ! ios1 must be initialized to 0 so that the error-handling
  ! section at the end of the program will work correctly
  ios1= 0

  ! Open the scratch file and the sequential data file
  OPEN (18, FILE='list', STATUS='UNKNOWN', IOSTAT=ios1, ERR=99)
  OPEN (17, STATUS='SCRATCH', ACCESS='DIRECT', FORM='FORMATTED',
  &
                IOSTAT=ios1, ERR=99, RECL=16)
  ! Use nonadvancing I/O to suppress newline at the end of output
  !   record, thus keeping the prompt on the same line with the
  !   input.
  WRITE (6, FMT='(A)', ADVANCE='NO')  &
       ' Enter number to insert in list:  '
  READ *, number_to_insert

  ! Read from sorted list and write to scratch file until we find
  !   where to insert number; then, write number_to_insert, and
  !   continue writing remaining sorted numbers to scratch file.
  DO WHILE (ios1 >= 0) ! loop only if OPEN didn't encounter EOF
    ! The END=15 specifier in the READ statement gets us out of
    ! the loop, once we're in it.
    READ (18, *, END=10, IOSTAT=ios2, ERR=99) number_in_list
    IF (number_to_insert <= number_in_list) THEN
      rec_num = rec_num + 1    ! add the new record
      WRITE(17, 100, REC=rec_num) number_to_insert
      DO
        rec_num = rec_num + 1
        WRITE(17, 100, REC=rec_num) number_in_list
        READ (18, *, END=15, IOSTAT=ios2, ERR=99) number_in_list
      END DO
    ELSE
      rec_num = rec_num + 1
      WRITE (17, 100, REC=rec_num) number_in_list
    END IF
```

```
  END DO
  ! The file is empty or the item goes at the end of file.  Add 1
  !   to rec_num for the record to be inserted.
10 rec_num = rec_num + 1
  WRITE (17, 100, REC=rec_num) number_to_insert

  ! Copy the scratch file to the data file.  But first rewind
  ! so that we start writing at beginning of the data file.
15 REWIND 18

  ! Read from scratch file and write to data file
  DO i = 1, rec_num
    READ (17, 100, REC=i) number_in_list
    WRITE (18, *) number_in_list
  END DO
  CLOSE (18)
  CLOSE (17)
  STOP 'Inserted!'

  ! Error handling section
99 IF (ios1 /= 0) THEN
    WRITE (7, 200) "Open error = ", ios1
  ELSE
    WRITE (7, 200) "Read error = ", ios2
  END IF

100 FORMAT (F16.6)
200 FORMAT (A, 2I6)
END PROGRAM main
```

Here are the command lines to compile and execute the program, along with the output from a sample run. Output from the `cat` command shows the contents of the `list` file before and after executing the program:

```
$ f90 file_access.f90
$ cat list
 0.5
 1.2
 2.5
 3.5
 26.15
$ a.out
 Enter number to insert in list:  4.7
STOP Inserted!
$ cat list
 0.5
 1.2
 2.5
 3.5
 4.7
 26.15
```

9 I/O formatting

I/O formatting occurs during data transfer operations when data is converted between its machine-readable binary representation and human-readable character format. Although unformatted data transfers are faster because they do not incur the overhead of data conversion, I/O formatting is useful for displaying data in a human-readable form and for transferring data between machines with different machine representations for a data type.

I/O formatting can be implicit or explicit. Implicit formatting occurs during list-directed and namelist-directed I/O: data is converted without programmer intervention, based on the data types of the I/O list items; see “List-directed I/O” on page 179 and “Namelist-directed I/O” on page 183. Explicit formatting occurs under the control of the programmer, who specifies how the data is to be converted.

This chapter describes explicit I/O formatting and includes information about the following:

- FORMAT statement
- Format specification
- Edit descriptors
- Embedded format specification
- Nested format specifications
- Format specification and I/O data list

FORMAT statement

The function of the FORMAT statement is to specify formatting information that can be used by one or more of the following data transfer statements:

- ACCEPT (extension)
- DECODE (extension)
- ENCODE (extension)
- PRINT
- READ
- TYPE (extension)
- WRITE

The syntax of the FORMAT statement is:

label FORMAT (*format-spec*)

where:

label	is a statement label.
format-spec	is a format specification consisting of a comma-separated list of edit descriptors. For detailed information about edit descriptors, see the next section.

The FORMAT statement must include *label* so that the data transfer statements can reference it. One FORMAT statement can be referenced by many data transfer statements. In the following example, both the READ and WRITE statements reference the same FORMAT statement:

```
READ(UNIT=22, FMT=10)ivar, fvar
WRITE(17, 10)ivar, fvar
...
10 FORMAT(I7, F14.3)
```

For additional information about the FORMAT statement and data transfer statements, see Chapter 10, “HP Fortran 90 statements,” on page 241.

Format specification

A format specification consists of a list of edit descriptors that define the format of data to be read with a `READ` statement, or written with a `WRITE` or `PRINT` statement. A format specification can appear either in a `FORMAT` statement or in a character expression in a data transfer statement.

The syntax of a format specification is:

`[`*descriptor1*`[,` *descriptor2*`...]]`

where:

descriptor is an edit descriptor that is used to convert data between its internal (binary) format and an external (character) format. Edit descriptors are described in detail in the following section.

Note that format specifications are not used in list-directed and namelist-directed I/O.

Edit descriptors

Edit descriptors are encoded characters that describe data conversion between an internal (binary) format and an external (character) format. There are three types of edit descriptors:

- Data edit descriptors define the format of data to be read or written, such as its type and width (in characters). All data edit descriptors are repeatable; that is, they can be preceded by a positive integer that specifies the number of times the edit descriptor is to be replicated.
- Control edit descriptors specify editing information, such as the number of spaces between input items, treatment of blanks in input, and scale factors. Of the control edit descriptors, only the slash (/) is repeatable.
- Character string edit descriptors output text. None of these is repeatable.

All of the edit descriptors supported by HP Fortran 90 are listed in Table 25. As indicated by the syntax descriptions included in the table, the field width specification (*w*) is optional for all data edit descriptors in HP Fortran 90. Note that the Fortran 90 Standard defines the field width specifier to be optional only for the A edit descriptor. The table also identifies which edit descriptors are repeatable and which can be used on input, output, or both.

Table 25 Edit descriptors

Descriptor	Type	Repeatable?	I/O use	Function
"..." or '...'	Character string	No	Output	Output enclosed string.
$	Control	No	Output	Suppress newline at end of output.
/ (slash)	Control	Yes	Input/output	End current record and begin new record.
: (colon)	Control	No	Input/output	Stop formatting if I/O list is exhausted.

Descriptor	Type	Repeatable?	I/O use	Function
A[*w*] or R[*w*]	Data	Yes	Input/output	Convert character data.
B[*w*[.*m*]]	Data	Yes	Input/output	Convert integer data, using binary base.
BN	Control	No	Input/output	Ignore blanks in numeric input data.
BZ	Control	No	Input/output	Treat blanks as zeroes in numeric input data.
D[*w*.*d*]	Data	Yes	Input/output	Convert real type data with exponent.
E[*w*.*d*[E*e*]]	Data	Yes	Input/output	Convert real type data with exponent.
EN[*w*.*d*[E*e*]]	Data	Yes	Input/output	Convert real type data, using engineering notation.
ES[*w*.*d*[E*e*]]	Data	Yes	Input/output	Convert real type data, using scientific notation.
F[*w*.*d*]	Data	Yes	Input/output	Convert real type data without exponent.
G[*w*.*d*[E*e*]]	Data	Yes	Input/output	Convert numeric data, all types.
Q[*w*.*d*]	Data	Yes	Input/output	Convert real type data with exponent.
*n*H*s*	Character String	No	Output	Output following *n* characters.
I[*w*[.*m*]]	Data	Yes	Input/output	Convert integer numeric data.
L[*w*]	Data	Yes	Input/output	Convert logical data.
O[*w*[.*m*]]	Data	Yes	Input/output	Convert integer data, using octal base.
*k*P	Control	No	Input/output	Set scale factor to *k*.

Descriptor	Type	Repeatable?	I/O use	Function
`Q`	Control	No	Input	Return number of bytes remaining to be read in current input record.
`S` or `SP`	Control	No	Output	Print optional plus sign.
`SS`	Control	No	Output	Do not print optional plus sign.
`T`*c*	Control	No	Input/output	Move to column *c*.
`TL`*c*	Control	No	Input/output	Move *c* columns to the left.
`TR`*c* or *c*`X`	Control	No	Input/output	Move *c* columns to the right.
`Z`[*w*[.*m*]]	Data	Yes	Input/output	Convert integer data, using hexadecimal base.

The following sections describe the edit descriptors.

NOTE

There is no single edit descriptor that defines a field for complex data. Instead, you must use two real edit descriptors—the first for the real part of the number, and the second for the imaginary part. The two edit descriptors may be different or the same, and you can insert control and character string edit descriptors between them.

Likewise, there are no edit descriptors for formatting derived types and pointers. For derived types, you must specify the appropriate sequence of edit descriptors that match the data types of the derived type's components. For pointers, you must specify the edit descriptor that matches the type of the target object.

Character string ('...' or "...") edit descriptor

The character string edit descriptor is used to write a character constant to a formatted output record. It cannot be used to format input. You can use either apostrophes or quotation marks to delimit the constant. Whichever you use, they must be balanced. That is, if you begin with an apostrophe, you must also end with it. If the enclosed character constant includes a delimiting character, it must be of the other type; or you can escape the delimiter by giving another of the same type. The width of the field is the number of characters enclosed by the character string edit descriptors, including any blanks.

Table 26 provides examples of the character string edit descriptor on output. Note that *b* represents a blank.

Table 26 **Character string edit descriptor output examples**

Descriptor	Field width	Output
`'Enter data:'`	11	`Enter data:`
`"David's turn"`	12	`David's turn`
`"`*bbb*`Spaces`*bbb*`"`	12	*bbb*`Spaces`*bbb*
`'That''ll do.'`	11	`That'll do.`
`"""That'll do!"""`	13	`"That'll do!"`
`""""`	1	`"`
`'"'`	1	`"`

Newline ($) edit descriptor

The newline edit descriptor is an HP extension that suppresses the generation of the newline character (that is, the carriage-return/linefeed sequence) during formatted, sequential output. By default, the cursor moves to a newline after each output statement. The newline edit descriptor causes the cursor to remain on the same line, immediately to the right of the last character output.

NOTE Nonadvancing I/O also suppresses the newline at the end of a record. Unlike the newline ($) edit descriptor, it is a standard feature of Fortran 90, and can be used on input and output. For more information, see “Nonadvancing I/O” on page 187 and the `ADVANCE=` I/O specifier in “OPEN” on page 379.

Slash (/) edit descriptor

The slash edit descriptor terminates the current record and begins processing a new record (such as a new line on a terminal). This edit descriptor has the same result for both input and output: it terminates the current record and begins a new one. For example, on output a newline character is printed, and on input a new line is read.

Keep in mind the following considerations when using the slash edit descriptor:

- If a series of two or more slashes are written at the beginning of a format specification, the number of records skipped is equal to the number of slashes.
- If *n* slashes appear other than at the beginning of a format specification (where *n* is greater than 1), processing of the current record terminates and *n* - 1 records are skipped.
- If a format contains only *n* slashes (and no other format specifiers), *n* + 1 records are skipped.

The `/` edit descriptor does not need to be separated from other descriptors by commas.

Colon (:) edit descriptor

The colon edit descriptor (`:`) is used when performing formatted I/O to terminate format control when the I/O list has been exhausted. If all items in an I/O list have been read or written, the colon edit descriptor stops any further format processing. If more items remain in the list, the colon edit descriptor has no effect.

Consider the following example:

```
WRITE (*, 40) 1, 2
WRITE (*, 50) 1, 2
40 FORMAT(3(' value =', I2))
50 FORMAT(3(:, ' value =', I2))
```

The first `WRITE` statement outputs the line:

```
 value = 1 value = 2 value =
```

The descriptor 'value =' is repeated a third time because format control is not terminated until the descriptor `I2` is reached and not satisfied.

The second `WRITE` statement outputs the line:

```
 value = 1 value = 2
```

This time, the colon descriptor terminates format control before the string `' value='` is output a third time.

A and R (character) edit descriptors

The `A` and `R` edit descriptors define fields for character data. The `A` edit descriptor specifies left-justification, and the `R` edit descriptor specifies right-justification.

The `R` edit descriptor is an HP extension.

The syntax for the character edit descriptors is:

[*r*]A[*w*]

[*r*]R[*w*]

where:

r	is a positive integer constant, specifying the repeat factor.
w	is the field width. If *w* is not specified, the default is the length in bytes of the corresponding I/O list item.

As a portability extension, the list item can be of any data type.

When the `A` and `R` edit descriptors are used for input and output, the results can differ according to whether the width (*w*) specified for the edit descriptor is less than, greater than, or equal to the length of the I/O list item. The results on input are summarized in Table 27; the results on output are summarized in Table 28.

Table 27 **Contents of character data fields on input**

Descriptor	Width/length relationship	Result
A	width < length	Data is left-justified in variable, followed by blanks.
	width >= length	Data is taken from rightmost characters in the field.
R	width < length	Data is right-justified in variable, preceded by nulls.
	width >= length	Data is taken from rightmost characters in the field.

Table 28 **Contents of character data fields on output**

Descriptor	Width/length relationship	Result
A	width <= length	Data is taken from leftmost characters in the field.
	width > length	Output the value, preceded by blanks.
R	width <= length	Data is taken from rightmost characters in the field.
	width > length	Output the value, preceded by blanks.

Examples of the use of character edit descriptors on input are provided in Table 29. In the table, *b* represents a blank and *z* represents a Null.

Table 29 **A and R edit descriptors: input examples**

Descriptor	Input field	Variable length	Value stored
A3	XYZ	3	XYZ
R3	XYZ	4	*z*XYZ
A5	ABC*bb*	10	ABC*bbbbbbb*
R9	RIGHTMOST	4	MOST
R8	CHAIR*bbb*	8	CHAIR*bbb*
R4	CHAIR	8	*zzzz*CHAI
A4	ABCD	2	CD

Table 30 provides examples of character edit descriptors on output. In the table, *b* represents a blank and *z* represents a Null.

Table 30 **A and R Edit descriptors: output examples**

Descriptor	Internal characters	Variable length	Output
A6	ABCDEF	6	ABCDEF
R4	ABCDEFGH	8	EFGH
A4	ABCDE	5	ABCD
A8	STATUS	6	*bb*STATUS
R8	STATUS	6	*bb*STATUS
R8	STATUS	8	STATUS*bb*

B (binary) edit descriptor

The B edit descriptor defines a field for binary data. It provides for conversion between an external binary number and its internal representation.

The syntax for the binary edit descriptor is:

[*r*]B[*w*[.*m*]]

where:

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.
m	is an unsigned integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

Input

Variables to receive binary input must be of type integer. The only legal characters are 0s and 1s. Nonleading blanks are ignored, unless the file is opened with BLANK='ZERO'.

If the file is opened with BLANK='ZERO', nonleading blanks are treated as zeroes. For more information about the BLANK= specifier, see “OPEN” on page 379. Plus and minus signs, commas, or any other symbols are not permitted. If a nonbinary digit appears, an error occurs. The presence of too many digits for the integer variable (or I/O list item) is illegal.

Table 31 provides examples of the binary edit descriptor on input.

Table 31 B Edit descriptor: input examples

Descriptor	Input field (binary)	Value stored (binary)
`B8`	` 1111`	`1111`
`B8`	`01111`	`1111`
`B4`	`10101`	`1010`
`B8`	` 1.1`	error: illegal character

Output

Unlike input, list items on output may be of any type, though character values are output only as the binary equivalent of their ASCII representation (without a length descriptor). If *w* is greater than the number of converted binary digits (excluding leading zeroes), the binary digits are right-justified in the output field.

If *w* is less than the number of converted binary digits, the field is filled with *w* asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of binary digits in the entire output value.

The field width required to fully represent the binary value of an item is eight times its size in bytes. For example, an `INTEGER*4` item could require a field *w* of up to 32 characters.

Only 1s and 0s are printed on output.

Table 32 provides examples of the binary edit descriptor on output.

Table 32 B Edit descriptor: output examples

Descriptor	Internal value	Output
`B5`	27	`11011`
`B8`	27	*bbb*`11011`
`B8.6`	27	*bb*`011011`
`B8`	-27	`********`

BN and BZ (blank) edit descriptors

The BN and BZ edit descriptors control the interpretation of embedded and trailing blanks in numeric input fields. The syntax of the blank edit descriptors is:

```
BN
BZ
```

At the beginning of the execution of an input statement, blank characters within numbers are ignored except when the unit is connected with BLANK='ZERO' specified in the OPEN statement. BN and BZ override the BLANK= I/O specifier for the current READ statement. For more details about the BLANK= I/O specifier, see “OPEN” on page 379.

If a BZ edit descriptor is encountered in the format specification, trailing and embedded blanks in succeeding numeric fields are treated as zeroes. The BZ edit descriptor remains in effect until a BN edit descriptor or the end of the format specification is encountered. If BN is specified, all embedded blanks are removed and the input number is right justified within the field width.

The BN and BZ edit descriptors affect only I, B, O, F, D, E, EN, ES, G, and Z format descriptors during the execution of an input statement. The BN and BZ edit descriptors do not affect character and logical edit descriptors.

Table 33 provides examples of the BN and BZ edit descriptors on input.

Table 33 BN and BZ edit descriptors: input examples

Descriptor	Input characters	BN editing in effect	BZ editing in effect
I4	1*b*2*b*	12	1020
F6.2	*b*4*b*.*b*2	4.2	40.02
E7.1	5*b*.*b*E1*b*	5.0×10^{1}	5.0×10^{11}
E5.0	3E4*bb*	3.0×10^{4}	3.0×10^{400} (overflow)

The BN and BZ edit descriptors are ignored during the execution of an output statement.

D, E, EN, ES, F, G, and Q (real) edit descriptors

The `D`, `E`, `EN`, `ES`, `F`, `G`, and `Q` edit descriptors define fields for real numbers. The I/O list item corresponding to a real descriptor must be a numeric type. (The Standard permits real and complex types only; as an extension, HP Fortran 90 allows integers.)

The syntax for these edit descriptors is:

[*r*]D[*w.d*]

[*r*]E[*w.d*[{E|D|Q}*e*]]

[*r*]EN[*w.d*[E*e*]]

[*r*]ES[*w.d*[E*e*]]

[*r*]F[*w.d*]

[*r*]G[*w.d*[{E|D|Q}*e*]]

[*r*]Q[*w.d*]

where:

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.
d	is a nonnegative integer constant, specifying the number of decimal places on output.
e	is a positive integer constant, specifying the number of digits in the exponent.

For formatting complex data, you can use two real edit descriptors—the first for the real part of the number and the second for the imaginary part. The two edit descriptors may be different or the same, and you can insert control and character string edit descriptors between them.

Real edit descriptors on input

The input field for the real descriptors consists of an optional plus or minus sign followed by a string of digits that may contain a decimal point. If the decimal point is omitted in the input string, then the number of digits equal to *d* from the right of the string are interpreted to be to the right of the decimal point. If a decimal point appears in the input string and conflicts with the edit descriptor, the decimal point in the input string takes precedence. This basic form can be followed by an exponent in one of the following forms:

- A signed integer constant
- An `E` followed by an optionally signed integer constant
- A `D` followed by an optionally signed integer constant
- A `Q` followed by an optionally signed integer constant

All four exponent forms are processed in the same way. Note, however, that *e* has no effect on input.

The `EN` and `ES` edit descriptors are the same as the `F` edit descriptor on input. The `Q` edit descriptor (an HP Fortran 90 extension) is the same as the `E` edit descriptor on input.

Table 34 provides examples of the real edit descriptors on input. The `BZ` edit descriptor listed in the “Descriptor” column treats nonleading blanks in numeric fields as zeroes.

Table 34 D, E, F, and G edit descriptors: input examples

Descriptor	Input field	Value stored
`F6.5`	`4.51E4`	45100
`G4.2`	`51-3`	.00051
`E8.3`	`7.1`*b*`E`*b*`5`	710000
`D9.4`	*bbb*`45E+35`	.0045 x 10^{35}
`BZ, F6.1`	`-54E3`*b*	-5.4 x 10^{30}

Real edit descriptors on output

The output field for the real descriptors consists of *w* character positions, filled with leading blanks (if necessary) and an optionally signed real constant with a decimal point, rounded to *d* digits after the decimal point. The following sections describe the real edit descriptors on output in detail.

D and E edit descriptors

The `D` and `E` edit descriptors define a normalized floating-point field for real and complex values. The value is rounded to *d* digits. The exponent part consists of *e* digits. If `E`*e* is omitted in a `D` or `E` edit descriptor, then the exponent occupies two or three positions, depending on its magnitude. The field width, *w*, should follow the general rule: *w* is greater than or equal to *d*+7. If `E`*e* is used, *w* is greater than or equal to *d*+*e*+5. This rule provides positions for a leading blank, the sign of the value, the decimal point, *d* digits, the exponent letter (`D`, `E`, or `Q`), the sign of the exponent, and the exponent. The `E`*e*, `D`*e*, and `Q`*e* specifications, which are available with the `E` edit descriptor, control which exponent letter is output.

Table 35 provides examples of the `E` and `D` edit descriptors on output.

Table 35 D and E edit descriptors: output examples

Descriptor	Internal value	Output
`D10.3`	+12.342	*bb*`.123D+02`
`E10.3E3`	-12.3454	`-.123E+002`
`E12.4`	+12.34	*bbb*`.1234E+02`
`D12.4`	-.00456532	*bb*`-.4565D-02`
`D10.10`	+99.99913	`**********`
`E11.5`	+999.997	*b*`.10000E+04`
`E10.3E4`	$+.624 \times 10^{-30}$	`.624E-0030`

EN and ES edit descriptors

The EN and ES descriptors format floating-point values, using engineering and scientific notation, respectively. They are similar in form to the E descriptor, except:

- The field produced by the EN descriptor has an exponent that is divisible by 3 and a significand that is in the range 1 to 999.
- The field produced by the ES descriptor has one digit before the decimal point.

Table 36 provides examples of the EN and ES edit descriptors on output.

Table 36 EN and ES edit descriptors: output examples

Descriptor	Internal value	Output
EN12.3	+3.141	*bbb*3.141E+00
ES12.3	+3.141	*bbb*3.141E+00
EN12.3	+.00123	*bbb*1.230E-03
ES12.3	+.00123	*bbb*1.230E-03
EN12.3	-.7	-700.000E-03
ES12.3	-.7	*bb*-7.000E-01
EN12.3	+1234.5	*bbb*1.235E+03
ES12.3	+1234.5	*bbb*1.235E+03

F edit descriptor

The `F` edit descriptor defines a field for real and complex values. The value is rounded to *d* digits to the right of the decimal point. The field width, *w*, should be four greater than the expected length of the number to provide positions for a leading blank, the sign, the decimal point, and a roll-over digit for rounding if needed.

Table 37 provides examples of the `F` edit descriptor on output.

Table 37 **F edit descriptor: output examples**

Descriptor	Internal value	Output
`F5.2`	+10.567	`10.57`
`F3.1`	-254.2	`***`
`F6.3`	+5.66791432	*b*`5.668`
`F8.2`	+999.997	*b*`1000.00`
`F8.2`	-999.998	`-1000.00`
`F7.2`	-999.997	`*******`
`F4.1`	+23	`23.0`

G edit descriptor

The `G` edit descriptor can be used with any data type but is commonly used to define a field for real and complex values.

According to the magnitude of the data, the `G` edit descriptor is interpreted as either an `E` or `F` descriptor. (For more information on these edit descriptors, refer to “D and E edit descriptors” on page 221 and “F edit descriptor” on page 223.) The `E` edit descriptor is used when one of the following conditions is true:

- The magnitude is less than 0.1 but not zero.
- The magnitude is greater than or equal to `10**`*d* (after rounding to *d* digits).

If the magnitude does not fit either of these rules, the `F` edit descriptor is used. When `F` is used, trailing blanks are included in the field where the exponent would have been.

For fixed- or floating-point format descriptors, the field width is *w*. The value is rounded to *d* digits, and the exponent consists of *e* digits. If E*e* is omitted, the exponent occupies two positions. If E*e* is omitted and the exponent is greater than 99 (that is, it requires three digits), the exponent letter is dropped from the output. The field width, *w*, should follow the general rule: *w* is greater than or equal to the sum of *d*+7; or, if E*e* is specified, *w* is greater than or equal to the sum of *d*+*e*+5. This rule provides positions for a leading blank, the sign of the value, *d* digits, the decimal point, and, if needed, the exponent letter (D, E, or Q), the sign of the exponent, and the exponent. Note that the E*e*, D*e*, and Q*e* specifications control which exponent letter is output.

When used to specify I/O fields for integer, character, and logical data, the G edit descriptor has the same syntax and same effect as the integer, character, and logical edit descriptors. The *d* and *e* values (if specified) have no effect.

Table 38 provides examples of the G edit descriptor on output.

Table 38 G edit descriptor: output examples

Descriptor	Internal value	Interpretation	Output
G10.3	+1234.0	E10.3	*b*0.123E+04
G10.3	-1234.0	E10.3	-0.123E+04
G12.4	+12345.0	E12.4	*bb*0.1235E+05
G12.4	+9999.0	F8.0, 4X	*bbb*9999.*bbbb*
G12.4	-999.0	F8.1, 4X	*bb*-999.0*bbbb*
G7.1	+.09	E7.1	0.9E-01
G5.1	-.09	E5.1	*****
G11.1	+9999.0	E11.1	*bbbb*0.1E+05
G8.2	+9999.0	E8.2	0.10E+05
G7.2	-999.0	E7.2	*******

Q edit descriptor

The Q edit descriptor (an HP extension) has the same effect as the E edit descriptor on output, except that it outputs a Q for the exponent instead of an E.

The Q edit descriptor can also be used to determine the number of bytes remaining to be read in an input record; see “Q (bytes remaining) edit descriptor” on page 233.

H (Hollerith) edit descriptor

The H edit descriptor outputs a specified number of characters. The syntax is:

*n*H*character-sequence*

where:

n
: is a positive integer that specifies the number of characters to output. This number must exactly match the actual number of characters in *character-sequence*.

character-sequence
: is the string of representable characters (including blanks) to output.

Table 39 provides examples of the Hollerith edit descriptor on output.

Table 39 H edit descriptor: output examples

Descriptor	Field width	Output
`12H`*bbb*`Spaces`*bbb*	12	*bbb*`Spaces`*bbb*
`14H”It`*b*`isn't`*b*`so.”`	14	`”It`*b*`isn't`*b*`so.”`

I (Integer) edit descriptor

The `I` edit descriptor defines a field for an integer number. As an HP extension, it can also be used on real and logical data. The corresponding I/O list item must be a numeric or logical type.

The syntax of the integer edit descriptor is:

`[`*r*`I][`*w*`[.`*m*`]]`

where:

r — is a positive integer constant, specifying the repeat factor.

w — is a positive integer constant, specifying the field width.

m — is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks. If *m* = 0 and the list item is zero, only blanks are output.

Input

The integer edit descriptor causes the interpretation of the next *w* positions of the input record. The number is converted to match the type of the list item currently using the descriptor. A plus sign is optional for positive values. A decimal point must not appear in the field.

Table 40 provides examples of the integer edit descriptor on input.

Table 40 **I edit descriptor: input examples**

Descriptor	Input field	Value stored
`I4`	*b*`1`*bb*	1
`I5`	*bbbbb*	0
`I5`	*bbbbb*`1`	0
`I2`	`-1`	-1
`I4`	`-123`	-123
`I3`	*b*`12`	12

Descriptor	Input field	Value stored
`I3`	`12`*b*	12
`I3`	`12`*b*	120
`I3`	`1.1`	error: illegal character

Output

The integer edit descriptor outputs a numeric variable as a right-justified integer value (truncated, if necessary). The field width, *w*, should be one greater than the expected number of digits to allow a position for a minus sign for negative values. If *m* is set to 0, a zero value is output as all blanks.

Table 41 provides examples of the integer edit descriptor on output.

Table 41 **I edit descriptor: output examples**

Descriptor	Internal value	Output
`I4`	+452.25	*b*`452`
`I2`	+6234	`**`
`I3`	-11.92	`-11`
`I5`	-52	*bb*`-52`
`I10`	123456.5	*bbbb*`123456`
`I6.3`	3	*bbb*`003`
`I3.0`	0	*bbb*
`I3`	0	*bb*`0`

L (Logical) edit descriptor

The `L` edit descriptor defines a field for logical data. Its syntax is:

`[`*r*`]L[`*w*`]`

where:

r is a positive integer constant, specifying the repeat factor.

w is a positive integer constant, specifying the field width.

The I/O list item corresponding to an `L` edit descriptor must be of type logical, short logical, or byte.

Input

The field width is scanned for optional blanks followed by an optional decimal point, followed by `T` (or `t`) for true or `F` (or `f`) for false. The first nonblank character in the input field (excluding the optional decimal point) determines the value to be stored in the declared logical variable. It is an error if the first nonblank character is not `T`, `t`, `F`, `f`, or a period(.).

Table 42 provides examples of the logical edit descriptor on input.

Table 42 L edit descriptor: input examples

Descriptor	Input field	Value dtored
`L1`	`T`	`.TRUE.`
`L1`	`f`	`.FALSE.`
`L6`	`.TRUE.`	`.TRUE.`
`L7`	`.false.`	`.FALSE.`
`L2`	`.t`	`.TRUE.`
`L8`	*bbbb*`TRUE`	`.TRUE.`
`L3`	`ABC`	error: illegal character

Output

The character `T` or `F` is right-justified in the output field, depending on whether the value of the list item is true or false. Table 43 provides examples of the logical edit descriptor on output.

Table 43 **L edit descriptor: output examples**

Descriptor	Internal value	Output (logical)
L5	false	*bbbb*F
L4	true	*bbb*T
L1	true	T

O (Octal) edit descriptor

The O edit descriptor defines a field for octal data. It provides conversion between an external octal number and its internal representation.

The syntax for the octal edit descriptor is:

[*r*]O[*w*[.*m*]]

where:

r is a positive integer constant, specifying the repeat factor.

w is a positive integer constant, specifying the field width.

m is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

Input

The presence of too many digits for the integer variable (or list item) to receive produces undefined results. Legal octal digits are 0 through 7. Plus and minus signs are illegal.

Table 44 provides examples of the octal edit descriptors on input.

Table 44 **O edit descriptor: input examples**

Descriptor	Input field (octal)	Value stored (octal)
`O8`	`12345670`	12345670
`O2`	`77`	77
`O3`	`064`	64
`O8`	`45r`	error: illegal character

Output

List items may be of any type, though character variables are output only as the octal equivalent of their ASCII representation (no length descriptor).

If *w* is greater than the number of converted octal digits (including blanks between words but excluding leading zeroes), the octal digits are right-justified in the output field. If *w* is less than the number of converted octal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of octal digits in the entire output value. If *m* is set to 0, a zero value is output as all blanks.

Table 45 provides examples of the octal edit descriptors on output.

Table 45 **O edit descriptor: output examples**

Descriptor	Internal value	Output (Octal)
`O6`	`80`	*bbb*`120`
`O2`	`80`	`**`
`O14`	`-9`	*bbb*`37777777767`
`O11`	`32767`	*bbbbbb*`77777`
`O6.4`	`79`	*bb*`0117`

Descriptor	Internal value	Output (Octal)
O12	1.1	*bb*7743146315
O12	'A'	*b*101
O12	'ABC'	*b*101*b*102*b*103

P (scale factor) edit descriptor

The *k*P edit descriptor causes a scale factor of *k* to be applied to all subsequent F, D, E, EN, ES, and G edit descriptors in the format specification.

If the P edit descriptor does not precede an F, D, E, EN, ES, or G edit descriptor, it should be separated from other edit descriptors by a comma. If the P edit descriptor immediately precedes an F, D, E, EN, ES, or G edit descriptor, the comma is optional.

For example, the format specification

```
(3P, I2, F4.1, E5.2)
```

is equivalent to

```
(I2, 3PF4.1, E5.2)
```

When a format specification is interpreted, the scale factor is initially set to 0. When a P edit descriptor is encountered, the specified scale factor takes effect for the format specification and remains in effect until another P edit descriptor is encountered.

The effect of the scale factor differs for input and output as follows:

Input

If the value in the input field does not have an exponent, the internal number is equal to the field value multiplied by 10-k. If the value in the input field has an exponent, the scale factor has no effect. See Table 46 for examples of the scale factor on input.

Output

The scale factor has no effect on the EN, ES, F and G (interpreted as F) edit descriptors. For the D, E, and G (interpreted as E) edit descriptors, the value of the list item is multiplied by 10k as it is output but the exponent part is decreased by *k*.

The value specified for the scale factor (*k*) must be in the range:

`-`*d* `<` *k* `< (`*d* `+ 2)`

where:

d is the number of digits in the fractional part of the number being written.

k is a signed integer that specifies the scale factor.

Table 46 provides examples of the scale factor on output.

Table 46 P edit descriptor: input and output examples

Format specification	Input field	Internal value	Output
`(-2PG15.5)`	`1.97E-4`	1.97×10^{-4}	*bbbbb*`.00197E-01`
`(2P, F15.5)`	`27.982`	.2798199	*bbbbbbb*`27.98200`
`(2P,ES15.5)`	`3518.`	35.18	*bbbb*`3.51800E+01`
`(-2P,EN15.5)`	`7.91E+5`	7.91×10^{5}	*bb*`791.00000E+03`
`(-2PE15.5)`	`.17694`	17.694	*bbbbb*`.00177E+04`

When part or all of a format specification is repeated, the current scale factor is not changed until another scale factor is encountered.

Q (bytes remaining) edit descriptor

The `Q` edit descriptor is an HP extension that returns the number of bytes remaining to be read in the input record, placing the result into the corresponding integer variable in the I/O list. The return value can be used to control the remaining input items.

The `Q` edit descriptor is valid on input only; it is ignored on output. It can be used for reading formatted, sequential, and direct-access files. The following program segment reads variable-length strings from a sequential file:

```
CHARACTER(LEN=80) :: string
INTEGER :: n, i
...
READ (11,'(Q,80A1)') n, (string (i:i), i=1, n)
```

For information about the `Q`*w.d* edit descriptor for editing real data, see “D, E, EN, ES, F, G, and Q (real) edit descriptors” on page 219.

S, SP, and SS (plus sign) edit descriptors

The S, SP, and SS edit descriptors control printing of the plus sign character in numeric output. The default behavior of HP Fortran 90 is not to print the plus sign. However, an SP edit descriptor in the format specification causes the plus sign to appear in any subsequent numeric output where the value is positive. The SS descriptor suppresses the plus sign in subsequent numeric output. The S edit descriptor restores the default behavior.

The sign edit descriptors have no effect on input.

T, TL, TR, and X (tab) edit descriptors

The tab edit descriptors position the cursor on the input or output record. Their syntax is:

T*n*

TL*n*

TR*n*

*n*X

where:

n	is a positive integer constant, specifying the number of column positions to skip for positioning within the current output or input record.

The T edit descriptor references an absolute column number, while the descriptors TL and TR reference a relative number of column positions to the left (TL) or right (TR) of the current cursor position. Note that the TR descriptor is identical to the X edit descriptor.

Z (hexadecimal) edit descriptor

The Z edit descriptor defines a field for hexadecimal data. This descriptor provides for conversion between an external hexadecimal number and its internal representation.

The syntax for the hexadecimal edit descriptor is:

[*r*]Z[*w* [.*m*]]

where:

r	is a positive integer constant, specifying the repeat factor.
w	is a positive integer constant, specifying the field width.
m	is a nonnegative integer constant, specifying the minimum number of digits that must be in the field and forcing leading zeroes as necessary up to the first nonzero digit. The *m* value is ignored on input. If *m* is not specified, a default value of 1 is assumed. If *m* is larger than *w*, the field is filled with *w* asterisks.

Input

Variables to receive hexadecimal input must be of type integer. Legal hexadecimal digits are `0` through `9`, and `A` through `F` (or `a` through `f`). Nonleading blanks are ignored, unless the file is opened with `BLANK='ZERO'`. If the file is opened with `BLANK='ZERO'`, nonleading blanks are treated as zeroes. For more information about the `BLANK=` specifier see "OPEN" on page 379. Plus and minus signs, commas, or any other symbols are neither permitted on input nor printed on output. The presence of too many digits for the integer variable (or list item) produces undefined results.

Table 47 provides examples of the hexadecimal edit descriptor on input.

Table 47 Z edit descriptor: input examples

Descriptor	Input field (hexadecimal)	Value stored (hexadecimal)
`Z4`	`FF3B`	FF3B
`Z4`	`fFfF`	FFFF
`Z2`	`ABCD`	AB
`Z3`	`1.1`	error: illegal character

Output

List items may be of any type, though character variables are output only as the hexadecimal equivalent of their ASCII representation (without a length descriptor). If *w* is greater than the number of converted hexadecimal digits (excluding leading zeroes), the hexadecimal digits are right-justified in the output field. If *w* is less than the number of converted hexadecimal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output in twos complement form, their high-order bits are nonzero and cause the field to be filled with asterisks when *w* is less than the number of hexadecimal digits in the entire output value. If *m* is set to 0, a zero value is output as all blanks.

The field width required to fully represent the hexadecimal value of an item is twice its size in bytes. For example, a `CHARACTER*12` item would require a field width of 24 characters.

Table 48 provides examples of the hexadecimal edit descriptor on output.

Table 48 **Z edit descriptor: output examples**

Descriptor	Internal value	Output
Z2	27	1B
Z6.4	27	*bb*001B
Z	'A'	*b*41
Z8	'ABCD'	41424344
Z8	1.1	3F8CCCCD

Embedded format specification

A format specification can be embedded in a data transfer statement as a character expression. Parentheses are included in the expression, and the first nonblank character must be a left parenthesis. The matching right parenthesis must also be in the expression. A list of edit descriptors appears between the parentheses. Any characters appearing after the matching right parenthesis are ignored.

If the character expression is a character constant, it must be delimited by either apostrophes or quotation marks. If the character constant contains another character constant, the nested character constant must also be delimited. If the inner set of delimiters is the same as the outer set they must be doubled. Each of the following statements is correct and will produce the same results:

```
PRINT "('i = ', i2)", i
PRINT "(""i = "", i2)", i
PRINT '("i = ", i2)', i
PRINT '(''i = '', i2)', i
WRITE (6, "('i = ', i2)") i
```

If the character expression is an array element, the entire specification must be within that element. If the expression is a whole character array, the format specification is the concatenation of the array elements in array element order. (As an extension, HP Fortran 90 allows the use of an integer array to contain a format specification.)

The following illustrates the use of a character array to hold the format specification:

```
CHARACTER(LEN=6), DIMENSION(2) :: fspec
fspec(1) = '(F8.3,'
fspec(2) = ' I5)'
PRINT fspec, fvar, ivar
```

If the value of `fvar` is 12.34567 and `ivar` is 123, the output would be:

*bb*12.346*bb*123

Nested format specifications

A format specification can include a nested format specification (another set of edit descriptors, enclosed in parentheses). You can also precede the nested format specification with a repeat factor, as in the following example:

```
(1H , 2(I5, F10.5))
```

This is equivalent to:

```
(1H , I5, F10.5, I5, F10.5)
```

Each nested specification is known as a group at nested level *n*. The value of *n* begins at 1. For each successive level of nesting, *n* is incremented by 1. Each group at nested level 1 can contain one or more groups at nested level 2, and so on.

For example:

```
(E9.3,I6,(2X,I4))
```

contains one group at nested level 1.

```
(L2,A3/(E10.3,4(A2,L4)))
```

has one group at nested level 1 and one at nested level 2.

```
(A,(3X,(I2,(A3)),I3),A)
```

contains one group at nested level 1, one at level 2, and one at level 3.

A nested format specification can be preceded by a repeat specification. For example, the following input record

*b*26*b*6.4336*b*373.86*b*39*bb*49.79*bb*4*bbb*4395.4972

could be accessed with the following `FORMAT` statement:

```
10 FORMAT (I3,F7.4,2(F7.2,I3),F12.4)
```

The list of variables following `READ` statement corresponds to the preceding `FORMAT` statement:

```
READ 10,i,a,b,j,d,k,f
```

The `READ` statement would read values for `i` and `a`; repeat the nested format specification `F7.2,I3` twice to read values for `b`, `j`, `d`, and `k`; and, finally, read a value for `f`.

Format specification and I/O data list

A formatted I/O statement references each item in an I/O list, and the corresponding format specification is scanned to find a format descriptor for each item. As long as an item is matched to an edit descriptor, normal execution continues.

If there are more edit descriptors than list items, format control terminates with the last list item. If there are fewer edit descriptors than list items, the following three steps are performed:

1. The current record is terminated.
2. A new record is started.
3. Format control is returned to the format specification based upon the following hierarchy:
   a. Control returns to the repeat specification for the rightmost group at nested level 1. For information about nested levels, see “Nested format specifications” on page 238.
   b. If no repeat specification exists in the rightmost group at nested level 1, control returns to the group itself.
   c. If there is no group at nested level 1, control returns to the first descriptor in the format specification.

Table 49 provides examples showing how control is returned to the format specification in different circumstances.

Table 49 Format control and nested format specifications

Format specification	Control returns to:	Explanation
`(I5,2(3X,I2,(I4)))`	`2(3X,I2,(I4))`	The rightmost group at nested level 1 is `3X,I2,(I4)`. Control returns to the repeat specifier for this group.
`(F4.1,I2)`	`(F4.1,I2)`	There is no group at nested level 1. Control returns to the first descriptor in the format specification.
`(A3,(3X,I2),4X,I4)`	`(3X,I2),4X,I4`	Control returns to the group at nested level 1.

10 HP Fortran 90 statements

This chapter describes the HP Fortran 90 statements and attributes, arranged in alphabetical order. The descriptions provide syntax information, applicable rules and restrictions, and examples.

The following descriptions for specific type declarations are located in this chapter. Generic type declaration information is described in "Type declaration for intrinsic types" on page 24:

- `BYTE`
- `CHARACTER`
- `COMPLEX`
- `DOUBLE COMPLEX`
- `DOUBLE PRECISION`
- `INTEGER`
- `LOGICAL`
- `REAL`
- `RECORD`
- `TYPE(`*type-name*`)`

This chapter *does not* describe the following:

- Assignment statements (instead, see "Assignment" on page 95)
- Statement functions (instead, see "Statement functions" on page 137)
- Constructs (instead, see "Data types and data objects" on page 21)

Attributes

Table 50 lists all the attributes that an HP Fortran 90 entity can have and indicates their compatibility. If the box at the intersection of two attributes contains a check mark, the attributes are mutually compatible and can be held simultaneously by an entity. The attributes are referred to throughout this chapter as well as in the rest of the book.

Table 50 **Attribute compatibility**

	ALLOCATABLE	AUTOMATIC	DIMENSION	EXTERNAL	Initialization	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	STATIC	TARGET	VOLATILE
ALLOCATABLE	✓	✓	✓								✓	✓	✓		✓	✓
AUTOMATIC	✓	✓	✓							✓					✓	✓
DIMENSION	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
EXTERNAL				✓				✓			✓	✓				
Initialization			✓		✓				✓		✓	✓	✓	✓	✓	✓
INTENT			✓			✓		✓							✓	✓
INTRINSIC							✓				✓	✓				
OPTIONAL			✓	✓		✓		✓		✓					✓	✓
PARAMETER			✓		✓				✓		✓	✓				
POINTER		✓	✓					✓		✓	✓	✓	✓	✓		✓
PRIVATE	✓		✓	✓	✓		✓		✓	✓	✓		✓	✓	✓	✓
PUBLIC	✓		✓	✓	✓		✓		✓	✓		✓	✓	✓	✓	✓
SAVE	✓		✓		✓					✓	✓	✓	✓	✓	✓	✓

	ALLOCATABLE	AUTOMATIC	DIMENSION	EXTERNAL	Initialization	INTENT	INTRINSIC	OPTIONAL	PARAMETER	POINTER	PRIVATE	PUBLIC	SAVE	STATIC	TARGET	VOLATILE
STATIC			✓		✓					✓	✓	✓	✓	✓	✓	✓
TARGET	✓	✓	✓		✓	✓		✓			✓	✓	✓	✓	✓	✓
VOLATILE	✓	✓	✓		✓	✓		✓		✓	✓	✓	✓	✓	✓	✓

NOTE `AUTOMATIC`, `STATIC`, and `VOLATILE` may be specified in a statement of the same name but not as attributes in a type declaration statement.

Statements and attributes

The remainder of this chapter describes all of the statements and attributes that you can use in an HP Fortran 90 program. The statement and attribute descriptions are listed in alphabetical order. For general information about statements—including the order in which statements must appear in a legal program—see “Statements” on page 11.

ACCEPT (extension)

Reads from standard input.

Syntax

The syntax of the `ACCEPT` statement can take one of two forms:

- Formatted and list-directed syntax:

  `ACCEPT` *format* `[,` *input-list* `]`

- Namelist-directed syntax:

  `ACCEPT` *name*

format

is one of the following:

- An asterisk (`*`), specifying list-directed I/O.
- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification.

input-list

is a comma-separated list of data items. The data items can include variables and implied-`DO` lists.

name

is the name of a namelist group, as previously defined by a `NAMELIST` statement. Using this syntax, the `ACCEPT` statement accepts data from standard input and transfers it to the namelist group. To perform namelist-directed I/O with a connected file, you must use the `READ` statement and include the `NML=` specifier.

Description

The `ACCEPT` statement is an HP Fortran 90 extension and is provided for compatibility with other versions of Fortran. The standard `READ` statement performs the same function, and standard-conforming programs should use it.

The ACCEPT statement transfers data from standard input to internal storage. (Unit 5 is preconnected to the HP-UX standard input.) The ACCEPT statement can be used to perform formatted, list-directed, and namelist-directed I/O only.

To read data from a connected file, use the READ statement.

Examples

The following example of the ACCEPT statement reads an integer and a floating-point value from standard input, using list-directed formatting:

```
INTEGER :: i
REAL :: x
ACCEPT *, i, x
```

Related statements

FORMAT, NAMELIST, PRINT and READ

Related concepts

For related information, see the following:

- “List-directed I/O” on page 179
- “Implied-DO loop” on page 194
- “Embedded format specification” on page 237

ALLOCATABLE (statement and attribute)

Declares an allocatable array with deferred shape.

Syntax

The syntax of a type declaration statement with the `ALLOCATABLE` attribute is:

type`,` *attrib-list* `::` *entity-list*

type

is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE(`*type-name*`)`, etc.), as described in Chapter 3, “Data types and data objects,” on page 21.

attrib-list

is a comma-separated list of attributes including `ALLOCATABLE` and optionally those attributes compatible with it, namely:

DIMENSION	PUBLIC	TARGET
PRIVATE	SAVE	

entity-list

is a comma-separated list of entities. Each entity is of the form:

array-name [(*deferred-shape-spec-list*)]

If (*deferred-shape-spec-list*) is omitted, it must be specified in another declaration statement.

array-name

is the name of an array being given the attribute `ALLOCATABLE`.

deferred-shape-spec-list

is a comma-separated list of colons, each colon representing one dimension. Thus the rank of the array is equal to the number of colons specified.

The syntax of the `ALLOCATABLE` statement is:

`ALLOCATABLE [::]` *array-name* `[(`*deferred-shape-spec-list*`)]`

`[,`*array-name* `[(`*deferred-shape-spec-list*`)]]...`

If `(`*deferred-shape-spec-list*`)` is omitted from the `ALLOCATABLE` statement, it must be specified in another declaration statement, such as a type or `DIMENSION` statement.

The `ALLOCATED` intrinsic inquiry function is described in "ALLOCATED(ARRAY)" on page 493. It can be used to determine whether an allocatable array is currently allocated.

Description

The `ALLOCATABLE` attribute or statement is used to declare an array whose extents in all its dimensions will be specified when an `ALLOCATE` statement is executed at run-time; for this reason it is known as "deferred-shape". When an allocatable array is declared, only its name and rank are given.

Examples

The following statements declare a rank-one deferred-shape array and illustrate its use with different extents.

```
! mls is deferred shape.
INTEGER, ALLOCATABLE :: mls(:)
ALLOCATE (mls (3))       ! Allocate 3 elements.
DEALLOCATE (mls)         ! mls is no longer allocated
ALLOCATE (mls (-n:n))    ! Allocate with different extent
```

Related statements

`ALLOCATE` and `DEALLOCATE`

Related concepts

See "Allocatable arrays" on page 59 for more information about allocatable arrays and the conditions applying to their use.

Array pointers provide a more general mechanism for the manipulation of deferred-shape arrays; see "Array pointers" on page 59.

ALLOCATE

Provides storage space for allocatable arrays and pointer targets.

Syntax

`ALLOCATE (`*allocation-list*`[, STAT=` *scalar-integer-variable*`])`

allocation-list

is a comma-separated list of *allocation*.

allocation

is *allocate-object* `[(`*allocate-shape-spec-list*`)]`.

allocate-object

is *variable-name* or *derived-type-component*. Each *allocate-object* must be an allocatable array or a pointer.

allocate-shape-spec-list

is a comma-separated list of allocate-shape-spec.

allocate-shape-spec

is `[`*lower-bound*`:]`*upper-bound*. The bounds in an *allocate-shape-spec* must be scalar integer expressions.

`STAT=`*scalar-integer-variable*

returns the error status after the statement executes. If given, it is set to zero if the statement successfully executed, and to one of the following nonzero values if an error occurred:

`1` Error occurred after the array was allocated; for example, an attempt to allocate a previously allocated array.

`2` Dynamic memory allocation failure (memory not available) or invalid size (array too large).

`3` Errors of both types 1 and 2 have occurred. This kind of an error can only occur if the same ALLOCATE statement is used to allocate more than one array, and both kinds of errors occur.

If there is no *scalar-integer-variable*, the occurrence of an error causes the program to terminate.

Description

The ALLOCATE statement creates space for allocatable arrays and targets for variables (scalars or arrays) with the POINTER attribute. The ALLOCATE and DEALLOCATE statements give the user the ability to manage space dynamically at execution time.

For allocatable arrays, an error occurs when an attempt is made to allocate an already allocated array or to deallocate an array that is not allocated. The ALLOCATED intrinsic function may be used to determine whether an allocatable array is allocated.

A pointer can be associated with a target, either with the pointer assignment statement or by use of the ALLOCATE statement. It is not an error to allocate an already associated pointer; its old target connection is replaced by a connection to the newly allocated space. However, if the previous target was allocated and no other pointer became associated with it, the space is no longer accessible.

Examples

In the following example, a complex array with the POINTER attribute is declared. Target space is allocated to it at run-time, the amount being determined by two integer values read in. Later in the program, the space is recovered by use of the DEALLOCATE statement.

```
COMPLEX, POINTER :: hermitian (:, :)
READ *, m, n
ALLOCATE (hermitian (m, n))
DEALLOCATE (hermitian, STAT = ierr)
```

In the next example, a real allocatable array is declared. The amount of space allocated to it depends on how much is available.

```
! Rank-2 allocatable array
REAL, ALLOCATABLE :: intense(:,:)

CALL init_i_j(i, j)
DO
   ALLOCATE (intense(i, j), STAT = ierr4)
   ! ierr4 will be positive if there is not enough space to
   !   allocate this array
   IF (ierr4 == 0) EXIT
   i = i/2; j = j/2
END DO
```

The derived type `node` in the next example is the basis of a binary tree structure. It consists of a real value component (`val`) and two pointer components, `left` and `right`, both of type `node`. The variable `top` (of type `node`) is declared, and space is allocated for targets for the pointers `top%left` and `top%right`.

The ALLOCATE and DEALLOCATE statements and pointer variables of type node make it possible to allocate space for nodes in such a tree structure, traverse it as required, and then recover the space when it is no longer needed.

```
TYPE node
   REAL val
   TYPE(node), POINTER :: left, right    ! Pointer components
END TYPE node
TYPE(node) top
ALLOCATE (top % left, top % right)
```

In the final example, two CHARACTER arrays, para and key, are declared with the POINTER attribute. para is allocated space; key is made to point at a section of para.

```
! Pointers to char arrays
CHARACTER, POINTER :: para(:), key(:)

CALL init_k_m(k, m)
ALLOCATE (para(1000))
key => para (k : k + m)
```

Related statements

ALLOCATABLE (statement and attribute), DEALLOCATE, NULLIFY, and POINTER (statement and attribute)

Related concepts

For related information, see the following:

- The descriptions of the ALLOCATED and ASSOCIATED intrinsics in Chapter 11, “Intrinsic procedures,” on page 475
- “Pointers” on page 47

ASSIGN

Assigns statement label to integer variable.

Syntax

`ASSIGN` *stmt-label* `TO` *integer-variable*

stmt-label

is the statement label for an executable statement or a `FORMAT` statement in the same scoping unit as the `ASSIGN` statement.

integer-variable

is a scalar variable of the default integer type. It cannot be a field of a derived type or record, or an array element.

Description

Once a variable is defined by an `ASSIGN` statement, it can be used in an assigned `GO TO` statement or as a format specifier in an input/output statement. It should not be used in any other way.

A variable that has been assigned a statement label can be reassigned another label or an integer value. If *integer-variable* is subsequently assigned an integer value, it no longer refers to a label.

Examples

```
   ASSIGN 20 TO last1
   GO TO last1
   ...
   ! ASSIGN used with FORMAT statement
   ASSIGN 10 TO form1
10 FORMAT(F6.1,2X,I5/F6.1
   READ(5,form1)sum,k1,ave1
20 ...
```

Related statements

`GO TO` (assigned)

Related concepts

For related information, see the following:

- “Statement labels” on page 10
- “Assigned GO TO statement” on page 115

AUTOMATIC (extension)

Makes procedure variables and arrays automatic.

Syntax

```
AUTOMATIC var-name-list
```

var-name-list

is a comma-separated list of names of variables and arrays to be declared as automatic. Array names may be followed by an optional *explicit-shape-spec*.

Description

The `AUTOMATIC` statement is provided as an HP extension.

If a variable or array declared within a procedure is declared as automatic, then there is one copy of it for each invocation of the procedure. Space is allocated on entry to the procedure and deallocated on exit. This is also the default for variables that do not have the `SAVE` or `STATIC` attribute, unless the `+save` option has been specified.

If it is required to have the *same* copy of a variable available to each invocation of the routine (for example, to keep a record of the depth of recursion), then the variable should have the `SAVE` attribute.

Note the following:

- The `AUTOMATIC` statement may only be used within a procedure.
- Local variables are `AUTOMATIC` by default.
- Arguments and function values are `AUTOMATIC`.
- Automatic variables may not appear in `EQUIVALENCE`, `DATA` or `SAVE` statements.
- The `AUTOMATIC` attribute is not the same as automatic arrays and automatic character strings.

Examples

```
AUTOMATIC r, s, u, v, w(10)
```

Related statements

`SAVE` and `STATIC`

Related concepts

For information about automatic and static variables, refer to the *HP Fortran 90 Programmer's Guide*.

BACKSPACE

Positions file at preceding record.

Syntax

The syntax of the BACKSPACE statement can take one of two forms:

- Short form:

  BACKSPACE *integer-expression*

- Long form:

  BACKSPACE (*io-specifier-list*)

integer-expression
: is the number of the unit connected to a sequential file.

io-specifier-list
: is a list of the following comma-separated I/O specifiers:

  [UNIT=] *unit*

  specifies the unit connected to an external file opened for sequential access. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*.

  ERR=*stmt-label*

  specifies the label of an executable statement to which control passes if an error occurs during statement execution.

  IOSTAT=*integer-variable*

  returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The BACKSPACE statement causes the external file connected to *unit* to be positioned just before the preceding record of the file. The file must be connected for sequential access.

Examples

The following statement causes the file connected to unit 10 to be positioned just before the preceding record:

```
BACKSPACE 10
```

The following statement causes the file connected to unit 17 to be positioned just before the preceding record. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in `ios`:

```
BACKSPACE (17, ERR=99, IOSTAT=ios)
```

Related statements

`ENDFILE`, `OPEN`, and `REWIND`

Related concepts

For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171, which lists example programs that use I/O. For information about I/O formatting, see Chapter 11, “Intrinsic procedures,” on page 475.

BLOCK DATA

Introduces a block data program unit.

Syntax

BLOCK DATA [*block-data-name*]

block-data-name

is an optional name. If a name is given in the END BLOCK DATA statement terminating a block data program unit, it must be the same as the *block-data-name* given in the BLOCK DATA statement introducing the program unit.

Description

A block data program unit is used to give initial values to variables in a named common blocks by means of DATA statements and must start with a BLOCK DATA statement. The block data program unit is an obsolescent feature of Fortran 90 and is effectively superseded by the module, as described in “Modules” on page 161.

As an extension, HP Fortran 90 allows blank—or unnamed—common blocks to be initialized.

Examples

The following block data program unit gives initial values to variables in the common blocks cb1 and cb2. All variables in each common block are specified completely.

```
BLOCK DATA
  REAL b(4)  DOUBLE PRECISION z(3)
  COMPLEX c
  COMMON /cb1/c,a,b /cb2/z,y
  DATA b, z, c /1.0, 1.2 ,2*1.3, 3*7.654321D0, (2.4,3.76)/
END
```

Related statements

COMMON, DATA, and END

Related concepts

The structure and syntax of the block data program unit is described in “Block data program unit” on page 169.

BUFFER IN (extension)

Provided for compatibility with the **BUFFER IN** Cray statement.

NOTE Asynchronous I/O with the BUFFER IN statements is not supported. HP Fortran 90 V2.0 supports these statements for synchronous I/O only.

Syntax

`BUFFER IN` *(unit, mode) (begin-loc, end-loc)*

unit

is a unit identifier (integer expression).

mode

is ignored.

begin-loc, end-loc

are symbolic names of the variables, arrays, or array elements that mark the beginning and end locations of the BUFFER IN operation. begin-loc and end-loc must be either elements of a single array (or equivalenced to an array) or members of the same common block.

Description

The `BUFFER IN` statement is an HP Fortran 90 extension that provides compatibility with the Cray `BUFFER IN` feature. The statement causes data to be transferred while allowing any subsequent statements to execute concurrently.

The `BUFFER IN` statement is provided as a porting aid for existing Cray code; it typically will not produce superior performance compared to conventional Fortran 90 I/O methods.

- Other Fortran I/O statements (i.e., `READ`, `WRITE`, `PRINT`, `ACCEPT`, and `TYPE`) cannot be used on the same unit as the `BUFFER IN` statement. Mixing the standard Fortran 90 I/O operations with `BUFFER IN` on the same logical unit number can confuse the input stream (`READ`) or corrupt the data file (`WRITE`).
- The `BACKSPACE` statement cannot be used with files that are capable of being transferred by the `BUFFER IN` statement. Such files are referred to as *pure-data* (unblocked) files.

Examples

The following program shows how to use the `BUFFER IN` and `BUFFER OUT` statements. The program must be compiled with the `+autodbl` option; see “Option Descriptions” on page 576.

```
PROGRAM bufferedIoTest
! buffered i/o example: compile with +autodbl
   INTEGER a(10)
   OPEN ( UNIT = 7, NAME = 'test.dat', FORM = 'UNFORMATTED' )
   a = (/ (i,i=1,10) /)                  ! initialize the array A
   BUFFER OUT ( 7, 0 ) ( a, a(10) )      ! write out A twice
   CALL unit  ( 7 )
   BUFFER OUT ( 7, 0 ) ( a, a(10) )
   CALL unit  ( 7 )
   ! now position the file 40 bytes (5 integer values) into the
   file
   CALL setpos ( 7, 5 )
   ! read the remainder of the 1st record, and half of the second
   BUFFER IN ( 7, 0 ) ( a, a(10) )
   WRITE(6,*) a
   CLOSE (7)
END PROGRAM bufferedIoTest
```

Related statements BUFFER OUT

BUFFER OUT (extension)

Provided for compatibility with Cray **BUFFER OUT** statement.

NOTE Asynchronous I/O with the BUFFER OUT statements is not supported. HP Fortran 90 V2.0 supports these statements for synchronous I/O only.

Syntax

```
BUFFER OUT  (unit, mode) (begin-loc, end-loc)
```

unit

is a unit identifier (integer expression).

mode

is ignored.

begin-loc, end-loc

are symbolic names of the variables, arrays, or array elements that mark the beginning and end locations of the BUFFER IN operation. begin-loc and end-loc must be either elements of a single array (or equivalenced to an array) or members of the same common block.

Description

The `BUFFER OUT` statement is an HP Fortran 90 extension that provides compatibility with the Cray `BUFFER OUT` feature. The statement causes data to be transferred while allowing any subsequent statements to execute concurrently.

The `BUFFER OUT` statement is provided as a porting aid for existing Cray code; it typically will not produce noticeably superior performance compared to conventional Fortran 90 I/O methods. In fact, the `BUFFER OUT` statement will always be slightly slower than unformatted fixed record length I/O.

- Other Fortran I/O statements (for example, `READ`, `WRITE`, `PRINT`, `ACCEPT`, and `TYPE`) cannot be used on the same unit as the `BUFFER OUT` statement. Mixing the standard Fortran 90 I/O operations with `BUFFER OUT` on the same logical unit number can confuse the input stream (`READ`) or corrupt the data file (`WRITE`).
- The `BACKSPACE` statement cannot be used with files that are capable of being transferred by the `BUFFER OUT` statement. Such files are referred to as *pure-data* (unblocked) files.

Examples

For an example of BUFFER IN, see "BUFFER IN (extension)" on page 257.

Related statements

BUFFER IN

BYTE (extension)

Declares entities of type integer.

Syntax

`BYTE [[,` *attrib-list*`] ::]` *entity-list*

attrib-list

is a comma-separated list of one or more of the following attributes:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name `[(` *array-spec* `)] [=` *initialization-expr*`]`

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds

initialization-expr

is a integer constant integer expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The BYTE statement is an HP extension that is used to declare the properties of entities. The entities can take values that are whole numbers and can be represented in one byte. It is equivalent to the `INTEGER(KIND=1)` statement.

The BYTE statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements. Note, however, that the BYTE statement does not have a kind parameter.

Example

The following are valid declarations:

```
BYTE i, j
BYTE :: k
BYTE, PARAMETER :: limit=120
! use an array constructor to initialize an array
BYTE, DIMENSION(4) :: bvec=(/1,2,3,4/)
! use slashes as initialization delimiters, an HP extension
BYTE b/12/, bb/27/  ! note, no double colon
```

Related statements

`INTEGER`

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80

CALL

Invokes a subroutine.

Syntax

CALL *subr-name*[([*subr-act-arg-spec-list*])]

subr-name

is the name of the subroutine being invoked.

actual-argument-list

is a comma-separated list of entities of the form:

[*keyword* =]*actual-argument*

actual-argument

is one of the following:

- *expression*
- *variable*
- *procedure-name*
- **label* or &*label*

keyword

is one of the dummy argument names of the subroutine being invoked. If any *keyword* is specified, the subroutine interface must be explicit.

Description

A CALL statement is used to invoke (call) a subroutine, and to specify actual arguments, if any. Execution of the subroutine begins with the first executable statement. The following sequence of events occurs when a CALL statement executes:

1. Actual arguments that are expressions are evaluated.
2. The actual arguments are associated with the corresponding dummy arguments.
3. Control transfers to the subroutine being called, and the subroutine executes.

4 Control returns from the subroutine, normally to the statement following the CALL statement, or to a statement label indicated by an alternate return argument—**label* or &*label*. (The & *label* form is provided as a compatibility extension and can be used in fixed source form only.)

A subroutine can call itself, directly or indirectly; in this case the keyword RECURSIVE must be specified in the SUBROUTINE statement of the subroutine definition.

The %VAL and %REF built-in functions are provided as HP extensions. They can be used to change argument-passing conventions calling a routine written in another language.

The only subroutine invocation other than by the CALL statement in Fortran 90 is through "defined assignment", where a defined type assignment operator that has been defined by means of a subroutine is used.

Examples

```
! Interface for subroutine draw
INTERFACE
   SUBROUTINE draw (x_start, y_start, x_end, y_end, form, scale)
      REAL x_start, y_start, x_end, y_end
      CHARACTER (LEN = 6), OPTIONAL :: form
      REAL, OPTIONAL :: scale
   END SUBROUTINE draw
END INTERFACE

! References to draw
! arguments given by position; optional argument scale omitted
CALL draw (5., -4., 2., .6, "DASHED")
! arguments given by keyword; optional argument form omitted
CALL draw (scale=.4, x_end=0., y_end=0., x_start=.5, y_start=.3)
```

Related statements

INTERFACE and SUBROUTINE

Related concepts

For related information, see the following:

- "Recursive reference" on page 131
- "Referencing a subroutine" on page 130
- "Arguments" on page 139
- "%VAL and %REF built-in functions" on page 148
- "Defined assignment" on page 157

CASE

Marks start of statement block in a CASE construct.

Syntax

```
CASE ( case-selector ) [ construct-name ]
```

case-selector

is a comma-separated list of ranges of values that are candidates for matching against the case index specified by the SELECT CASE statement. Each item in the list can take one of the following forms:

- *case-value*
- *low*:
- :*high*
- *low*:*high*
- DEFAULT

where:

case-value, *low*, and *high*

are scalar initialization expressions of type integer, character, or logical

DEFAULT

indicates the statement block to execute if none of the other CASE statements in the CASE construct produces a match.

construct-name

is the name given to the CASE construct.

Description

The CASE statement is used in a CASE construct to mark the start of a statement block. The CASE construct can consist of multiple blocks; at most, one is selected for execution. Selection is determined by comparing the case index produced by the SELECT CASE statement to the *case-selector* in each CASE statement. If a match is found, the statement block under the matching *case-selector* executes. A match between the case index (*c*) and *case-selector* is determined for each form of *case-selector*, as follows:

case-value

For integer and character types, a match occurs if *c* `.EQ.` *case-value*.

For logical types, a match occurs if *c* `.EQV.` *case-value*.

low:

For integer and character types, a match occurs if *c* `.GE.` *low*.

:*high*

For integer and character types, a match occurs if *c* `.LE.` *high*.

low : *high*

For integer and character types, a match occurs if *c* `.GE.` *low* `.AND.` *c* `.LE.` *high*.

`DEFAULT`

For integer, character, and logical types, a match occurs if no match is found with any other *case-selector* and `DEFAULT` is specified as a *case-selector*.

If `CASE DEFAULT` is not present and no match is found with any of the other `CASE` statements, none of the statement blocks within the `CASE` construct executes and execution resumes with the first executable statement following the `END SELECT` statement.

At most only one `DEFAULT` selector can appear within a `CASE` construct.

Each `CASE` statement must specify a unique value or range of values within a particular `CASE` construct. Only one match can occur, and only one statement block can execute.

All *case-selectors* and the case index within a particular `CASE` construct must be of the same type: integer, character, or logical. However, the lengths of character types can differ.

The colon forms—*low*`:`, `:`*high*, or *low*`:`*high*—are not permitted for a logical type.

Although putting the `CASE` statements in order according to range may improve readability, it is not necessary for correct or optimal execution of the `CASE` construct. In particular, `DEFAULT` can appear anywhere among the `CASE` statements and need not be the last.

CASE statements inside a named CASE construct need not specify *construct-name*; but if they do, the name they specify must match that of the SELECT CASE.

A CASE statement can have an empty statement block.

Examples

The following example considers a person's credits and debits and prints a message indicating whether a resulting account balance will be overdrawn, empty, uncomfortably small, or sufficient:

```
INTEGER :: credits, debits

SELECT CASE (credits - debits)
CASE (:-1)
  PRINT *, 'OVERDRAWN'
  CALL TRANSFERFUNDS
CASE (0)
  PRINT *, 'NO MONEY LEFT'
CASE (1:50)
  PRINT *, 'BALANCE LOW'
CASE (51:)
  PRINT *, 'BALANCE OKAY'
END SELECT
```

Related statements

SELECT CASE and END (construct)

Related concepts

The CASE construct is described in "CASE construct" on page 105.

CHARACTER

Declares entities of type character.

Syntax

`CHARACTER` [*char-selector*] `[[,` *attrib-list*`]` `::]` *entity-list*

char-selector

specifies the length and kind of the character variable. It takes one of the following forms:

- `([LEN=]`*len-spec*`[, KIND=`*kind-param*`])`
- `(`*len-spec*`, [KIND=]`*kind-param*`)`
- `(KIND=`*kind-param*`[, LEN=`*len-spec*`])`
- `*`*len-const* `[,]`
- `*(`*len-spec*`[) ,]`

where *kind-param* (if specified) must be 1, the default; *len-spec* is either an asterisk (*) or a specification expression; and *len-const* is an integer constant. In the last form, *len-param* is enclosed in parentheses, and the optional comma may be included only if the double colon does not appear in the type declaration statement. If *len-spec* evaluates to a negative value, a zero-length string is declared. If *len-spec* is unspecified, the default is 1.

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name[(*array-spec*)][**len-spec*][= *initialization-expr*]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, *len-spec* is either an asterisk (*) or a specification expression, and *initialization-expr* is a character constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The CHARACTER statement is used to declare the length and properties of character data. It is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

To indicate that the length of a character can vary, you may use an assumed character length parameter by specifying an asterisk (*) for *len-param*. The asterisk may be used only when doing the following:

- Declaring the type of a function. The function must not be an internal or module function, nor must it be array-valued, pointer-valued, or recursive.
- Declaring a dummy argument of a procedure.
- Declaring a named constant (see the PARAMETER statement).

Examples

The following are valid declarations:

```
CHARACTER c1, c2
CHARACTER(LEN=80) :: text(0:25)
CHARACTER(2, 1), PARAMETER :: limit='ZZ'
! initialize an array, using an array constructor
CHARACTER(4) :: response(3) = (/"Yes.",  "No!!", "Huh?"/)
! use slashes as initialization delimiters, an HP extension
CHARACTER*10 c1/'Tom'/,c2/'Jones'/  ! note, no double colon
```

The following are valid uses of the assumed length parameter:

```
CHARACTER(*) dummy_arg_name
CHARACTER(*), PARAMETER :: hello="Hi Sam"
CHARACTER(LEN=*), PARAMETER :: hello="Hi Sam"
```

Assuming that `c` is an ordinary variable and not the dummy argument to a procedure, the following declaration is an illegal use of the assumed length parameter:

```
CHARACTER*(*) c  ! illegal
```

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Character strings as automatic data objects” on page 37
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80
- “LEN(STRING)” on page 539

CLOSE

Terminates file connection.

Syntax

```
CLOSE ( io-specifier-list )
```

io-specifier-list

is a list of the following comma-separated I/O specifiers:

[UNIT=]*unit*

specifies the unit connected to an external file. *unit* must be a positive integer-valued expression. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*.

ERR=*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution. If neither IOSTAT= or ERR= is specified and an error occurs, the program aborts and a system error message is issued. *stmt-label* must be in the same scoping unit as the CLOSE statement with the ERR= specifier.

IOSTAT=*integer-variable*

returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred. If neither IOSTAT= or ERR= is specified and an error occurs, the program aborts and a system error message is issued.

STATUS=*character-expression*

specifies the state of the file after it is closed. *character-expression* can be one of the following arguments:

'KEEP' Preserve the file after it is closed (default).

'DELETE' Do not preserve the file after it is closed.

The STATUS= specifier is ignored if the file was opened as a scratch file. See “OPEN” on page 379 for a description of the OPEN statement.

Description

The CLOSE statement closes the file whose unit number was obtained from an OPEN statement. A CLOSE statement must contain a unit number and at most one each of the other I/O specifiers.

A CLOSE statement need not be in the same program unit as the OPEN statement that connected the file to the specified unit. If a CLOSE statement specifies a unit that does not exist or has no file connected to it, no action occurs.

Examples

The following examples illustrate different uses of the CLOSE statement. In the first example, the CLOSE statement closes the file connected to unit 10; after it is closed, the file will continue to exist, unless it was opened with the STATUS='SCRATCH' specifier:

```
CLOSE (10)
```

In the next example, after the file connected to unit 6 is closed, it will cease to exist:

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

The following code produces the same results as the previous example:

```
CHARACTER(LEN=6) cstat
cstat='delete'
CLOSE(UNIT=6,STATUS=cstat)
```

The following example closes the file connected to unit 5. If an error occurs, control is transferred to the executable statement labeled 100, and the error code is stored in the variable ios:

```
CLOSE(5,IOSTAT=ios,ERR=100)
```

Related statements

OPEN

Related concepts

For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171, which also lists example programs that use I/O.

COMMON

Specifies common blocks.

Syntax

COMMON [/[[*common-block-name*]]/] *object-list*

[,]/[*common-block-name*]/ *object-list*]...

common-block-name

is the name of a labeled common block.

object-list

is a comma-separated list of scalar variables, arrays, records, and derived-type objects. If an array is specified, it may be followed by an explicit-shape specification expression.

Description

The COMMON statement defines one or more storage areas to be shared by different program units. It also identifies the objects—that is, variables, arrays, records, and derived-type objects—to be stored in those areas. Objects in common that are shared by different program units are made accessible by storage association.

Each object following a common-block name is declared to be in that common block. If /*common-block-name*/ is omitted, all objects in the corresponding *object-list* are specified to be in blank common. It is also possible to declare variables in blank common by specifying two slashes without *common-block-name*. Consider the following examples:

```
!Declare variables a, b, c in blank common.
COMMON a, b, c

! Declare pay and time in blank common,
! and red in the named common block color.
COMMON pay, time, /color/red

! Variables a1 and a2 are in common block a; array x and variable
! are in blank common; and variable d is in common block c
COMMON/a/a1,a2,//x(10),y,/c/d
```

Any common block name or blank common specification can appear more than once in one or more COMMON statements within the same program unit. The variable list following each successive appearance of the same common block name is treated as a continuation of the list for that common block name. For example, the following COMMON statements:

```
COMMON a,b,c /x/y,x,d //w,r
COMMON /cap/hat,visor, //tax, /x/o,t
```

are equivalent to:

```
COMMON a,b,c,w,r,tax
COMMON /x/y,x,d,o,t
COMMON /cap/hat,visor
```

Unlike named common blocks, blank common can differ in size in different scoping units. However, blank common cannot be initialized.

As an extension, HP Fortran 90 saves all common blocks in static memory.

The following restrictions apply to the use of common blocks:

- All common block names must be distinct from subprogram names.
- The size of a named common block must be the same in all program units where it is declared. Note, however, that the size of blank common can differ.
- The following data items must not appear in a COMMON statement:
  - Dummy arguments in a subprogram
  - Functions, subroutines, or intrinsic functions
  - Pointees declared by Cray-style pointers
  - Variables accessible by use association
  - Automatic entities, including automatic character strings
  - Allocatable arrays
- Derived-type objects may appear in common if they have been defined with the SEQUENCE attribute.
- A variable can only appear in one COMMON statement within a program unit.
- Zero-sized common blocks are allowed. Zero-sized common blocks with the same name are storage associated.
- Array bounds in a COMMON statement must be constant specification expressions.
- A pointer may appear in common if it has the same type, type parameter, and rank in every instance of that common block.

Initializing common blocks

As an extension to the Standard, HP Fortran 90 allows common blocks to be initialized outside of a block data program unit; for example, in a subroutine. However, note that all data initialization for a given common block must occur in the same compilation unit.

HP Fortran 90 also allows blank—or unnamed—common to be initialized.

Common block size

The size of a common block is determined by the number and type of the variables it contains. In the following example, the common block `my_block` takes 20 bytes of storage: `b` uses 8 (2 bytes per element) and `arr` uses 12 (4 bytes per element):

```
INTEGER(2) b(4)
INTEGER(4) arr(3)
COMMON /cb/b, arr
```

Data space within the common area for arrays `b` and `arr` shown in this example is allocated as follows:

Bytes	Common block variables
0, 1, 2, 3	`b(1), b(2)`
4, 5, 6, 7	`b(3), b(4)`
8, 9, 10, 11	`arr(1)`
12, 13, 14, 15	`arr(2)`
16, 17, 18, 19	`arr(3)`

Allocation common block storage

Common block storage is allocated at link time. It is not local to any one program unit.

Each program unit that uses the common block must include a `COMMON` statement that contains the block name, if a name was specified. Variables assigned to the common block by the program unit need not correspond by name, type, or number of elements with those of any other program unit. The only consideration is the size of the common blocks referenced by the different program units. Correspondence between objects in different instances of the same common block is established by storage association.

Note the following HP Fortran 90: when types with different alignment restrictions are mixed in a common block, the compiler may insert padding bytes as necessary.

Examples

The following example illustrates how the same common block can be declared in different program units with different variables but the same size:

```
! common declaration for program unit 1
INTEGER i, j, k
COMMON /my_block/ i, j, k

! common declaration for program unit 2
INTEGER n(3)
COMMON /my_block/ n(3)
```

The variables `i`, `j`, and `k` in program unit 1 share the same storage with the array `n` in program unit 2: `i` in program unit 1 matches up with `n(1)` in program unit 2, `j` with `n(2)`, and `k` with `n(3)`.

Related statements

`EQUIVALENCE`

Related concepts

For information about data alignment, see Table 5 and “Alignment of derived-type objects” on page 44.

COMPLEX

Declares entities of type complex.

Syntax

`COMPLEX [`*kind-spec*`] [[, `*attrib-list*`] ::] `*entity-list*

kind-spec

is the kind type parameter that specifies the range and precision of the entities in *entity-list*. *kind-spec* takes the form:

([KIND=]*kind-param*)

where *kind-param* represents the kind of both the real and imaginary parts of the complex number. It can be a named constant or a constant expression that has the integer value of 4 or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

**len-param*

where *len-param* is the integer 8 or 16 (default = 8), which represents the size of the whole complex entity.

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is a complex constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The `COMPLEX` statement is used to declare the length and properties of data that are approximations to the mathematical complex numbers. A complex number consists of a real part and an imaginary part. A kind parameter (if specified) indicates the representation method.

The `COMPLEX` statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

As a portability extension, HP Fortran 90 allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If *array-spec* is specified, **len* may appear on either side of *array-spec*. If *name* appears with **len*, it overrides the length specified by *kind-spec*.

Examples

The following are valid declarations:

```
COMPLEX x, y
COMPLEX(KIND=8) :: z
COMPLEX,PARAMETER :: t1(2)=(/(3.2, 0), (.04, -1.1)/)
! initialize an array, using an array constructor
COMPLEX, DIMENSION(2) :: &
     cvec=(/(2.294, 6.288E-2), (-1.0096E7, 0)/)
! use slashes as initialization delimiters, an HP extension
COMPLEX cx/(2.294, 6.288E-2)/   ! note, no double colon
! the following declarations are equivalent; the second uses the
! HP length specification extension
COMPLEX(KIND = 8) x
COMPLEX(8) x*16
```

Related statements

`DOUBLE COMPLEX`

Related concepts

For related information, see the following:

- "Type declaration for intrinsic types" on page 24
- "Implicit typing" on page 28
- "Array declarations" on page 54
- "Array constructors" on page 71
- "Expressions" on page 80
- "KIND(X)" on page 537

CONTAINS

Introduces an internal procedure or a module procedure.

Syntax

```
CONTAINS
```

Description

The CONTAINS statement introduces an internal procedure or a module procedure, separating it from the program unit that contains it. The statement can be used in:

- A main program, external subprogram, or module subprogram; in each case, it precedes one or more internal procedures.
- A module, where it precedes any module procedures.

When a CONTAINS statement is present, at least one subprogram must follow it.

Examples

The first example illustrates CONTAINS introducing an internal subroutine. It also illustrates how the internal subroutine mechanism can provide an alternative to the FORTRAN 77 statement function mechanism.

```
PRINT *, double_real(6.6)
CONTAINS
   FUNCTION double_real (x); REAL x
     double_real = 2.0 * x
   END FUNCTION
END
```

The next example illustrates a main program with an internal procedure part.

```
PROGRAM electric      ! Program header
   REAL  current      ! Specification part
   current = 100.5    ! Execution part begins
   CALL compute_resistance( voltage, current, resistance )
   CONTAINS           ! Internal procedure part
      SUBROUTINE  compute_resistance( v, i, r )
         REAL  i
         r = v / i
      END SUBROUTINE
END PROGRAM electric
```

The third example is of a module that contains a module subprogram, which in turn contains an internal subprogram.

```
MODULE one
   CONTAINS
   SUBROUTINE two(x)     ! Module subprogram
      CONTAINS
      LOGICAL FUNCTION three(y)   !Internal subprogram
      END FUNCTION three
   END SUBROUTINE two
END MODULE one
```

Related statements

SUBROUTINE and FUNCTION

Related concepts

For related information, see the following:

- “Program units” on page 122
- “Internal procedures” on page 135
- “Module program unit” on page 161

CONTINUE

Establishes reference point within a program unit.

Syntax

```
CONTINUE
```

Description

The `CONTINUE` statement has no effect on program execution. Control passes to the next executable statement. The `CONTINUE` statement is generally used to mark a place for a statement label, especially when it occurs as the terminal statement of a FORTRAN 77-style `DO` loop.

Examples

```
count = 0
DO 20 i = 1, 10
   count = count + i
20 CONTINUE
PRINT *, count
```

Related statements

`DO`

Related concepts

For related information, see the following:

- “DO construct” on page 107
- “Flow control statements” on page 112

CYCLE

Interrupts current iteration of a DO loop.

Syntax

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

do-construct-name

is the name of a DO construct that must contain this CYCLE statement.

Description

The CYCLE statement is used to control the execution of a DO loop. When it executes, it interrupts a currently executing loop iteration and passes control to the next iteration, making the appropriate adjustments to the loop index. It may be used with either the DO construct or the FORTRAN 77-style DO loop.

A CYCLE statement belongs to a particular DO loop. If *do-construct-name* is not given, the CYCLE statement resumes the immediately enclosing DO loop. If *do-construct-name* is given, the CYCLE statement resumes an enclosing named DO loop with the same name.

Examples

The following example uses the CYCLE statement to control a bubble sort:

```
LOGICAL :: swap
INTEGER :: i, j
outer: DO i = 1, n-1
  swap = .FALSE.
  inner: DO j = n, i+1, -1
    IF (a(j) >= a(j-1)) CYCLE inner
    swap = .TRUE.
    atmp = a(j)
    a(j) = a(j-1)
    a(j-1) = atmp
  END DO inner
  IF (.NOT. swap) EXIT outer
END DO outer
```

Related statements

DO and EXIT

Related concepts

For related information, see the following:

- “DO construct” on page 107
- “Flow control statements” on page 112

DATA

Initializes program variables.

Syntax

```
DATA var-list1 / val-list1 / [[,]var-list2 / val-list2 /]...
```

var-list

is a comma-separated list of entities, including the following:

- A variable name
- An array name
- An array triplet section; for example:
- points(1:10:2)
- An array element reference; for example:
- scores(0)
- A substring name; for example:
- name(1:10)
- An implied-DO loop; for example:
- ((matrix(i,j),i=0,5),j=5,10)
- An object of a derived type
- A component of a derived-type object

The following cannot appear in *var-list*:

- Pointer-based variables
- Records and record field references. However, you can initialize a record's fields in the record's structure definition. See "RECORD (extension)" on page 420.
- Automatic objects, including automatic character strings
- Dummy arguments
- Allocatable arrays: that is, arrays declared with a specified rank, but no specified bounds within each dimension

- The result variable of a function
- Objects made available by use or host association
- Procedure names

val-list

is a list of constant values, separated by commas. Each constant in the list represents a value to be assigned to the corresponding variable in *var-list*. A constant value can be optionally repeated by preceding the constant with a repetition factor. The syntax of a repeated constant is:

*r***val*

where *r* is a positive integer specifying the number of times that *val*, the constant value, is to be specified.

Description

The `DATA` statement initializes variables local to a program unit before the program unit begins execution. Initialization occurs as follows:

The var-list is expanded to form a sequence of scalar variables, and the val-list is expanded to form a sequence of scalar constants. The number of items in each expanded sequence must be the same, and there must be a one-to-one correspondence between the items in the two expanded lists. The variables in the expanded sequence of var-list are initialized on the basis of the correspondence.

If *var-list* contains an array name, the expanded sequence of constants must contain a constant for every element in the array.

A zero-sized array or an implied-`DO` list with an iteration count of zero in *var-list* contributes no variables to the expanded sequence of variables. However, a zero-length character variable does contribute a variable to the list.

If a constant is of any numeric or logical type, the corresponding variable can be of any numeric type. If an object is of derived type, the corresponding constant must be of the same type. If the type of the constant does not agree with the type of the variable, type conversion is performed, as described in Table 14.

Variables can be initialized with binary, octal, or hexadecimal constants.

A variable or array element must not appear in a `DATA` statement more than once. If two variables share the same storage space through an `EQUIVALENCE` statement, only one can appear in a `DATA` statement. If a substring of a character variable or other array element appears in a DATA statement, no overlapping substring (including the entire variable or array element) can appear in any DATA statement.

The length of a character constant and the declared length of its corresponding character variable need not be the same. If the constant is shorter than the variable, blank characters are placed in the remaining positions. If the constant is longer than the variable, the constant is truncated from the right until it is the same length as the variable

If a subscripted array element appears in *var-list*, then the subscript must be a specification expression.

DATA statements can be interspersed among executable statements. However, they initialize prior to runtime and, therefore, cannot be used as executable assignment statements.

Fortran 90 extensions

A variable of type other than integer may be initialized with a binary, octal, or hexadecimal constant. The data type for a constant is determined from the type of the corresponding variable. The size (in bytes) of the variable determines how many digits of the octal or hexadecimal constant are used. If the constant lacks enough digits, the value is padded on the left with zeros. If the constant has too many digits, it is truncated on the left.

An integer, binary, octal, or hexadecimal constant can initialize a character variable of length one, as long as the value of the constant is in the range 0 to 255.

Examples

The following DATA statement initializes integer, logical, and character variables:

```
INTEGER i
LOGICAL done
CHARACTER(LEN=5) prompt
DATA i, done, prompt/10, .FALSE., 'Next?'/
```

The next DATA statement specifies a repetition factor of 3 to assign the value of 2 to all three elements of array `i`:

```
INTEGER, DIMENSION(3) :: i
DATA i/3*2/
```

The next DATA statement uses two nested implied-DO loops to assign the literal value X to each element of an array of 50 elements, `k(10,5)`:

```
CHARACTER, DIMENSION(10,5) :: k
DATA ((k(i,j),i=1,10),j=1,5)/50*'X'/
```

Related statements BYTE, CHARACTER, COMPLEX, DOUBLE COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, and REAL

Related concepts For related information, see the following:

- "Initialization expressions" on page 90
- "Assignment statement" on page 95
- "Implied-DO loop" on page 194

DEALLOCATE

Deallocates allocatable arrays and pointer targets.

Syntax

`DEALLOCATE (`*alloc-obj-list*`[, STAT=`*scalar-int-var*`])`

alloc-obj-list

is a comma-separated list of pointers or allocatable arrays.

`STAT=`*scalar-int-var*

returns the error status after the statement executes. If given, it is set to a positive value if an error is detected, and to zero otherwise. If there is no status variable, the occurrence of an error causes the program to terminate.

Description

The `DEALLOCATE` statement deallocates allocatable arrays and pointer targets, making the memory available for reuse. A specified allocatable array then becomes not allocated (as reported by the `ALLOCATED` intrinsic), while a specified pointer becomes disassociated (as reported by the `ASSOCIATED` intrinsic).

An error occurs if an attempt is made to deallocate an allocatable array that is not currently allocated or a pointer that is not associated. Errors in the operation of `DEALLOCATE` can be reported by means of the optional `STAT=` specifier.

You can deallocate an allocatable array by specifying the name of the array with the `DEALLOCATE` statement. You cannot deallocate a pointer that points to an object that was not allocated.

Some or all of a target associated with a pointer by means of the `ALLOCATE` statement can also be associated subsequently with other pointers. However, it is not permitted to deallocate a pointer that is not currently associated with the whole of an allocated target object.

Deallocation of a pointer target causes the association status of any other pointer associated with all or part of the target to become undefined. When a pointer is deallocated, its association status becomes disassociated, as if a `NULLIFY` statement had been executed.

Examples

The following example declares a complex array with the POINTER attribute. The ALLOCATE statement allocates target space to the array at run-time; the amount is determined by the input values to the READ statement. Later in the program, the DEALLOCATE statement will recover the space.

```
COMPLEX, POINTER :: hermitian (:, :)
...
READ *, m, n
ALLOCATE (hermitian (m, n))
...
DEALLOCATE (hermitian, STAT = ierr)
```

Related statements

ALLOCATABLE, ALLOCATE, NULLIFY, and POINTER

Related concepts

For related information, see the following:

- “Pointers” on page 47
- “Allocatable arrays” on page 59
- The descriptions of the ALLOCATED and ASSOCIATED intrinsics are described in Chapter 11, “Intrinsic procedures,” on page 475.

DECODE (extension)

Inputs formatted data from internal storage.

Syntax

`DECODE` (*count*, *format*, *unit*, *io-specifier-list*) [*in-list*]

count

is an integer expression that specifies the number of characters (bytes) to translate from character format to internal (binary) format. *cnt* must precede *format*.

format

specifies the format specification for formatting the data. *format* can be one of the following:

- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification.

format must be the second of the parenthesized items, immediately following *count*. Note that the keyword `FMT=` is not used.

unit

is the internal storage designator. It must be a scalar variable or array name. Assumed-size and adjustable-size arrays are not permitted. Note that *char-var-name* is not a unit number and that the keyword `UNIT=` is not used.

unit must be the third of the parenthesized items, immediately following *format*.

io-specifier-list

is a comma-separated list of I/O specifiers. Note that the unit and format specifiers are required; the other I/O specifiers are optional. The following I/O specifiers can appear in *io-specifier-list*:

ERR=*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

IOSTAT=*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.

in-list

is a comma-separated list of data items for input. The data items can include expressions and implied-DO lists.

Description

The DECODE statement is an HP extension that is provided for compatibility with other versions of Fortran. The internal-I/O capabilities of the standard READ statement provide similar functionality and should be used to ensure portability.

The DECODE statement translates formatted character data into its binary (internal) representation.

Examples

The following example program illustrates the DECODE statement:

```
PROGRAM decode_example
  CHARACTER(LEN=20) :: buf
  INTEGER i, j, k
  buf = 'XX1234  45 -12XXXXXX'
  DECODE (15,'(2X,3I4,1X)', buf) i, j, k
  ! The equivalent READ statement is:
  ! READ (buf, '(2X,3I4,1X)') i, j, k
  PRINT *, i, j, k
END PROGRAM decode_example
```

When compiled and executed, this program produces the following output:

```
 1234 45 -12
```

Related statements ENCODE and READ

Related concepts For related information, see the following:

- "Internal files" on page 174
- "Performing I/O on internal files" on page 176
- "Implied-DO loop" on page 194
- "Embedded format specification" on page 237

DIMENSION (statement and attribute)

Declares a variable to be an array.

Syntax

A type declaration statement with the DIMENSION attribute is:

type, DIMENSION (*array-spec*) [[, *attrib-list*]::] *entity-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE(*type-name*), etc.).

array-spec

is one of the following:

- *explicit-shape-spec-list*
- *assumed-shape-spec-list*
- *deferred-shape-spec-list*
- *assumed-size-spec*

explicit-shape-spec

is

[*lower-bound* :] *upper-bound*

lower-bound, *upper-bound*

are specification expressions.

assumed-shape-spec

is

[*lower-bound*] :

deferred-shape-spec

is

:

assumed-size-spec

is

[*explicit-shape-spec-list* ,] [*lower-bound* :] *

That is, *assumed-size-spec* is *explicit-shape-spec-list* with the final upper bound specified as *.

attrib-list

is a comma-separated list of attributes including DIMENSION and optionally those attributes compatible with it, namely:

ALLOCATABLE	PARAMETER	PUBLIC
INTENT	POINTER	SAVE
OPTIONAL	PRIVATE	TARGET

entity-list

is

object-name[(*array-spec*)]

If (*array-spec*) is present, it overrides the (*array-spec*) given with the DIMENSION keyword in *attribute-list*; see the example below.

The syntax of the DIMENSION statement is:

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

Description

An array consists of a set of objects called the array elements, all of the same type and type parameters, arranged in a pattern involving columns, and possibly rows, planes, and higher dimensioned configurations. The type of the array elements may be intrinsic or user-defined. In HP Fortran 90, an array may have up to seven dimensions. The number of dimensions is called the rank of the array and is fixed when the array is declared. Each dimension has an extent that is the size in that dimension (upper bound minus lower bound plus one). The size of an array is the product of its extents. The shape of an array is the vector of its extents in each dimension. Two arrays that have the same shape are said to be conformable.

It is not necessary for the keyword DIMENSION to appear in the declaration of a variable to give it the DIMENSION attribute. This attribute, as well as the rank, and possibly the extents and the bounds of an array, may be specified in the entity declaration part of any of the following statements:

- type declaration
- DIMENSION
- ALLOCATABLE
- COMMON
- POINTER
- TARGET

The *array-spec* (see **Syntax**, above) determines the category of the array being declared. “Array declarations” on page 54, describes these categories as:

- Explicit-shape array
- Assumed-shape array
- Assumed-size array
- Deferred-shape array

Examples

```
! These 2 declaration statements are equivalent.
REAL a (20,2), b (20,2), c (20,2)
REAL, DIMENSION (20,2) :: a, b, c

DIMENSION x(100), y(100)  ! x and y are 1-dimensional

! lower bounds specified for jj (if not given, they default to 1)
INTEGER jj (0:100, -1:1)

! l is a 4-dimensional, allocatable, deferred shape logical array
LOGICAL l
ALLOCATABLE l(:,:,:,:)

COMPLEX s         ! s has explicit shape and
TARGET :: s(10,2) !    the target attribute

DOUBLE PRECISION d
! d has 5 dimensions and is declared in common
COMMON /stuff/ d(2,3,5,9,8)

! arr1 is an adjustable array, arr2 an automatic array
```

```
SUBROUTINE calc(arr1, ib1, ib2)
REAL, DIMENSION (ib1, ib2) :: arr1, arr2

! arr3 is a deferred-shape array with the pointer attribute
REAL, POINTER, DIMENSION(:,:) :: arr3

! all three arrays have explicit shape; array specifier (10,10)
! overrides specifier (10,20) for tb declaration only
LOGICAL, DIMENSION(10,20) :: ta, tb(10,10), tc
```

Related statements

ALLOCATABLE, COMMON, POINTER, TARGET, TYPE, and the type declaration statements

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- Chapter 11, “Intrinsic procedures,” on page 475
- The following array-inquiry intrinsics described in Chapter 11:
  - LBOUND
  - RESHAPE
  - SHAPE
  - SIZE
  - UBOUND

DO

Controls execution of DO loop.

Syntax

[*construct-name* :] DO [*label*] [*loop-control*]

construct-name

is the name given to the `DO` construct. If *construct-name* is specified, an `END DO` statement must appear at the end of the `DO` construct and have the same *construct-name*.

label

is the label of an executable statement that terminates the `DO` loop. If you specify *label*, you can terminate the `DO` loop either with an `END DO` statement or with an executable statement; the terminating statement must include *label*. If you do not specify *label*, you must terminate the `DO` loop with the `END DO` statement.

loop-control

is information used by the `DO` statement to control the loop. It can take one of the following forms:

- *index* = *init*, *limit* [, *step*]
- `WHILE` (*logical-expression*)
- *loop-control* is omitted

In the first form, *index* is a scalar variable of type integer or real; *init*, *limit*, and *step* are scalar expressions of type integer or real. In the second form, *logical-expression* is a scalar logical expression. In the third form, *loop-control* is omitted. If you use the second or third form, you must terminate the `DO` loop with the `END DO` statement.

Description

The syntax of the `DO` statement allows for the following types of `DO` loops:

- Counter-controlled loop: a loop count is calculated that controls the number of times the block is executed, unless a prior exit occurs. A loop variable is incremented or decremented after each execution.

- While loop: a condition (*logical-expression*) is tested before each execution of the block; when it is false, execution ceases. An exit may occur at any time.
- Infinite loop: there is no *loop-control*; repeated execution of the block ceases only when an exit from the loop occurs.

When *label* is present in the `DO` statement, it specifies the label of the terminating statement of the `DO` loop. The terminating statement *cannot* be any of the following statements:

- `GO TO` (unconditional)
- `GO TO` (assigned)
- `IF` (arithmetic)
- `IF` (block)
- `ELSE` or `ELSE IF`
- `END`, `END IF`, `END SELECT`, or `END WHERE`
- `RETURN`
- `STOP`
- DO
- Any nonexecutable statement

Note, however, that the terminating statement can be an `IF` (logical) or an `END DO` statement.

To maintain compatibility with some older versions of Fortran, you can use the `+onetrip` compile-line option to ensure that every counter-controlled `DO` loop in the program executes at least once.

Extended-range DO loops

Extended-range `DO` loops—a compatibility extension—allow a program to transfer control outside the `DO` loop's range and then back into the `DO` loop. Extended-range `DO` loops work as follows: if a control statement inside a `DO` loop transfers control to a statement outside the `DO` loop, then any subsequent statement can transfer control back into the body of the `DO` loop.

For example, in the following code, the range of the `DO` loop is extended to include the statement `GOTO 20`, which transfers control back to the body of the `DO` loop:

```
   DO 50 i = 1, 10
20   n = n + 1
     IF (n > 10) GOTO 60
50 CONTINUE     ! normally, the range ends here
60 n = n + 100  ! this is the extended range,
   GOTO 20      ! which extends down to this line
```

Examples

The following `DO` construct displays the integers 1 through 10:

```
DO i = 1, 10
  WRITE (*, *) i
END DO
```

The next example is a FORTRAN 77-style `DO` loop that does the same as the preceding example:

```
   DO 50 i = 1, 10
     WRITE (*, *) i
50 CONTINUE
```

The following `DO` construct iterates 5 times, decrementing the loop index from 10 to 2:

```
DO i = 10, 1, -2
END DO
```

The following is an example of a `DO WHILE` loop:

```
DO WHILE (sum < 100.0)
    sum = sum + get_num(unit)
END DO
```

The following example illustrates the use of the `EXIT` statement to exit from a nested `DO` loop. The loops are named to control which loop is exited. Note that *loop-control* is missing from both the inner and outer loops, which therefore can be exited only by means of one of the `EXIT` statements:

```
outer:DO
  READ *, val
  new_val = 0
  inner:DO
    new_val = new_val + proc_val(val)
    IF (new_val >= max_val) EXIT inner
    IF (new_val == 0) EXIT outer
  END DO inner
END DO outer
```

The next DO construct never executes:

```
DO i = 10, 1
END DO
```

Related statements CONTINUE, CYCLE, END (construct), and EXIT

Related concepts For related information, see the following:

- “DO construct” on page 107
- “EXIT statement” on page 114

DOUBLE COMPLEX (extension)

Declares entities of type double complex.

Syntax

DOUBLE COMPLEX [[, *attrib-list*] ::] *entity-list*

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds

initialization-expr

is a complex constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The DOUBLE COMPLEX statement is an HP Fortran 90 extension that declares the properties of complex data that has greater precision than data of default type complex. The two parts of a double complex value are each a double precision value.

The DOUBLE COMPLEX statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements. Note however, that the DOUBLE COMPLEX statement does not have a kind parameter.

Examples

The following are valid declarations:

```
DOUBLE COMPLEX x, y
DOUBLE COMPLEX, PARAMETER :: t1(2)=(/(1.2, 0), (-1.01, 0.0009)/)
! use an array constructor to initialize a double complex array
DOUBLE COMPLEX, DIMENSION(2) :: dc_vec =  &
   (/(2.294D-8, 6.288D-4), (-4.817D4, 0)/)
! use slashes as initialization delimiters, an HP extension
DOUBLE COMPLEX dcx/(2.294D-8, 6.288D-4)/  ! note, no double colon
```

Related statements

COMPLEX

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80

DOUBLE PRECISION

Declares entities of type double precision.

Syntax

`DOUBLE PRECISION [ [,` *attrib-list*`] ::]` *entity-list*

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name `[(` *array-spec* `)] [=` *initialization-expr*`]`

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds

initialization-expr

is a real constant expression that can be evaluated at compile time. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The DOUBLE PRECISION statement is used to declare the properties of real data that has greater precision than data of default type real. By default, the DOUBLE PRECISION statement is equivalent to the REAL(KIND=8) statement.

The DOUBLE PRECISION statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements. Note, however, that the DOUBLE PRECISION statement does not have a kind parameter.

Examples

The following are valid declarations:

```
DOUBLE PRECISION x, y
DOUBLE PRECISION, PARAMETER :: pi=3.1415927D0
! use an array constructor to initialize a double precision array
DOUBLE PRECISION, DIMENSION(4) :: dp_vec=  &
    (/4.7D0, 5.2D0, 3.3D0, 2.9D0/)
! use slashes as initialization delimiters, an HP extension
DOUBLE PRECISION dp1/5.28D0/, dp2/72.3D0/  ! note, no double
colon
```

Related statements

REAL

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80

ELSE

Provides a default path of execution for IF construct.

Syntax

```
ELSE [ construct-name ]
```

construct-name

is the name given to the IF construct. If *construct-name* is specified, the same name must also appear in the IF statement and in the END IF statement.

Description

The ELSE statement is used in an IF construct to provide a statement block for execution if none of the logical expressions in the IF and ELSE IF statements in the IF construct evaluates to true.

An IF construct may contain (at most) one ELSE statement. If present, it must follow all ELSE IF statements within the IF construct.

Examples

```
IF (a > b) THEN
  max = a
ELSE IF (b > max) THEN
  max = b
ELSE
  PRINT *, 'The two numbers are equal.'
  STOP 'Done'
END IF
```

Related statements

ELSE IF, END IF, and IF (construct)

Related concepts

See “IF construct” on page 111.

ELSE IF

Provides alternate path of execution for IF construct.

Syntax

`ELSE IF (`*logical-expression*`) THEN [`*construct-name*`]`

logical-expression

is a scalar logical expression.

construct-name

is the name given to the `IF` construct. If *construct-name* is specified, the same name must also appear in the `IF` statement and in the `END IF` statement.

Description

The `ELSE IF` statement executes the immediately following statement block, if the following conditions are met:

- None of the logical expressions in the `IF` statement and any previous `ELSE IF` statements evaluates to true.
- *logical-expression* evaluates to true.

Branching to an `ELSE IF` statement is illegal.

Examples

```
INTEGER temperature
INTEGER, PARAMETER :: hot=1, cold=2
IF (temperature == hot) THEN
  PRINT *, 'Turn down your thermostat.'
ELSE IF (temperature == cold) THEN
  PRINT *, 'Turn up your thermostat.'
ELSE
  PRINT *, 'Your thermostat is working OK.'
END IF
```

Related statements

`ELSE`, `END IF`, and `IF` (construct)

Related concepts

See “IF construct” on page 111.

ELSEWHERE

Introduces optional ELSEWHERE block within a WHERE construct.

Syntax

```
ELSEWHERE
```

Description

The ELSEWHERE statement introduces an ELSEWHERE block, which is an optional component of the WHERE construct. The ELSEWHERE statement executes on the complement of the WHERE condition. For additional information, see “WHERE (statement and construct)” on page 466.

Examples

```
WHERE( b .GE. 0.0 )
   ! Assign to sqrt_b only where logical array b is 0 or positive
   sqrt_b = SQRT(b)
ELSEWHERE
   sqrt_b = 0.0     ! Assign sqrt_b where b is negative
END WHERE
```

Related statements

WHERE and END (construct)

Related concepts

For information about the WHERE construct, see “Masked array assignment” on page 99.

ENCODE (extension)

Outputs formatted data to internal storage.

Syntax

`ENCODE (`*count*`,` *format*`,` *unit*`,` *io-specifier-list*`)` `[`*out-list*`]`

count

is an integer expression that specifies the number of characters (bytes) to translate from character format to internal (binary) format. *count* must precede *format*.

format

specifies the format specification for formatting the data. *format* can be one of the following:

- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification. For information about embedded format specifications, see "Embedded format specification" on page 237.

format must be the second of the parenthesized items, immediately following *count*. Note that the keyword `FMT=` is not used.

unit

is the internal storage designator. It must be a scalar variable or array name. Assumed-size and adjustable-size arrays are not permitted. Note that *char-var-name* is not a unit number and that the keyword `UNIT=` is not used.

unit must be the third of the parenthesized items, immediately following *format*.

io-specifier-list

is a comma-separated list of I/O specifiers. Note that the unit and format specifiers are required; the other I/O specifiers are optional. The following I/O specifiers can appear in *io-specifier-list*:

ERR=*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

IOSTAT=*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.

out-list

is a comma-separated list of data items for output. The data items can include expressions and implied-DO lists (see “Implied-DO loop” on page 194).

Description

The ENCODE statement is a nonstandard feature of HP Fortran 90 and is provided for compatibility with other versions of Fortran. The internal-I/O capabilities of the standard WRITE statement provide similar functionality and should be used to ensure portability.

The ENCODE statement translates data from its internal (binary) representation into formatted character data.

Examples

The following example program uses the ENCODE statement to write to an internal file:

```
PROGRAM encode_example
  CHARACTER(LEN=20) :: buf
  ENCODE (LEN(buf), '(2X, 3I4, 1X)', buf)  1234, 45, -12
  PRINT *, buf
END PROGRAM encode_example
```

When compiled and executed, this program outputs the following (where *b* represents a blank character):

*bb*1234*bb*45*b*-12*bbbbb*

Related statements

DECODE and WRITE

Related concepts

For related information, see the following:

- “Internal files” on page 174
- “Performing I/O on internal files” on page 176
- “Implied-DO loop” on page 194
- “Embedded format specification” on page 237

END

Marks the end of a program unit or procedure.

Syntax

```
END [keyword [name]]
```

keyword

is one of the keywords BLOCK DATA, FUNCTION, MODULE, PROGRAM, or SUBROUTINE. When the END statement is used for an internal procedure or module procedure, the FUNCTION or SUBROUTINE keyword is required.

name

is the name given to the program unit. If *name* is specified, *keyword* must also be specified.

Description

The END statement is the last statement of a program unit (that is, a main program, function, subroutine, module, or block data subprogram), an internal procedure, or a module procedure. It is the only statement that is required within a program unit.

Examples

The following example illustrates the use of the END statement to indicate the end of a main program. Notice that, even though the main program unit is given a name, the END PROGRAM statement does not require it:

```
PROGRAM main_prog
...
END PROGRAM
```

In the next example, the END statement marks the end of an internal function and must therefore specify the keyword FUNCTION. However, it is not required that the name, get_args, be also specified:

```
FUNCTION get_args (arg1, arg2)
...
END FUNCTION get_args
```

The following example uses the END statement to indicate the end of a block data subprogram. Because the END statement specifies the program unit name, it must also specify the keyword BLOCK DATA:

```
BLOCK DATA main_data
...
END BLOCK DATA main_data
```

Related statements `BLOCK DATA`, `FUNCTION`, `MODULE`, `PROGRAM`, and `SUBROUTINE`

Related concepts For information about program units, see “Program units” on page 122.

END (construct)

Terminates a CASE, DO, IF, or WHERE construct.

Syntax

`END` *construct-keyword* [*construct-name*]

construct-keyword

is one of the keywords `DO`, `IF`, `SELECT CASE`, or `WHERE`.

construct-name

is the name given to the construct terminated by this statement.

Description

The `END` (construct) statement terminates a `CASE`, `DO`, `IF`, or `WHERE` construct. If *construct-name* appears in the statement that introduces the construct, the same name must also appear in the `END` statement. If no *construct-name* is given in the introducing statement, none must appear in the `END` statement.

Examples

For examples of the `END` (construct) statement, see the descriptions of the `DO`, `IF`, `SELECT`, or `WHERE` statements throughout this chapter.

Related statements

`DO`, `IF`, `SELECT CASE`, and `WHERE`

Related concepts

For related information, see the following:

- “Masked array assignment” on page 99
- “Control constructs and statement blocks” on page 104

END (structure definition, extension)

Terminates the definition of a structure or union.

Syntax

`END` *record-keyword*

record-keyword

is one of the keywords `MAP`, `STRUCTURE`, or `UNION`.

Description

The `END` (record definition) statement is an HP Fortran 90 extension that is used to delimit the definition of a structure (`END STRUCTURE`) or a union within a structure (`END UNION` and `END MAP`). For more information, refer to "STRUCTURE (extension)" on page 437.

Related statements

`INTERFACE`, `STRUCTURE`, and `UNION`

END INTERFACE

Terminates a procedure interface block.

Syntax

```
END INTERFACE
```

Description

In Fortran 90, external procedures may be given explicit interfaces by means of procedure interface blocks. Such a block is always terminated by the `END INTERFACE` statement.

Examples

The following makes the interface of function `r_ave` explicit, giving it the generic name `g_ave`.

```
INTERFACE g_ave
   FUNCTION r_ave(x)
      ! get the size of array x from module ave_stuff
      USE ave_stuff, ONLY: n
      REAL r_ave, x(n)
   END FUNCTION r_ave
END INTERFACE
```

Related statements

`INTERFACE`

Related concepts

Interface blocks are described in “Interface blocks” on page 152.

END TYPE

Terminates a derived type definition.

Syntax

`END TYPE [`*type-name*`]`

type-name

is the name of the derived type being defined. *type-name* is optional. If given, it must be the same as the *type-name* specified in the `TYPE` statement introducing the derived type definition.

Description

The `END TYPE` statement terminates the definition of a derived type.

Examples

The following is a simple example of a derived type with two components, `high` and `low`:

```
TYPE temp_range
   INTEGER high, low
END TYPE temp_range
```

Related statements

`TYPE` (definition)

Related concepts

Derived types are described in "Derived types" on page 39.

ENDFILE

Writes end-of-file record to file.

Syntax

The syntax of the `ENDFILE` statement can take one of the following forms:

- Short form:

  `ENDFILE` *integer-expression*

- Long form:

  `ENDFILE (`*io-specifier-list*`)`

integer-expression
: is the number of the unit connected to a sequential file.

io-specifier-list
: is a list of the following comma-separated I/O specifiers:

  `[UNIT=]`*unit*

  specifies the unit connected to a device or external file opened for sequential access. *unit* must be an integer expression that evaluates to a nonnegative number. If the optional keyword `UNIT=` is omitted, *unit* must be the first item in *io-specifier-list*.

  `ERR=`*stmt-label*

  specifies the label of the executable statement to which control passes if an error occurs during statement execution.

  `IOSTAT=`*integer-variable*

  returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The `ENDFILE` statement writes an end-of-file record to the file or device connected to the specified unit at the current position and positions the file after the end-of-file record.

An end-of-file record can occur only as the last record of a disk file. After execution of an `ENDFILE` statement, the file is positioned beyond the end-of-file record; any records beyond the current position are lost—that is, the file is truncated.

Some devices (for example, magnetic tape units) can have multiple end-of-file records, with or without intervening data records.

An end-of-file record can be written to a sequential file only.

Examples

The following statement writes an end-of-file record to the file connected to unit 10:

```
ENDFILE 10
```

The following statement writes an end-of-file record to the file connected to unit 17. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in `ios`:

```
INTEGER :: ios
...
ENDFILE (17, ERR=99, IOSTAT=ios)
```

Related statements

`BACKSPACE`, `OPEN`, and `REWIND`

Related concepts

For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171, which also lists example programs that use I/O. For information about I/O formatting, see Chapter 9, “I/O formatting,” on page 205.

ENTRY

Provides an additional external or module subprogram entry point.

Syntax

```
ENTRY entry-name [([dummy-arg-list])
      [RESULT (result-name)]]
```

entry-name

is the name of the entry point (subroutine or function) defined by the ENTRY statement. It must differ from the original subroutine or function name, and from other ENTRY statement *entry-name*s specified in the subprogram in which it appears.

dummy-arg-list

is a comma-separated list of dummy arguments for the subroutine or function defined by the ENTRY statement. The same rules and restrictions apply as for subroutine dummy arguments or function dummy arguments, as appropriate.

result-name

is the result variable for a function defined by an ENTRY statement. *result-name* is optional; if not specified, the result variable is *entry-name*.

The RESULT (*result-name*) clause can only be specified when the ENTRY statement is included in a function subprogram.

Description

When an ENTRY statement appears in a function subprogram, it effectively provides an additional FUNCTION statement in the subprogram: execution starts from the ENTRY statement when the *entry-name* is invoked (by being used). Similarly, an ENTRY statement in a subroutine subprogram effectively provides an additional SUBROUTINE statement in the subprogram, and execution starts from the ENTRY statement when the *entry-name* is called.

The following restrictions apply to the ENTRY statement:

- The ENTRY statement can appear in an external subprogram or a module subprogram; it may not appear in an internal subprogram. If the ENTRY statement appears in a function subprogram, it defines an additional function; if it appears in a subroutine subprogram, it defines an additional subroutine. The entry points thus defined can be referenced in the same way as for a normal function name or subroutine name, as appropriate. Execution starts at the ENTRY statement, and continues in the normal manner, ignoring any ENTRY statements subsequently encountered, until a RETURN statement or the end of the procedure is reached.

- The RESULT (*result-name*) clause can only be specified when the ENTRY statement is included in a function subprogram. If specified, *result-name* must differ from *entry-name*, and *entry-name* must not appear in any specification statement in the scoping unit of the function subprogram; *entry-name* assumes all the attributes of *result-name*. The RESULT clause in an ENTRY statement has the same syntax and semantics as in a FUNCTION statement.
- If the ENTRY statement appears in a function, the result variable is that specified in the FUNCTION statement; if none is specified, the result variable is *entry-name*.
- If the characteristics of the result variable specified in the ENTRY statement are the same as those of the result variable specified in the FUNCTION statement, then the result variable is the same, even though the names are different. If the characteristics are different, then the result variables must be:
  - Nonpointer scalars of intrinsic type
  - Storage associated
  - If any is of character type, they must all be of character type and must all have the same length. If any is of noncharacter type, they must all be of noncharacter type.
- The result variable may not appear in a COMMON, DATA, or EQUIVALENCE statement. Also, the result variable may not have the ALLOCATABLE, INTENT, OPTIONAL, PARAMETER, or SAVE attribute.
- If RECURSIVE is specified on the FUNCTION statement at the start of a function subprogram, and RESULT is specified on an ENTRY statement within the subprogram, then the interface of the function defined by the ENTRY statement is explicit within the function subprogram; the function can thus be invoked recursively. (Note that the keyword RECURSIVE is not given on the ENTRY statement, but only on the FUNCTION statement.)

- If RECURSIVE is specified on the SUBROUTINE statement at the start of a subroutine subprogram, the interface of the subroutine defined by an ENTRY statement within the subprogram is explicit within the subprogram; the subroutine can thus be called recursively.
- A dummy argument in an ENTRY statement must not appear in an executable statement preceding the ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement preceding the executable statement.
- If a dummy argument in a subprogram—that is, as specified in a FUNCTION or SUBROUTINE statement at the start of the subprogram or in any ENTRY statements within the subprogram—is used in an executable statement, then the statement may only be executed if the dummy argument appears in the dummy argument list of the procedure name actually referenced in the current call. The same restrictions apply when you use a dummy argument in a specification expression to specify an array bound or character length.
- A procedure defined by an ENTRY statement may be given an explicit interface by use of an INTERFACE block. The procedure header in the interface body must be a FUNCTION statement for an entry to a function subprogram, and a SUBROUTINE statement for an entry to a subroutine subprogram.

The ENTRY statement was often used in FORTRAN 77 programs in situations where a set of subroutines or functions had slightly different dummy argument lists but entailed computations involving identical data and code. In Fortran 90 the use of the ENTRY statement in such situations can be replaced by the use of optional arguments.

Examples

The following example defines a subroutine subprogram with two dummy arguments. The subprogram also contains an ENTRY statement that takes only the first dummy argument specified in the SUBROUTINE statement.

```
SUBROUTINE Full_Name (first_name, surname)
CHARACTER(20) :: first_name, surname
...
ENTRY Part_Name (first_name)
```

The following example creates a stack. It shows the use of ENTRY to group the definition of a data structure together with the code that accesses it, a technique known as encapsulation. (This example could alternatively be programmed as a module, which would be preferable in that it does not rely on storage association.)

```
SUBROUTINE manipulate_stack
  IMPLICIT NONE
  INTEGER size, top /0/, value
  PARAMETER (size = 100)
  INTEGER, DIMENSION(size) :: stack
  SAVE stack, top

  ENTRY push(value)        ! Push value onto the stack
  IF (top == size) STOP 'Stack Overflow'
  top = top + 1
  stack(top) = value
  RETURN

  ENTRY pop(value)         ! Pop top of stack and place in value
  IF (top == 0) STOP 'Stack Underflow'
  value = stack(top)
  top = top - 1
  RETURN
END SUBROUTINE manipulate_stack
```

Here are examples of CALL statements associated with the preceding example:

```
CALL push(10)
CALL push(15)
CALL pop(I)
CALL pop(J)
```

Related statements

FUNCTION, SUBROUTINE, and CALL

Related concepts

For information about external procedures, see “External procedures” on page 128.

EQUIVALENCE

Associates different objects with same storage area.

Syntax

`EQUIVALENCE (`*equivalence-list1*`) [, (`*equivalence-list2*`)]...`

equivalence-list

is a comma-separated list of two or more object names to be storage associated. Objects can include simple variables, array elements, array names, and character substrings.

Description

All objects in each *equivalence-list* share the same storage area. Such objects become storage associated and are equivalenced to each other. Equivalencing may also cause other objects to become storage associated.

The following items must not appear in *equivalence-list*:

- Automatic objects, including character variables whose length is specified with a nonconstant
- Allocatable arrays
- Function names, result names, or entry names
- Dummy arguments
- Records or record field references
- Nonsequenced derived-type objects
- Derived-type components
- Pointers or derived-type objects containing pointers
- Named constants

Derived-type objects may appear in an `EQUIVALENCE` statement if they have been defined with the `SEQUENCE` attribute.

The following restrictions apply to objects that can appear in an `EQUIVALENCE` statement:

- Objects in the same *equivalence-list* must be explicitly or implicitly declared in the same scoping unit.

- The name of an equivalenced object must not be made available by use association.

The Fortran 90 standard imposes the following type restrictions on equivalenced objects:

- If one of the objects in *equivalence-list* is of type default integer, default real, double precision real, default complex, double complex, default logical, or numeric sequence type, then all objects in *equivalence-list* must be one of these types.

  HP Fortran 90 relaxes this restriction and allows character and noncharacter items to be equivalenced. Note, however, that use of this extension can impact portability.

- If one of the objects in *equivalence-list* is of derived type that is not a numeric sequence or character sequence type, then all objects in *equivalence-list* must be of the same type.

- If one of the objects in *equivalence-list* is of intrinsic type other than default integer, default real, double precision real, default complex, double complex, default logical, or default character, then all objects in *equivalence-list* must be of the same type with the same kind type parameter value.

  HP Fortran 90 relaxes this restriction.

The `EQUIVALENCE` statement does not cause type conversion or imply mathematical equivalence. If an array and a scalar share the same storage space through the `EQUIVALENCE` statement, the array does not have the characteristics of a scalar and the scalar does not have the characteristics of an array. They only share the same storage space.

Care should be taken when data types of different sizes share the same storage space, because the `EQUIVALENCE` statement specifies that each data item in *equivalence-list* has the same first storage unit. For example, if a 4-byte integer variable and a double-precision variable are equivalenced, the integer variable shares the same space as the 4 most significant bytes of the 8-byte double-precision variable.

Proper alignment of data types is always enforced. The compiler will issue a diagnostic if incorrect alignment is forced through an `EQUIVALENCE` statement. For data type alignment rules, see “Intrinsic data types” on page 22.

The lengths of the equivalenced objects need not be the same.

Equivalencing character data

An EQUIVALENCE statement specifies that the storage sequences of character data items whose names are specified in *equivalence-list* have the same first character storage unit. This causes the association of the data items in *equivalence-list* and can cause association of other data items as well. Consider the following example:

```
CHARACTER(LEN=4) :: a, b
CHARACTER(LEN=3) :: c(2)
EQUIVALENCE (a, c(1)), (b, c(2))
```

As a result of this EQUIVALENCE statement, the fourth character in a, the first character in b, and the first character in c(2) share the same storage.

Strings of the same or different lengths can be equivalenced to start on the first element, and you can use substring notation to specify other associations, as in the following:

```
CHARACTER (10) :: s1, s2
EQUIVALENCE (s1(2:2), s2(3:3)
```

Substring subscripts must be integer initialization expressions, and the substring length must be nonzero.

Equivalencing arrays

To determine equivalence between arrays with different dimensions, HP Fortran 90 views all elements of an array in linear sequence. Each array is stored as if it were a one-dimensional array. Array elements are stored in ascending sequential, column-major order; for information about how arrays are laid out in memory, see “Array fundamentals” on page 52.

Array elements can be equivalenced with elements of a different array or with scalars. No equivalence occurs outside the bounds of any of the equivalenced arrays.

If equivalenced arrays are not of the same type, they may not line up element by element.

If an array name appears without subscripts in an EQUIVALENCE statement, it has the same effect as specifying an array name with the subscript of its first element.

It is illegal to equivalence different elements of the same array to the same storage area. For example, the following is illegal:

```
INTEGER :: a(2), b
EQUIVALENCE (a(1), b), (a(2), b)
```

Likewise, it is illegal to use the EQUIVALENCE statement to force consecutive array elements to be noncontiguous, as in the following example:

```
REAL :: a(2), r(3)
EQUIVALENCE (a(1), r(1)), (a(2), r(3))
```

Array subscripts must be integer initialization expressions.

Equivalence in common blocks

An EQUIVALENCE statement must not cause two common blocks to be associated. However, you can use the EQUIVALENCE statement to place objects in common by equivalencing them to objects already in common. If one element of an array is equivalenced to an object in common, the whole array is placed in common with equivalence maintained for storage units preceding and following the data element in common. The common block is always extended when it is necessary to fit an array that shares storage space in the common block. It may be extended after the last entry, but not before the first.

Consider the following example, which puts array i in blank common and equivalences array element j(2) to i(3):

```
INTEGER :: i(6), j(6)
COMMON i
EQUIVALENCE (i(3), j(2))
```

The effect of the EQUIVALENCE statement is to extend blank common to include element j(6). This is entirely legal because the extension occurs at the end of the common block.

But if the EQUIVALENCE statement were changed as follows:

```
EQUIVALENCE (i(1), j(2))  ! illegal
```

it would result in an illegal equivalence, because storage would have to be inserted in front of the block in order to accommodate element j(1).

Examples

In the following example, the variables a, b, and c share the same storage space; array elements d(2) and e(5) share the same storage space; variables f, g, and h share the same storage:

```
INTEGER :: a, b, c, d(20), e(30), f, g, h
EQUIVALENCE (a, b, c), (d(2), e(5)), (f, g, h)
```

Related statements

COMMON

Related concepts

For information about data alignment, see Table 5 and “Alignment of derived-type objects” on page 44.

EXIT

Terminates a DO loop.

Syntax

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

do-construct-name

is the name given to the DO construct. If *do-construct-name* is specified, it must be the name of a DO construct that contains the EXIT statement.

Description

If you do not specify *do-construct-name*, the EXIT statement terminates the immediately enclosing DO loop. If you do specify it, the EXIT statement terminates the enclosing DO loop with the same name.

Examples

```
DO i = 1, 20
  n(i) = 0
  READ *, j
  IF (j < 0) EXIT
  n(i) = j
END DO
```

Related statements

CYCLE and DO

Related concepts

For related information, see the following:

- “DO construct” on page 107
- “Flow control statements” on page 112

EXTERNAL (statement and attribute)

Declares a name to be external.

Syntax

A type declaration statement with the EXTERNAL attribute is:

type , *attrib-list* :: *function-name-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.).

attrib-list

is a comma-separated list of attributes including EXTERNAL and optionally those attributes compatible with it, namely:

OPTIONAL	PRIVATE	PUBLIC

function-name-list

is a comma-separated list of function names to be designated EXTERNAL.

The syntax of the EXTERNAL statement is:

EXTERNAL *external-name-list*

Note that the syntax of the EXTERNAL statement does not permit optional colons.

Description

An EXTERNAL attribute or statement specifies that a name may be used as an actual argument in subroutine calls and function references. The name is either an external procedure, a dummy procedure, or a block data program unit.

A name that appears in a type statement specifying the EXTERNAL attribute must be the name of an external procedure or of a dummy argument that is a procedure.

The following rules and restrictions apply:

- A name can appear once in an EXTERNAL statement, in a declaration statement with an EXTERNAL attribute, or in an interface body, but not in more than one of these.

- The EXTERNAL attribute cannot be used with subroutines. To declare a subroutine as EXTERNAL, use the statement form.
- If the name is a dummy argument, an EXTERNAL statement declares it to be a dummy procedure.
- If a user-defined procedure or library routine has the same name as an intrinsic procedure, then it must either be declared to have the EXTERNAL attribute or have an explicit interface. The intrinsic procedure is then no longer available in such program units.
- The INTRINSIC and EXTERNAL attributes are mutually exclusive.

Examples

```
SUBROUTINE sub (fourier)
! fourier is a dummy procedure; actual argument corresponding to
! to fourier can be external, intrinsic, or module procedure
   REAL fourier
   EXTERNAL fourier              ! statement form
REAL, EXTERNAL :: SIN, COS, TAN  ! attribute form
! SIN, COS, and TAN are no longer intrinsic procedures; functions
! with these names must be defined in the program
...
END SUBROUTINE sub
SUBROUTINE gratx (x, y)
! Specify init_block_a as the block data
! subprogram that initializes common block a
EXTERNAL init_block_a
! Common block available in subroutine gratx
COMMON /a/ temp, pressure
END SUBROUTINE gratx

BLOCK DATA init_block_a
! init_block_a initializes the objects in common block a
COMMON /a/ temp, pressure
DATA temp, pressure/ 98.6, 15.5 /
END BLOCK DATA init_block_a
```

Related statements

INTRINSIC

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Procedures” on page 123
- “Declaring library routines as EXTERNAL” on page 590

FORMAT

Describes how I/O data is to be formatted.

Syntax

label FORMAT (*format-list*)

label

is a statement label.

format-list

is a comma-separated list of format items, where each item in the list can be either one of the edit descriptors described in Table 25 or (*format-list*). If *format-list* is a list item, it may be optionally preceded by a repeat specification—a positive integer that specifies how may times *format-list* is to be repeated.

Description

The FORMAT statement holds the format specification that indicates how data in formatted I/O is to be translated between internal (binary) representation and formatted (ASCII) representation. The translation makes it possible to represent data in a humanly readable format.

Although a format specification can be embedded within a data transfer statement, the point to using a FORMAT statement is to make it available to any number of data transfer statements. Several data transfer statements can use the same format specification contained in a FORMAT statement by referencing *label*.

Another advantage of the FORMAT statement over the use of embedded format specifications is that it is ”pre-compiled”, reducing the runtime overhead of processing the format specification and providing compile-time error checking of the FMT= specifier.

Examples

```
   PROGRAM format_example
   WRITE (15,FMT=20) 1234, 45, -12
20 FORMAT (I6, 2I4)
   END PROGRAM format_example
```

When compiled and executed, this program outputs the following (where *b* represents the blank character):

*bb*1234*bb*45*b*-12

Related statements `READ` and `WRITE`

Related concepts For information about I/O formatting, see Chapter 9, “I/O formatting,” on page 205.

FUNCTION

Introduces a function subprogram.

Syntax

```
[RECURSIVE] [type-spec] FUNCTION
        function-name ([dummy-arg-name-list])
        [RESULT (result-name)]
```

RECURSIVE

: is a keyword that must be specified in the FUNCTION statement if the function is either directly or indirectly recursive. The RECURSIVE clause can appear at most once, either before or after *type-spec*. It is not an error to specify RECURSIVE for a nonrecursive function.

: A recursive function that calls itself directly must also have the RESULT clause specified (see below).

type-spec

: is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.). The type and type parameters of the function result can be specified by *type-spec* or by declaring the result variable within the function subprogram, but not by both. The implicit typing rules apply if the function is not typed explicitly.

: If the function result is array-valued or a pointer, the appropriate attributes for the result variable (which is *function-name*, or *result-name* if specified) must be specified within the function subprogram.

function-name

: is the name of the function subprogram being defined.

dummy-arg-name-list

: is a comma-separated list of dummy argument names for the function.

result-name

: is the result variable. If the RESULT clause is not specified, *function-name* becomes the result variable. If *result-name* is given, it must differ from *function-name*, and *function-name* must not then be declared within the function subprogram.

As noted above, a recursive function that calls itself directly must have the `RESULT` clause specified. For other functions, the `RESULT` clause is optional.

Description

A `FUNCTION` statement introduces an external, module, or internal function subprogram.

Examples

```
PROGRAM main
...
CONTAINS
   ! f is an internal function
   FUNCTION f(x)
         f = 2*x + 3
   END FUNCTION f
   ! recursive function, which must specify RESULT clause
   RECURSIVE INTEGER FUNCTION factorial (n) &
        RESULT (factorial_value)
     IMPLICIT INTEGER (a-z)
     IF (n <= 0) THEN
       factorial_value = 1
     ELSE
       factorial_value = n * factorial (n-1)
     END IF
   END FUNCTION factorial
END PROGRAM main
```

Related statements

`CONTAINS`, `END`, `INTENT`, `INTERFACE`, `OPTIONAL`, and the type declaration statements

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “External procedures” on page 128
- “Arguments” on page 139
- “Defined operators” on page 155

GO TO (assigned)

Transfers control to a variable that was assigned a label.

Syntax

`GO TO` *integer-variable* `[[,] (`*label-list*`)]`

integer-variable

is a scalar variable of default type integer.

label-list

is a list of statement labels, separated by commas.

Description

The assigned `GO TO` statement transfers control to the statement whose label was most recently assigned to a variable with the `ASSIGN` statement.

integer-variable must be given a label value of an executable statement through an `ASSIGN` statement prior to execution of the `GO TO` statement. When the assigned `GO TO` statement is executed, control is transferred to the statement whose label matches the label value of *integer-variable*.

label-list is a list of labels that *integer-variable* might assume.

integer-variable must not be an array element or an integer component of a derived type.

The use of this statement can hinder the ability of the compiler to optimize the program in which it occurs.

Examples

```
ASSIGN 10 TO out
GO TO out
```

Related statements

`ASSIGN`, `GO TO` (computed), and `GO TO` (unconditional)

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

GO TO (computed)

Transfers control to one of several labels.

Syntax

`GO TO (` *label-list* `) [,]` *arithmetic-expression*

label-list

is a list of statement labels, separated by commas.

arithmetic-expression

is a scalar integer expression. As an extension, HP Fortran 90 also allows the expression to be of type real or double precision.

Description

The computed `GO TO` statement transfers control to one of several labeled statements, depending on the value of *arithmetic-expression*. After *arithmetic-expression* is evaluated (and, if necessary, truncated to an integer value), control transfers to the statement label whose position in *label-list* corresponds to the truncated value of *arithmetic-expression*.

If the value of *arithmetic-expression* is less than 1 or greater than the total number of labels in *label-list*, control transfers to the executable statement immediately following the computed `GO TO` statement.

Examples

```
index = 3
! Branch made to the statement labeled 30.
GO TO (10, 20, 30, 40) index
```

Related statements

`SELECT CASE`, `GO TO` (assigned), and `GO TO` (unconditional)

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

GO TO (unconditional)

Transfers control to a specified label.

Syntax

```
GO TO label
```

label

is the label of an executable statement.

Description

The unconditional `GO TO` statement transfers control directly to the statement at the specified label. The executable statement with *label* can occur before or after the `GO TO` statement, but it must be within the same scoping unit.

Examples

```
   GO TO 30
30 CONTINUE
```

Related statements

`GO TO` (assigned) and `GO TO` (computed)

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

IF (arithmetic)

Transfers control to one of three labels.

Syntax

`IF (`*arithmetic-expression*`)` *labelN*`,` *labelZ*`,` *labelP*

arithmetic-expression

is an arithmetic expression of any numeric type except complex and double complex.

label

is a label of an executable statement.

Description

The arithmetic `IF` statement transfers control to the statement whose label is determined by *arithmetic-expression*. If *arithmetic-expression* evaluates to a negative value, control transfers to *labelN*; if it evaluates to 0, control transfers to *labelZ*; and if it evaluates to a positive value, control transfers to *labelP*.

The same label may appear more than once in the same arithmetic `IF` statement.

Each label must be that of an executable statement in the same scoping unit as the arithmetic `IF`.

Examples

```
i = -1

! Branch to statement labeled 10
IF (i) 10, 20, 30
```

Related statements

`IF` (construct) and `IF` (logical)

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

IF (block)

Begins an IF construct.

Syntax

`[construct-name :] IF (logical-expression) THEN`

construct-name

is the name given to the IF construct. If *construct-name* is specified, the same name must also appear in the END IF statement.

logical-expression

is a scalar logical expression.

Description

The IF statement executes the immediately following statement block if *logical-expression* evaluates to true.

The IF construct, which the IF statement begins, may include ELSE IF statements and an ELSE statement to provide alternate statement blocks for execution.

The block following the IF statement may be empty.

As an extension, HP Fortran 90 allows the transfer of control into an IF construct from outside the construct.

Examples

```
IF (x <= 0.0 .AND. y > 1.0) THEN
  CALL fix_coord(x, y)
END IF
```

Related statements

ELSE, ELSE IF, IF (arithmetic), IF (logical), and END (construct)

Related concepts

For information about the IF construct, see “IF construct” on page 111.

IF (logical)

Conditionally executes a statement.

Syntax

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

logical-expression
: is a logical expression.

statement
: is any executable statement other than the following:

- A statement used to begin a construct
- Any END statement
- Any IF statement

Description

The logical `IF` statement is a two-way decision maker. If *logical-expression* evaluates to is true, *statement* executes and control passes to the next statement. If *logical-expression* evaluates to false, *statement* does not execute and control passes to the next statement in the program.

Examples

```
IF (a .EQ. b) PRINT *, 'They are equal.'
```

Related statements

`IF` (arithmetic) and `IF` (construct)

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

IMPLICIT

Changes or voids default typing rules.

Syntax

The `IMPLICIT` statement can take either of the following forms:

- First form:

  `IMPLICIT` *type* `(`*range-list*`)[,` *type* `(`*range-list*`) ,]...`

- Second form:

  `IMPLICIT NONE`

type
: is the data type to be associated with the corresponding letters in *range-list*.

range-list
: is a comma-separated list of letters or ranges of letters (for example, `A-Z` or `I-N`) to be associated with *type*. Writing a range of letters has the same effect as writing a list of single letters.

Description

The `IMPLICIT` statement can be used either to change or void the default typing rules within the program unit in which it appears, depending on which of the two forms the statement takes.

First form

This form of the `IMPLICIT` statement specifies *type* as the data type for all variables, arrays, named constants, function subprograms, `ENTRY` names in function subprograms, and statement functions that begin with any letter in *range-list* and that are not explicitly given a type.

Within the specification statements of a program unit, `IMPLICIT` statements must precede all other specification statements, except possibly the `DATA` and `PARAMETER` statements.

The same letter must not appear as a single letter or be included in a range of letters, more than once in all of the `IMPLICIT` statements in a scoping unit.

For information on how the `IMPLICIT` and `PARAMETER` statements interact, refer to “PARAMETER (statement and attribute)” on page 391.

Second form

The IMPLICIT NONE statement disables the default typing rules for all variables, arrays, named constants, function subprograms, ENTRY names, and statement functions (but not intrinsic functions). All such objects must be explicitly typed. The IMPLICIT NONE statement must be the only IMPLICIT statement in the scoping unit, and it must precede any PARAMETER statement. Types of intrinsic functions are not affected.

You can also use the +implicit_none compile-line option to void the default typing rules. A program compiled with this option may include IMPLICIT statements, which the compiler will honor.

Examples

The following statement causes all variables and function names beginning with I, J, or K to be of type complex, and all data items beginning with A, B, or C to be of type integer:

```
IMPLICIT COMPLEX (I, J, K), INTEGER (A-C)
```

Related concepts

For related information, see the following:

- “Implicit typing” on page 28

INCLUDE

Imports text from a specified file.

Syntax

`INCLUDE` *character-literal-constant*

character-literal-constant
: is the name of the file to include.

Description

The keyword `INCLUDE` and *character-literal-constant* form an `INCLUDE` line, which is used to insert text into a program prior to compilation. The inserted text replaces the `INCLUDE` line; the `INCLUDE` line should therefore appear in your program where you want the inserted text. When the end of an included file is reached, the compiler continues processing with the line following the `INCLUDE` line.

character-literal-constant can be either a file name or a device name. It must not have a kind parameter that is a named constant.

The `INCLUDE` line must appear on one line with no other text except possibly a trailing comment. It should not have a statement label. Thus, you cannot branch to it, and it cannot be an action statement that is part of a Fortran 90 `IF` statement. You cannot use the "`;`" operator to add a second `INCLUDE` line, nor can you use the "&" operator to continue it over another line.

The compiler searches directories for the named include files in the following order:

1. The current source directory
2. Directories specified by the `-I` compile-line option, in the order specified
3. The current working directory
4. The directory `/usr/include`

`INCLUDE` lines can be nested to a maximum of ten levels. However, they must be nested nonrecursively. That is, inserted text must not specify an `INCLUDE` line that was encountered at an earlier level of nesting.

Line numbering within the listing of an included file begins at 1. When the included file listing ends, the include level decreases appropriately, and the previous line numbering resumes.

Examples

```
INCLUDE 'my_common_blocks'
INCLUDE "/my_stuff/declarations.h"
```

Related concepts

For related information, see the following:

- “INCLUDE line” on page 19

INQUIRE

Returns information about file properties.

Syntax

The syntax of the INQUIRE statement has two forms:

- Inquiry by output list:

  `INQUIRE ( IOLENGTH=` *integer-variable* `)` *output-list*

- Inquiry by unit or file:

  `INQUIRE (` *io-specifier-list* `)`

integer-variable

is the length of the unformatted record that would result from writing *output-list* to a direct-access file. The value returned in *integer-variable* can be used with the RECL= specifier in an OPEN statement to specify the length of each record in an unformatted direct-access file that will hold the data in *output-list*.

output-list

is a comma-separated list of data items, similar to what would be included with the WRITE or PRINT statement. The data items can include variables and implied-DO lists (see “Implied-DO loop” on page 194).

io-specifier-list

is a list of comma-separated I/O specifiers. As noted in the following descriptions, most of the specifiers return information about the specified unit or file. *io-specifier-list* must include either the UNIT= or FILE= specifier, but not both. The following paragraphs describe all the I/O specifiers that can appear in *io-specifier-list*:

[UNIT=]*unit*

specifies the unit connected to an external file. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword UNIT= is omitted, *unit* must be the first item in *io-specifier-list*. If *unit* appears in *io-specifier-list*, the FILE= specifier must not be used.

`ACCESS=`*character*

returns the following values, indicating the method of access:

`'SEQUENTIAL'`	File is connected for sequential access.
`'DIRECT'`	File is connected for direct access.
`'UNDEFINED'`	File is not connected.

`ACTION=`*character-variable*

returns the following values, indicating the direction of the transfer:

`'READ'`	File is connected for reading only.
`'WRITE'`	File is connected for writing only.
`'READWRITE'`	File is connected for reading and writing.
`'UNDEFINED'`	File is not connected.

`BLANK=`*character-variable*

returns the type of blank control that is in effect. For information about blank control, see the `BLANK=` specifier for the `OPEN` statement. The values returned by the `BLANK=` specifier are:

`'NULL'`	Null blank control is in effect.
`'ZERO'`	Zero blank control is in effect.
`'UNDEFINED'`	File is not connected for formatted I/O.

`DELIM=`*character-variable*

returns the following values, indicating the character to use (if any) to delimit character values in list-directed and namelist formatting:

`'APOSTROPHE'`	An apostrophe is used as the delimiter.
`'QUOTE'`	The double quotation mark is used as the delimiter.
`'NONE'`	There is no delimiting character.
`'UNDEFINED'`	File is not connected for formatted I/O.

`DIRECT=`*character-variable*

returns the following values, indicating whether or not the file is connected for direct access:

`'YES'`	File is connected for direct access.
`'NO'`	File is not connected for direct access.
`'UNKNOWN'`	It cannot be determined whether or not file is connected for direct access.

`ERR=`*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

`EXIST=`*logical-variable*

returns the following values, indicating whether or not the file or unit exists:

`'TRUE'`	File exists or unit is connected.
`'FALSE'`	File does not exist or unit is not connected.

`FILE=`*character-expression*

specifies the name of a file for inquiry. The file does not have to be connected or even exist. If the `FILE=` specifier appears in *io-specifier-list*, the `UNIT=` specifier must not be used.

`FORM=`*character-variable*

returns the following values, indicating whether the file is connected for formatted or unformatted I/O:

`'FORMATTED'`	File is connected for formatted I/O.
`'UNFORMATTED'`	File is connected for unformatted I/O.
`'UNDEFINED'`	File is not connected.

`FORMATTED=`*character-variable*

returns the following values, indicating whether or not the file is connected for formatted I/O:

`'YES'`	File is connected for formatted I/O.
`'NO'`	File is not connected for formatted I/O.
`'UNKNOWN'`	It cannot be determined whether or not file is connected for formatted I/O.

`IOSTAT=`*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred..

`NAME=`*character-variable*

returns the name of file connected to the specified unit. If the file has no name or is not connected, `NAME=` returns the string `UNDEFINED`.

`NAMED=`*logical-variable*

returns the following values, indicating whether or not the file has a name:

`'TRUE'`	File has a name.
`'FALSE'`	File does not have a name.

`NEXTREC=`*integer-variable*

returns the number of the next record to be read or written in a file connected for direct access. The value is the last record read or written +1. A value of 1 indicates that no records have been processed. If the file is not connected or it is a device file or its status cannot be determined, integer-variable is undefined.

`NUMBER=`*integer-variable*

returns the unit number that is connected to the specified file. If no unit is connected to the named file, *integer-variable* is undefined.

`OPENED=`*logical-variable*

returns the following values, indicating whether or not the file has been opened (that is, is connected):

`'TRUE'`	File is connected.
`'FALSE'`	File is not connected.

`PAD=`*character-variable*

returns a value indicating whether or not input records are padded with blanks. For more information about padding, see the `PAD=` specifier for the `OPEN` statement. The return values are:

`'YES'`	File or unit is connected with `PAD='YES'` in `OPEN` statement.
`'NO'`	File or unit is connected with `PAD='NO'` in `OPEN` statement.

`POSITION=`*character-variable*

returns the following values, indicating the file position:

`'REWIND'`	File is connected with its position at the start of the first record.
`'APPEND'`	File is connected with its position at the end-of-file record.
`'ASIS'`	File is connected without changing its position.
`'UNDEFINED'`	File is not connected or is connected for direct access.

`READ=`*character-variable*

returns the following values, indicating whether or not reading is an allowed action for the file:

`'YES'`	Reading is allowed for file.
`'NO'`	Reading is not allowed for file.
`'UNKNOWN'`	It cannot be determined whether or not reading is allowed for file.

`READWRITE=`*character-variable*

returns the following values, indicating whether or not reading and writing are allowed actions for the file:

`'YES'`	Both reading and writing are allowed for file.
`'NO'`	Reading and writing are not both allowed for file.
`'UNKNOWN'`	It cannot be determined whether or not reading and writing are both allowed for file.

`RECL=`*integer-variable*

returns the record length of the specified unit or file, measured in bytes. The file must be a direct-access file. If the file is not a direct-access file or does not exist, *integer-variable* is undefined.

`SEQUENTIAL=`*character-variable*

returns the following values, indicating whether or not the file is connected for direct access:

`'YES'`	File is connected for sequential access.
`'NO'`	File is not connected for sequential access.
`'UNKNOWN'`	It cannot be determined whether or not file is connected for sequential access.

`UNFORMATTED=`*character-variable*

returns the following values, indicating whether or not the file is connected for formatted I/O:

`'YES'`	File is connected for unformatted I/O.
`'NO'`	File is not connected for unformatted I/O.
`'UNKNOWN'`	It cannot be determined whether or not file is connected for unformatted I/O.

`WRITE=`*character-variable*

returns the following values, indicating whether or not writing is an allowed action for the file:

`'YES'`	Writing is allowed for file.
`'NO'`	Writing is not allowed for file.
`'UNKNOWN'`	It cannot be determined whether or not writing is allowed for file.

Description

The `INQUIRE` statement returns selected properties of a specified file or unit number. (It is illegal to include both the `UNIT=` specifier and the `FILE=` specifier in the same `INQUIRE` statement.) Inquiring by unit number should be used on connected files; inquiring by filename is typically used on unconnected files.

In addition, the `INQUIRE` statement can also be used to determine the record length of a new or existing file. That is, you can use `INQUIRE` to obtain the record length before creating the file and then use the return value as the argument to the `RECL=` specifier in an `OPEN` statement.

Examples

The following examples illustrate different uses of the INQUIRE statement.

Inquiry by file

The INQUIRE statement in this example returns the following information about the file named my_file:

- The EXIST= specifier determines if the file is connected.
- The DIRECT= specifier determines if it is connected for direct access.
- The READWRITE= specifier determines if it can be read and written.

```
LOGICAL :: exist
CHARACTER(LEN=9) :: dir_acc, rw_sts
INQUIRE (FILE='my_file', EXIST=exist, &
         DIRECT=dir_acc, READWRITE=rw_sts)
```

Inquiry by unit

The following INQUIRE statement returns the following information about the file connected to the unit in u_num:

- The OPENED= specifier determines if the file is connected to u_num.
- The NAMED= specifier determines if it is a named file or a scratch file.
- The NAME= specifier returns its name.

```
LOGICAL :: opened, named
INTEGER :: u_num
CHARACTER(LEN=80) :: fname
...
INQUIRE (UNIT=u_num, NAMED=named,  OPENED=opened, NAME=fname)
```

Inquiry by output list

When using the OPEN statement to create a direct-access file, you must specify the record length for the file with the RECL= specifier. Previous to Fortran 90, you had to resort to a nonportable strategy to determine record length. The Fortran 90 INQUIRE statement provides a portable solution: use the INQUIRE statement to inquire by output list, and specify the return value from the INQUIRE statement as the argument to the OPEN statement. The following is an example:

```
INTEGER :: rec_len, ios

INQUIRE (IOLENGTH=rec_len) x, y, i, j
OPEN (UNIT=32, FILE='new_file', IOSTAT=ios, &
      ACCESS='DIRECT', RECL=rec_len)
```

Related statements OPEN

Related concepts For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171.

INTEGER

Declares entities of type integer.

Syntax

`INTEGER [`*kind-spec*`] [[,` *attrib-list*`] ::]` *entity-list*

kind-spec

is the kind type parameter that specifies the range of the entities in *entity-list*. *kind-spec* takes the form:

([KIND=] *kind-param*)

where *kind-param* can be a named constant or a constant expression that has the integer value of 1, 2, 4, or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

**len-param*

where *len-param* is the integer 1, 2, 4, or 8 (default = 4).

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds

initialization-expr

is an integer constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The `INTEGER` statement is used to declare the length and properties of data that are whole numbers. A kind parameter (if present) indicates the representation method.

The `INTEGER` statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

As a portability extension, HP Fortran 90 allows the following syntax for specifying the length of an entity:

name [*len] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*). If *name* appears with **len*, it overrides the length specified by `INTEGER*`*size*.

Examples

The following are valid declarations:

```
INTEGER i, j
INTEGER(KIND=2) :: k
INTEGER(2), PARAMETER :: limit=420
! initialize an array, using an array constructor
INTEGER, DIMENSION(4) :: ivec = (/1, 2, 3, 4 /)
! use the slash notation (an HP extension) to initialize
INTEGER i/-1/, j/-2/, k/-7/  ! note, no double colon
! the following declarations are equivalent; the second uses the
! HP length specification extension
INTEGER (KIND = 8) int1
INTEGER*4 int1*8
```

Related statements BYTE

Related concepts For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80
- “KIND(X)” on page 537

INTENT (statement and attribute)

Specifies the intended use of dummy arguments.

Syntax

A type declaration statement with the `INTENT` attribute is:7

type , *attrib-list* :: *dummy-arg-name-list*

type

is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (`*name*`)`, etc.).

attrib-list

is a comma-separated list of attributes including `INTENT(`*intent-spec*`)` and the optional attributes compatible with it, shown below:

DIMENSION	OPTIONAL	TARGET

intent-spec

is one of `IN`, `OUT`, or `INOUT`. (The form `IN OUT` is valid.)

dummy-arg-name-list

is a comma-separated list of subprogram dummy arguments to which *intent-spec* is to apply.

The syntax of the `INTENT` statement is:

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

Description

The `INTENT` attribute declares whether a dummy argument is intended for transferring a value into a procedure, or out of it, or both. The `INTENT` attribute helps detect the use of arguments inconsistent with their intended use, and may also assist the compiler in generating more efficient code.

If a dummy argument has intent `IN`, the procedure must not change it or cause it to become undefined. If the actual argument is defined, this value is passed in as the value of the dummy argument.

If a dummy argument has intent OUT, the corresponding actual argument must be definable; that is, it cannot be a constant. When execution of the procedure begins, the dummy argument is undefined; thus it must be given a value before it is referenced. The dummy argument need not be given a value by the procedure.

If a dummy argument has intent INOUT, the corresponding actual argument must be definable. If the actual argument is defined, this value is passed in as the value of the dummy argument. The dummy argument need not be given a value by the procedure.

The following points should also be noted:

- Intent specifications apply only to dummy arguments and may only appear in the specification part of a subprogram or interface body.
- If there is no intent specified for an argument in a subprogram, the limitations imposed by the actual argument apply to the dummy argument. For example, if the actual argument is an expression that is not a variable, the dummy argument must not redefine its value.
- The intent of a pointer dummy argument must not be specified.

Examples

```
! x, y, and z are dummy arguments
SUBROUTINE electric (x, y, z)
   REAL, INTENT (IN) :: x, y   ! x and y are used only for input
   ! z is used for input and output
   COMPLEX, INTENT (INOUT), TARGET :: z(1000)
   ...
SUBROUTINE pressure (true, tape, a, b)
   USE a_module
   TYPE(ace), INTENT(IN) :: a, b   ! a and b are only for input
   INTENT (OUT) true, tape         ! true and tape are for output
   ...
SUBROUTINE lab_ten (degrees, x, y, z)
   COMPLEX, INTENT(INOUT) :: degrees
   REAL, INTENT(IN), OPTIONAL :: x, y
   INTENT(IN) z
   ...
PROGRAM pxx
   CALL electric (a+1, h*c, d)   ! First subroutine defined above
   CALL lab_ten (dg, e, f, g+1.0)
END PROGRAM pxx
```

Related statements

FUNCTION and SUBROUTINE

Related concepts

For related information, see the following:

- "Type declaration for intrinsic types" on page 24
- "INTENT attribute" on page 148

INTERFACE

Introduces an interface block.

Syntax

INTERFACE [*generic-spec*]

generic-spec

is one of:

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

generic-name

is the name of a generic procedure.

defined-operator

is one of:

- An intrinsic operator
- .*operator*., where *operator* is a user-defined name

Description

The INTERFACE statement is the first statement of an interface block. Interface blocks constitute the mechanism by which external procedures may be given explicit interfaces and also provide additional functionality, as described below.

The INTERFACE *generic-name* form defines a generic interface for the procedures in the interface block.

The INTERFACE OPERATOR (*defined-operator*) form is used to define a new operator or to extend the meaning of an existing operator.

The INTERFACE ASSIGNMENT(=) form is used to extend the assignment operator so that it can be used (for example) with derived-type objects.

Examples

The following examples illustrate different forms of the interface block:

```
! make explicit the interfaces of external function spline
! and external subroutine sp2
INTERFACE
   REAL FUNCTION spline(x,y,z)
   END FUNCTION spline
```

```
   SUBROUTINE sp2(x,z)
   END SUBROUTINE sp2
END INTERFACE

! Make the interface of function r_ave explicit and give
!  it the generic name g_ave
INTERFACE g_ave
   FUNCTION r_ave(x)
   ! Get the size of x from the module ave_stuff
   USE ave_stuff, ONLY: n
   REAL r_ave, x(n)
   END FUNCTION r_ave
END INTERFACE

! Make the interface of external function b_or  explicit, and
use!  it to extend the + operator
INTERFACE OPERATOR ( + )
   FUNCTION b_or(p, q)
   LOGICAL b_or, p, q
   INTENT (IN) p, q
   END FUNCTION b_or
END INTERFACE
```

Related statements

`END INTERFACE`, `FUNCTION`, and `SUBROUTINE`

Related concepts

For related information, see the following:

- “Derived types” on page 39
- “Interface blocks” on page 152

INTRINSIC (statement and attribute)

Identifies an intrinsic procedure.

Syntax

The syntax of the type declaration statement with the INTRINSIC attribute is:

type , *attrib-list* :: *intrinsic-function-name-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE(*name*), etc.).

attrib-list

is a comma-separated list of attributes including INTRINSIC and optionally those attributes compatible with it, namely:

PRIVATE	PUBLIC

intrinsic-function-name-list

is a comma-separated list of *intrinsic-function*-names. (Note that subroutine names cannot appear in type statements, so that intrinsic subroutine names can only be identified as such by use of the INTRINSIC statement, described below.)

The syntax of the INTRINSIC statement is:

INTRINSIC *intrinsic-procedure-name-list*

where *intrinsic-procedure-name-list* is a comma-separated list of procedure names.

Note that, like the EXTERNAL statement, the INTRINSIC statement does not have optional colons.

Description

The INTRINSIC statement and attribute identifies a specific or generic name as that of an intrinsic procedure, enabling it to be used as an actual argument. (Only a specific function name—or a generic name that is the same as the specific name—can be used as an actual argument; see “Procedure dummy argument” on page 142.) The INTRINSIC statement is necessary to inform the compiler that a name is intrinsic and is not the name of a variable. Whenever an intrinsic name is passed as an actual argument and no other appearance of the name in the same scoping unit indicates that it is a procedure, it must be specified by the calling program in an INTRINSIC statement, or (if a function name) in a type declaration statement that includes the INTRINSIC attribute.

Each name can appear only once in an INTRINSIC statement and in at most one INTRINSIC statement within the same scoping unit. Also, a name cannot appear in both an EXTERNAL and an INTRINSIC statement within the same scoping unit.

Examples

```
SUBROUTINE subr  ! caller
  DOUBLE PRECISION :: dsin,x,y,func
  INTRINSIC dsin
  ...
  y = func(dsin,x)
  ...
END SUBROUTINE subr

DOUBLE PRECISION FUNCTION func(proc,y)  ! callee
  DOUBLE PRECISION :: y, proc
  ...
  func = proc(y)
  ...
END FUNCTION func
```

Related statements

EXTERNAL

Related concepts

For additional information about passing user-defined and intrinsic procedures as arguments, see “Procedure dummy argument” on page 142. Intrinsic procedures are described in “Intrinsic procedure specifications” on page 487.

LOGICAL

Declares entities of type logical.

Syntax

`LOGICAL` [*kind-spec*] [[, *attrib-list*] ::] *entity-list*

kind-spec

specifies the size of the logical entity in bytes. *kind-spec* takes the form:

([KIND=] *kind-param*)

where *kind-param* can be a named constant or a constant expression that has the integer value of 1, 2, 4, or 8. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

**len-param*

where *len-param* is the integer 1, 2, 4, or 8 (default = 4).

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds

initialization-expr

is a logical constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The LOGICAL statement is constrained by the rules for type declaration statements, including the requirement that it precede all executable statements.

As a portability extension, HP Fortran 90 allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*). If *name* appears with **len*, it overrides the length specified by LOGICAL**size*.

Examples

The following are valid declarations:

```
LOGICAL log1, log2
LOGICAL(KIND=2) :: log3
LOGICAL(2), PARAMETER :: test=.TRUE.
! initialize an array, using an array constructor
LOGICAL, DIMENSION(2) :: lvec=(/.TRUE.,.FALSE./)
! use the slash notation (an HP extension) to initialize
LOGICAL log1/.TRUE./, log2/.FALSE./  ! note, no double colon
! the following declarations are equivalent; the second uses the
! HP length specification extension
LOGICAL (KIND = 8) log8
LOGICAL*4 log8*8
```

Related statements

INTEGER

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80
- “KIND(X)” on page 537

MAP (extension)

Defines a union within a structure.

Syntax

```
MAP
   field-def
   ...
END MAP
```

field-def

is one of the following:

- A type declaration statement
- Another nested structure
- A nested record
- A union definition

Description

The `MAP` statement is an HP compatibility extension that is used with the `UNION` statement to define a union within a structure. For detailed information about the `MAP` and `UNION` statements, see "STRUCTURE (extension)" on page 437.

MODULE

Introduces a module.

Syntax

`MODULE` *module-name*

module-name
: is a unique module name.

Description

Modules are nonexecutable program units that can contain type definitions, object declarations, procedure definitions (module procedures), external procedure interfaces, user-defined generic names, and user-defined operators and assignments. Any such definitions not specified to be private to the module containing them are available to those program units that specify the module in a USE statement. Modules provide a convenient sharing and encapsulation mechanism for data, types, procedures, and procedure interfaces.

Examples

```
! Make data objects and a data type sharable via a module
MODULE shared
   COMPLEX gtx (100, 6)
   REAL, ALLOCATABLE :: y(:), z(:,:)
   TYPE peak_item
      REAL peak_val, energy
      TYPE(peak_item), POINTER :: next
   END TYPE peak_item
END MODULE shared

! Define a data abstraction for rational arithmetic via a module
MODULE rational_arithmetic
   TYPE rational
      PRIVATE
      INTEGER numerator, denominator
   END TYPE rational   ! Generic extension of =
   INTERFACE ASSIGNMENT (=)
      MODULE PROCEDURE eqrr, eqri, eqir
   END INTERFACE
   INTERFACE OPERATOR (+)   ! Generic extension of +
      MODULE PROCEDURE addrr, addri, addir
   END INTERFACE
   ...
   CONTAINS
     FUNCTION eqrr (. . .)     ! A specific definition of =
     ...
     FUNCTION addrr (. . .)    ! A specific definition of +
     ...
END MODULE rational_arithmetic
```

Related statements `CONTAINS`, `END`, `PRIVATE`, `PUBLIC`, and `USE`

Related concepts For more information about modules, see “Modules” on page 161.

MODULE PROCEDURE

Specifies module procedures in a generic interface.

Syntax

`MODULE PROCEDURE` *module-procedure-name-list*

module-procedure-name-list

is a comma-separated list of *module-procedure-names*.

Description

A `MODULE PROCEDURE` statement appears within an interface block. It is used when the specification is generic and a specific procedure is defined within the module rather than as an external procedure. The `MODULE PROCEDURE` statement only names the subprograms; it does not contain the definition of the interface. The named subprograms must be defined within the current module or within another module that is accessible by use association.

Examples

```
MODULE path
! module data environment; module procedures contained in this
! module have access to this data environment
REAL x, y, z
! Generic name substance for procedures air and water
INTERFACE substance
   MODULE PROCEDURE air, water
END INTERFACE
INTERFACE OPERATOR (*)
   MODULE PROCEDURE rational_multiply
END INTERFACE
...
! Module procedures are preceded by CONTAINS
CONTAINS
   SUBROUTINE air (contents)
   ...
   END SUBROUTINE air
   SUBROUTINE water (x, a, z)
      ! x is a dummy argument, y is from the module data
      !   environment
      a = x + y
      ...
   END SUBROUTINE water
   FUNCTION rational_multiply (x, y)
      TYPE (rational) :: rational_multiply
      TYPE (rational), INTENT (IN) :: x, y
      rational_multiply = ...
      ...
   END FUNCTION rational_multiply
END MODULE path
```

Related statements FUNCTION, SUBROUTINE, and INTERFACE

Related concepts For information about module procedures, see “Module program unit” on page 161.

NAMELIST

Names a group of variables for I/O processing.

Syntax

```
NAMELIST /group-name/var-list [ [ ,]/group-name/var-list ]...
```

group-name

is a unique namelist group name.

var-list

is a comma-separated list of scalar and array variable names.

Description

The `NAMELIST` statement declares *var-list* as a namelist group and associates the group with *group-name*.

Variables appearing in *var-list* may be of any type, including objects of derived types or their components, saved variables, variables on the local stack, and subroutine parameters. The following, however, are not allowed:

- Record or composite references
- Pointers or their targets
- Automatic objects
- Allocatable array
- Character substrings
- Assumed-size array parameters
- Adjustable-size array parameters
- Assumed-size character parameters
- Individual components of a derived type object

The *var-list* explicitly defines which items may be read or written in a namelist-directed I/O statement. It is not necessary for every item in *var-list* to be defined in namelist-directed input, but every input item must belong to the namelist group. The order of items in *var-list* determines the order of the values written in namelist-directed output.

More than one NAMELIST statement with the same *group-name* may appear within the same scoping unit. Each successive *var-list* in multiple NAMELIST statements with the same *group-name* is treated as a continuation of the list for *group-name*.

The same variable name may appear in different NAMELIST statements within the same scoping unit.

Examples

```
PROGRAM
  INTEGER i, j(10)
  CHARACTER*10 c
  NAMELIST /n1/ i, j, c
  ! Define the namelist group n1
  READ (UNIT=5,NML=n1)
  WRITE (6, n1)
END
```

When this program is compiled and executed with the following input record:

```
&n1
j(8) = 6, 7, 8
i = 5c = 'xxxxxxxxx'
j = 5*0, -1, 2
c(2:6) = 'abcde'
/
```

its output is:

```
 &n1
 I      = 5
 J      = 0 0 0 0 0 -1 2 6 7 8
 C      ='xabcdexxx'
 /
```

Related statements

ACCEPT, OPEN, INQUIRE, PRINT, READ, and WRITE

Related concepts

Namelist-directed I/O is described in “Namelist-directed I/O” on page 183.

NULLIFY

Disassociates a pointer from a target.

Syntax

NULLIFY (*pointer-object-list*)

pointer-object-list

is a comma-separated list of variable names and derived-type components.

Description

The NULLIFY statement disassociates a pointer from any target. A NULLIFY statement is also used to change the status of a pointer from undefined to disassociated.

Examples

The following example shows the declaration and use of a variable with the pointer attribute:

```
REAL, TARGET :: value  ! value can be target
REAL, POINTER :: pt    ! for the pointer
pt.pt => value         ! Associate pt with value
NULLIFY (pt)           ! Disassociate pt

! ASSOCIATED intrinsic is valid in next statement if (and only
! if) pt has been previously allocated, assigned (as above), or
! nullified (as above)
IF (.NOT.ASSOCIATED(pt)) pt => x
```

The next example shows how a derived type can be used in list processing applications:

```
TYPE list_node
   INTEGER value
   TYPE (list_node), POINTER :: next
END TYPE list_node
TYPE (list_node), POINTER :: list
ALLOCATE (list)        ! Create new list node
list % value = 28      ! Initialize data field
NULLIFY (list % next) ! Nullify pointer to the next node
```

Related statements

ALLOCATE, DEALLOCATE, POINTER, and TARGET

Related concepts

For information about pointers, see “Pointers” on page 47.

ON (extension)

Specifies the action to take when program execution is interrupted.

Syntax

`ON` *interrupt-condition action*

interrupt-condition

is the interrupt to be handled, either an arithmetic error or a keyboard interrupt.

action

is one of the following:

- `CALL` *trap-routine*
- `ABORT`
- `IGNORE`

where:

trap-routine

is an external subroutine name.

Description

The `ON` statement is an HP extension. It is an executable statement that specifies the action to be taken after the occurrence of an exception that interrupts program execution.

For each *interrupt-condition*, you can specify one of the following actions:

- `CALL`: specifies a subroutine to be called.
- `ABORT`: causes the program to abort.
- `IGNORE`: causes the interrupt to be ignored.

Table 51 lists the range of values for *interrupt-condition*. The first column identifies the type of trap; the second gives the keywords that must appear on the `ON` statement, immediately following the word `ON`; and the third column gives equivalent keywords you can specify instead of those in the second column. For example, the following `ON` statement causes the program to trap an attempt to divide by zero with 8-byte floating-point operands, passing control to a user-written trap handler called `div_zero_trap`:

```
ON REAL(8) DIV 0 CALL trap_div_by_zero
```

The following ON statement does the same thing, but it specifies the equivalent keywords from the third column of the table:

```
ON DOUBLE PRECISION DIV 0 CALL trap_div_by_zero
```

Table 51 Exceptions handled by the ON statement

Exceptions	Exception keywords	Alternate keywords
Division by zero	REAL(4) DIV 0	REAL DIV 0
	REAL(8) DIV 0	DOUBLE PRECISION DIV 0
	REAL(16) DIV 0	(none)
	INTEGER(2) DIV 0	INTEGER*2 DIV 0
	INTEGER(4) DIV 0	INTEGER DIV 0
Overflow	REAL(4) OVERFLOW	REAL OVERFLOW
	REAL(8) OVERFLOW	DOUBLE PRECISION OVERFLOW
	REAL(16) OVERFLOW	(none)
	INTEGER(2) OVERFLOW	INTEGER*2 OVERFLOW
	INTEGER(4) OVERFLOW	INTEGER OVERFLOW
Underflow	REAL(4) UNDERFLOW	REAL UNDERFLOW
	REAL(8) UNDERFLOW	DOUBLE PRECISION UNDERFLOW
	REAL(16) UNDERFLOW	(none)
Invalid (illegal) operation	REAL(4) ILLEGAL	REAL ILLEGAL
	REAL(8) ILLEGAL	DOUBLE PRECISION ILLEGAL
	REAL(16) ILLEGAL	(none)
Inexact result	REAL(16) INEXACT	(none)
	REAL(4) INEXACT	REAL INEXACT
	REAL(8) INEXACT	DOUBLE PRECISION INEXACT
Control-C	CONTROLC	(none)

To use the ON statement to trap for integer overflow, you must also include the HP CHECK_OVERFLOW directive. This is described in the *HP Fortran 90 Programmer's Guide*.

Using the ON statement at optimization levels 2 and above is restricted. When compiling at optimization level 2 or above, the optimizer makes assumptions about the program that do not take into account the behavior of procedures called by the ON statement. Such procedures must therefore be “well-behaved”—in particular, they must meet the following criteria:

- The ON procedure must not assume that any variable in the interrupted procedure or in its caller has its current value. (The optimizer may have placed the variable in a register to be stored there until after the call to the interrupted procedure is complete.)
- The ON procedure must not change the value of any variable in the interrupted procedure or in its caller if the effect of the ON procedure is to return program control to the point of interrupt.

NOTE If you include the ON statement in a program that is compiled at optimization level 2 or higher and the program takes an exception, the results may vary from those you would get from the unoptimized program or from the same program without the ON statement.

Examples

The following example uses the ON statement to call the procedure trap_div_by_zero if the function do_div is passed 0 in argument y. If trap_div_by_zero is called, it prints an error message and assigns 0 to the result.

```
REAL FUNCTION do_div(x, y)
  REAL :: x, y
  ON REAL DIV 0 CALL trap
  do_div = x/y  ! causes an interrupt if y = 0
  RETURN
END FUNCTION do_div

SUBROUTINE trap(res)
  REAL :: res
  PRINT *, "Don't do that."
  res = 0
END SUBROUTINE trap
```

Related concepts

The *HP Fortran 90 Programmer's Guide* provides detailed information about using the ON statement, including example programs that use the ON statement.

OPEN

Connects file to a unit.

Syntax

`OPEN (`*io-specifier-list*`)`

io-specifier-list

is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*unit*

specifies the unit to connect to an external file. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword `UNIT=` is omitted, *unit* must be the first item in *io-specifier-list*.

`ACCESS=`*character-expression*

specifies the method of file access. *character-expression* can be one of the following arguments:

`'DIRECT'`	Open file for direct access.
`'SEQUENTIAL'`	Open file for sequential access (default).
`' POSITION= APPEND'`	To open a file for append (to position the file just before the end-of-file record)

`ACTION=`*character-expression*

specifies the allowed data-transfer operations. *character-expression* can be one of the following arguments:

`'READ'`	Do not allow `WRITE` and `ENDFILE` statements.
`'WRITE'`	Do not allow `READ` statements.
`'READWRITE'`	Allow any data transfer statement (default).

`BLANK=`*character-expression*

specifies treatment of blanks within numeric data on input. This specifier is applicable to formatted input only. *character-expression* can be one of the following arguments:

`'NULL'`	Ignore blanks (default).
`'ZERO'`	Substitute zeroes for blanks.

`DELIM=`*character-expression*

specifies the delimiter to use (if any) when delimiting character constants in list-directed and namelist-directed formatting. This specifier is applicable to formatted output only. *character-expression* can be one of the following arguments:

`'APOSTROPHE'`	Use the apostrophe to delimit character constants in list-directed and namelist-directed formatting.
`'QUOTE'`	Use double-quotation marks to delimit character constants in list-directed and namelist-directed formatting.
`'NONE'`	Use no delimiter to delimit character constants in list-directed and namelist-directed formatting (default).

`ERR=`*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

`FILE=`*character-expression*

specifies the name of the file to be connected to *unit*. *character-expression* can also be the ASCII representation of a device file. If this specifier does not appear in the `OPEN` statement, a temporary scratch file is created.

`FORM=`*character-expression*

specifies whether the file is connected for formatted or unformatted I/O. *character-expression* can be one of the following arguments:

`'FORMATTED'`	Specify formatted I/O. If the file is to be opened for sequential access, this is the default.
`'UNFORMATTED'`	Specify unformatted I/O. If the file is to be opened for direct access, this is the default.

`IOSTAT=`*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

`PAD=`*character-expression*

specifies whether or not to pad the input record with blanks if the record contains fewer characters than required by the format specification. This specifier is applicable to formatted input only. *character-expression* can be one of the following arguments:

`'YES'`	Pad input records with blanks (if necessary) to fill it out to length required by format specification (default).
`'NO'`	Do not pad input record with blanks if it is not as long as record specified by format specification.

`POSITION=`*character-expression*

specifies the position of an existing file to be opened for sequential access. *character-expression* can be one of the following arguments:

`'ASIS'`	Leave file position unchanged (default).
`'REWIND'`	Position the file at its start.
`'APPEND'`	Position the file just before the end-of-file record.

If the file to be opened does not exist, this specifier is ignored. New files are always positioned at their start.

`RECL=`*integer-expression*

specifies the length of each record in a file to be opened for direct access. The length is measured in characters (bytes). This specifier must be present when a file is opened for direct access and is ignored if file is opened for sequential access.

`STATUS=`*character-expression*

specifies the state of the file when it is opened. *character-expression* can be one of the following arguments:

`'OLD'`	Open an existing file. `FILE=` must also be specified and the named file must exist.
`'NEW'`	Create a new file. `FILE=` must also be specified and the named file must not exist.
`'UNKNOWN'`	If the file named in `FILE=` exists, open it with the status of `OLD`; if it does not exist, open it with the status of `NEW`. This is the default status.
`'REPLACE'`	If the file does not exist, create it with a status of `OLD`; if it does exist, delete it and open it with a status of `NEW`. If `STATUS='REPLACE'` is specified, `FILE=` must also be specified.
`'SCRATCH'`	Create a scratch file. `FILE=` specifier must *not* be specified. For information about scratch files, see “Scratch files” on page 173.

Description

The `OPEN` statement connects a unit to a file so that data can be read from or written to that file. Once a file is connected to a unit, the unit can be referenced by any program unit in the program.

I/O specifiers do not have to appear in any specific order in the `OPEN` statement. However, if the optional keyword `UNIT=` is omitted, *unit* must be the first item in the list.

Only one unit can be connected to a file at a time. That is, the same file cannot be connected to two different units. Attempting to open a file that is connected to a different unit will produce undefined results.

However, multiple OPENs can be performed on the same unit. In other words, if a unit is connected to a file that exists, it is permissible to execute another OPEN statement for the same unit. If FILE= specifies a different file, the previously opened file is automatically closed before the second file is connected to the unit. If FILE= specifies the same file, the file remains connected in the same position; the values of the BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers can be changed, but attempts to change the values of any of the other specifiers will be ignored.

Examples

The following examples illustrate different uses of the OPEN statement.

Opening a file for sequential access

The following OPEN statement connects the existing file inv to unit 10 and opens it (by default) for sequential access. Only READ statements are permitted to perform data transfers. If an error occurs, control passes to the executable statement labeled 100 and the error code is placed in the variable ios:

```
OPEN(10, FILE='inv', ERR=100, IOSTAT=ios,  &
     ACTION='READ', STATUS='OLD')
```

Opening a file for direct access

The following OPEN statement opens the file whose name is contained in the variable next1, connecting it to unit 4 as a formatted, direct-access file with a record length of 50 characters:

```
OPEN(ACCESS="DIRECT", UNIT=4, RECL=50,  &
     FORM="FORMATTED", FILE=next1)
```

Opening a device for I/O transfers

The next example connects the system device /dev/console to unit 6; all data transfers that specify unit 6 will go to this device:

```
OPEN(6,FILE='/DEV/CONSOLE')
```

Opening a scratch file

The following two `OPEN` statements produce the same results: open a scratch file that is connected to unit 19 (if the `FILE=`*name* specifier had appeared in the first statement, the named file would have been opened instead):

```
OPEN (UNIT=19)
OPEN (UNIT=19, STATUS="SCRATCH")
```

I/O specifiers in an OPEN statement

Because the I/O specifiers that can be used in an `OPEN` statement do not have to appear in any specific order, the following three `OPEN` statements are all equivalent:

```
OPEN(UNIT=3, STATUS='NEW', FILE='OUT.DAT')
OPEN(3, STATUS='NEW', FILE='OUT.DAT')
OPEN(STATUS='NEW', FILE='OUT.DAT', UNIT=3)
```

Note, however, that in the second `OPEN` statement the number 3 must appear first because of the omission of the optional keyword `UNIT=`. Thus, the following `OPEN` statement is illegal:

```
OPEN(STATUS='NEW', 3, FILE='OUT.DAT') ! illegal
```

Related statements `CLOSE`, `INQUIRE`, `READ`, and `WRITE`

Related concepts For information about I/O concepts and examples of programs that perform I/O, see Chapter 8, “I/O and file handling,” on page 171. For information about I/O formatting, see Chapter 9, “I/O formatting,” on page 205.

OPTIONAL (statement and attribute)

Identifies optional arguments for procedures.

Syntax

The syntax of the type declaration statement with the `OPTIONAL` attribute is:

type , *attrib-list* :: *dummy-argument-name-list*

type

is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (` *name*`)`, etc.).

attrib-list

is a comma-separated list of attributes including `OPTIONAL` and optionally those attributes compatible with it, namely:

DIMENSION	INTENT	TARGET
EXTERNAL	POINTER	VOLATILE

dummy-argument-name-list

is a comma-separated list of *dummy-argument-names*.

The syntax of the `OPTIONAL` statement is:

`OPTIONAL [::]` *dummy-argument-name-list*

Description

If a dummy argument has the `OPTIONAL` attribute, the corresponding actual argument need not appear in a procedure reference. In cases where there are arguments that generally do not change from one reference to another, it is convenient to specify that the arguments are optional and provide default values for them. They can then be omitted from references in these general cases. The presence of an optional argument in a procedure may be determined by using the `PRESENT` intrinsic function.

Many uses of the `ENTRY` statement in FORTRAN 77 programs can be replaced by the use of optional arguments.

The following restrictions apply to the use of the `OPTIONAL` attribute:

- The OPTIONAL attribute may be specified only for dummy arguments. It may occur in a subprogram and in any corresponding interface body.
- An optional dummy argument whose actual argument is not present may not be referenced or defined (or invoked if it is a dummy procedure), except that it may be passed to another procedure as an optional argument and will be considered not present.
- When an argument is omitted in a procedure reference, all arguments that follow it must use the keyword form.
- If a procedure has an optional argument, the procedure interface must be explicit.

Examples

The following are two examples of the OPTIONAL statement. In the first example, the call to the subroutine trip can legally omit the path argument because it has the OPTIONAL attribute:

```
CALL TRIP ( distance = 17.0 ) ! path is omitted
SUBROUTINE trip ( distance, path )
   OPTIONAL distance, path
```

In the next example, the subroutine plot uses the PRESENT function to determine whether or not to execute code that depends on the presence of arguments that have the OPTIONAL attribute:

```
SUBROUTINE plot (pts, o_xaxis, o_yaxis, smooth)
   TYPE (point) pts
   REAL, OPTIONAL :: o_xaxis, o_yaxis
   ! Origin - default (0.,0.)
   LOGICAL, OPTIONAL :: smooth
   REAL ox, oy
   IF (PRESENT (o_xaxis)) THEN
     ox = o_xaxis
   ELSE
     ox = 0.
   ! Note that the o_xaxis dummy argument cannot be referenced if
   ! the actual argument is not present.  The same applies
   ! to o_yaxis (below).
   END IF
   IF (PRESENT (o_yaxis)) THEN
     oy = o_yaxis
   ELSE
     oy = 0.
   END IF
   IF (PRESENT(smooth)) THEN
      IF (smooth) THEN
         ...                     ! Smooth algorithm
         RETURN
      END IF
   END IF
```

```
      ...                      ! Plot points
END SUBROUTINE plot

! Some valid calls to plot.
CALL plot (points)
CALL plot (observed, o_xaxis = 100.,  o_yaxis = 1000.)
CALL plot (random_pts, smooth = .TRUE.)
```

Related statements

SUBROUTINE and FUNCTION

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Arguments” on page 139
- The description of the PRESENT intrinsic in Chapter 11, “Intrinsic procedures,” on page 475

OPTIONS (extension)

Lowers the optimization level used by the HP Fortran 90 compiler.

Syntax

`OPTIONS +O`*n*

where `+O`*n* (or `-O`*n*) specifies a level of optimization that is equal to or less than the level specified on the command line.

Description

The `OPTIONS` statement is an extension of HP Fortran 90 and is used to specify a level of optimization that is equal to or less than the level specified on the command line. If the level specified by the `OPTIONS` statement is higher than that specified on the command line, the statement is ignored.

The `OPTIONS` statement must be placed outside all program units. The changed level of optimization applies to the beginning of the next program unit and remains in effect for all succeeding program units or until superseded by another `OPTIONS` statement or by the `!$HP$ OPTIMIZE` directive.

The `OPTIONS` statement differs from the `OPTIMIZE` directive in that the `OPTIMIZE` directive enables or disables optimization but does not change the optimization level. The `!$HP$ OPTIMIZE` directive is described in the *HP Fortran 90 Programmer's Guide*.

The `OPTIMIZE` directive takes precedence over the `OPTIONS` statement: when the `OPTIMIZE` directive is used to disable optimization, any subsequent `OPTIONS` statement has no effect until a later directive enables optimization.

Examples

In the following example, the first OPTIONS statement optimizes the subroutine go_fast at optimization level 3. The second OPTIONS statement lowers the optimization level to 2.

```
OPTIONS +O3
SUBROUTINE go_fast
...
END SUBROUTINE go_fast

OPTIONS +O2
SUBROUTINE not_so_fast
...
END SUBROUTINE not_so_fast
```

PARAMETER (statement and attribute)

Defines a named constant.

Syntax

A type declaration statement with the PARAMETER attribute is:

type, *attrib-list* :: *cname1* = *cexpr1*[, *cname2* = *cexpr2*]...

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.).

attrib-list

is a comma-separated list of attributes including PARAMETER and optionally those attributes compatible with it, namely:

DIMENSION	PUBLIC
PRIVATE	SAVE

Specifying the SAVE attribute in a PARAMETER statement has no effect.

cname

is the name that will represent the constant.

cexpr

is an initialization expression that evaluates to the constant represented by *cname*. In the case of an array constant, *cexpr* must be an array constructor. In the case of a derived type constant, *cexpr* must be a structure constructor.

The syntax of the PARAMETER statement is:

PARAMETER (*cname1* = *cexpr1* [, *cname2* = *cexpr2*]...)

Description

The PARAMETER statement associates a symbolic name with a constant. A symbolic name defined in a PARAMETER statement is known as a **named constant**. A named constant must not become defined more than once in a program unit. Once defined, it can be used only as a named constant. This means that a named constant cannot be assigned a value like a variable.

When the PARAMETER attribute is used, the value of the named constant must be provided by the initialization part of the statement in which the PARAMETER attribute appears.

The type of a named constant is determined by the implicit typing rules, unless its type is specified by a type declaration statement prior to its first appearance in a PARAMETER statement or by a type declaration statement that includes PARAMETER as one of its attributes. If a PARAMETER statement declares and implicitly types a named constant, the named constant may appear in a subsequent type declaration or IMPLICIT statement, but only to confirm the type of the named constant.

When the type of the symbolic name and the constant do not agree, the value of the named constant is assigned in accordance with assignment statement type-conversion rules, as given in Table 14.

The following rules apply to type agreement between the constant and the symbolic name:

- If *cname* is of numeric type, *cexpr* must be an arithmetic constant expression.
- If *cname* is of type character, the corresponding *cexpr* must be a character constant expression.
- If *cname* is of type logical, the corresponding *cexpr* may be either an arithmetic or logical constant expression.

Any symbolic name of a constant that appears in *cexpr* must have been defined previously in the same or a different PARAMETER statement in the same program unit. For example, the expression in the second PARAMETER statement below is built from the expression in the first PARAMETER statement, and is legal:

```
PARAMETER (limit = 1000)
PARAMETER (limit_plus_1 = limit + 1)
```

The logical operators (.EQ., .NE., .LT., .LE., .GT., and .GE.), as well as the following intrinsic functions, can appear in the PARAMETER statement:

ABS	IAND	IXOR	MAX
CHAR	ICHAR	LEN	MIN
CMPLX	IEOR	LGE	MOD
CONJB	IMAG	LGT	NINT
DIM	IOR	LLE	NOT
DPROD	ISHFT	LLT	

If these intrinsic functions are used in a PARAMETER statement, their arguments must be constants.

If the named constant is of type character and its length is not specified, the length must be specified in a type declaration statement or IMPLICIT statement prior to the first appearance of the named constant. Its type and/or length must not be changed by subsequent statements, including IMPLICIT statements. If a symbolic name of type CHARACTER*(*) is defined in a PARAMETER statement, its length becomes the length of the expression assigned to it.

If the named constant is an array, the bounds must be explicit and determined by an initialization expression.

Once such a symbolic name is defined, that name can appear in any subsequent statement of the defining program unit as a constant in an expression or DATA statement.

Examples

```
! PARAMETER used in a type declaration statement as an attribute
REAL, DIMENSION(4), PARAMETER :: const =  &
    (/1.2, 1.45, 0.9, 24.3/)

INTEGER year
! PARAMETER used as a statement
PARAMETER year = 1996

! Type declaration statement declaring a derived-type constant
TYPE (postal_info), PARAMETER :: package = &
     postal_info (9.5, (/10.0, 5.5, 2.25/) )
```

Related concepts

For information about the type declaration statement, see “Type declaration for intrinsic types” on page 24.

PAUSE

Temporarily stops program execution.

Syntax

PAUSE *pause-code*

pause-code

is a character constant or a list of up to 5 digits.

Description

The PAUSE statement suspends program execution and prints a message, depending on whether digits, characters, or nothing has been specified in the PAUSE statement:

- If digits, the message “PAUSE *digits*” is written to standard error.
- If a character expression, the message “PAUSE *character-expression*” is written to standard error.
- If nothing appears after PAUSE, the word “PAUSE” is written to standard error.

After displaying the appropriate message, the PAUSE statement writes to standard output one of two messages that give information on resuming the program. If the standard input device is a terminal, the message is:

```
To resume program execution, type GO.
```

At this point the program is suspended and remains so until the operator types the word GO and presses the Return key. The program will terminate if anything other than GO is entered.

If the standard input device is other than a terminal, the message is:

```
To resume execution, execute a kill -15 pid  &
command
```

where *pid* is the unique process identification number of the suspended program. This command can be issued at any terminal at which the user is logged in.

Examples

```
! Write ”PAUSE 7777” to standard error
PAUSE 7777

! Write ”PAUSE MOUNT TAPE” to standard error
PAUSE 'MOUNT TAPE'

! Write ”PAUSE” to standard error
PAUSE
```

Related statements STOP

Related concepts For information about flow control statements, see “Flow control statements” on page 112.

POINTER (Cray-style extension)

Declares Cray-style pointers and their objects.

Syntax

`POINTER (`*pointer1*`, `*pointee1*`) [, (`*pointer2*`, `*pointee2*`)]...`

pointer

is a pointer.

pointee

is a variable name or array declarator.

Description

HP Fortran 90 supports both the standard Fortran 90 `POINTER` statement as well as the Cray-style `POINTER` statement. The Cray-style `POINTER` statement is supported for compatibility with older, FORTRAN 77 programs. The following information applies only to the Cray-style `POINTER` statement; the Fortran 90 `POINTER` statement is described in "POINTER (statement and attribute)" on page 400.

The following restrictions apply to *pointer*:

- It should be of type `INTEGER(4)`. If it is not, the compiler interprets its type as `INTEGER(4)` regardless of other implicit or explicit type declarations.
- It cannot be declared of any other data type.
- Another pointer cannot point to it.
- It cannot appear in a `PARAMETER` or `DATA` statement.
- It cannot be in a derived type object.

You can increase the size of *pointer* with the `+autodbl` or `+autodbl4` option; see "Option Descriptions" on page 576.

pointee may be of any type, including an array, a derived type, a record, or a character string.

The following restrictions apply to *pointee*:

- It cannot be a dummy argument, function name, function value, common block element, automatic object, generic interface block name, or derived type.

- It cannot be used in a COMMON, DATA, EQUIVALENCE, or NAMELIST statement.
- It cannot have any of the following attributes: ALLOCATABLE, EXTERNAL, INTENT, INTRINSIC, OPTIONAL, PARAMETER, POINTER, SAVE, and TARGET.
- Pointees that are arrays with nonconstant bounds can be used only in subroutines and functions, not in main programs.
- Variables used in an array-bound expression that appears in a POINTER statement must be either subprogram formal arguments or common block variables. The value of the expression cannot change after subprogram entry.

You associate memory with a pointer by assigning it the address of an object. Typically, this is done with the libU77 function, LOC. The LOC function returns the address of its argument, which can be assigned to a pointer. The following example assigns 0 to the pointee i:

```
INTEGER i, j
POINTER (p, i)

p = LOC(j)
j = 0
```

You can also use the MALLOC intrinsic to allocate memory from the heap and assign its return value to a pointer. Once you are done with the allocated memory, you should use the FREE intrinsic to release the memory so that it is available for reuse.

If you are using the pointer to manipulate a device that resides at a fixed address, you can assign the address to the pointer, using either an integer constant or integer expression.

Under certain circumstances, Cray-style pointers can cause erratic program behavior—especially if the program has been optimized. To ensure correct behavior, observe the following:

- Subroutines and functions must not save the address of any of their arguments between calls.
- A function must not return the address of any of its arguments.
- Only those variables whose addresses are explicitly taken with the LOC function must be referenced through a pointer.

Examples

In the following example, the intrinsic MALLOC returns either the address of the block of memory it allocated or 0 if MALLOC was unable to allocate enough memory. The formal argument nelem contains the number of array elements and is multiplied by 4 to obtain the number of bytes that MALLOC is to allocate. The FREE intrinsic returns memory to the heap for reuse.

```
SUBROUTINE print_iarr(nelem)
  POINTER (p, iarr(nelem))

  p = MALLOC( 4*nelem )

  IF (p.EQ.0) THEN
    PRINT *, 'MALLOC failed.'
  ELSE
    DO i = 1,nelem
      iarr(i) = i
    END DO

    PRINT *, (iarr(i),i=1,nelem)
    CALL FREE( p )
  ENDIF
  RETURN
END SUBROUTINE print_iarr
```

Related statements

POINTER (standard Fortran 90)

Related concepts

For related information, see the following:

- “Pointers” on page 47
- The description of the LOC routine in Table 64
- The descriptions of the MALLOC and FREE intrinsics in Chapter 11, “Intrinsic procedures,” on page 475

POINTER (statement and attribute)

Specifies variables with the POINTER attribute.

Syntax

The syntax of a type declaration statement with the POINTER attribute is:

type, *attrib-list* :: *dummy-argument-name-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.).

attrib-list

is a comma-separated list of attributes including POINTER and optionally those attributes compatible with it, namely:

DIMENSION	PRIVATE	SAVE
OPTIONAL	PUBLIC	

dummy-argument-name-list

is a comma-separated list of *dummy-argument-names*.

The syntax of the POINTER statement is:

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

object-name

is a data object or function result.

deferred-shape-spec-list

is a comma-separated list of colons.

Description

A POINTER attribute or statement specifies that the named variables may be pointers to some target object. Pointers provide a capability for creating dynamic objects, such as dynamic-sized arrays and linked lists. An object with a pointer attribute initially has no space reserved for its target. A pointer is assigned space for its target when an ALLOCATE statement is executed or when it is assigned to point to a target using a pointer assignment statement.

Examples

In the first example, two array pointers are declared and used.

```
! Extents are not specified; they are determined during execution
REAL, POINTER :: weight (:,:,:)
REAL, POINTER :: w_reg (:,:,:)

READ *, i, j, k
ALLOCATE (weight (i, j, k))   ! create weight

! w_reg is an alias for an array section
w_reg => weight (3:i-2, 3:j-2, 3:k-2)
avg_w = sum (w_reg) / ((i-4) * (j-4) * (k-4))

DEALLOCATE (weight)  ! weight no longer needed
```

The next example illustrates the use of pointers in a list-processing application.

```
TYPE link
   REAL value
   TYPE (link), POINTER :: next
END TYPE link

TYPE(link), POINTER :: list, save_list
NULLIFY (list)       ! Initialize list
DO
   READ (*, *, IOSTAT = no_more) value
   IF (no_more /= 0) EXIT
   save_list => list
   ALLOCATE (list)   ! Add link to head of list
   list % value = value
   list % next => save_list
END DO
! Linked list removed when no longer needed
DO
   IF (.NOT.ASSOCIATED (list) ) EXIT
   save_list => list % next
   DEALLOCATE (list)
   list => save_list
END DO
```

Related statements

ALLOCATE, DEALLOCATE, NULLIFY and TARGET

Related concepts

For related information, see the following:

- “Pointers” on page 47
- “Pointer assignment” on page 97
- The description of the ASSOCIATED intrinsic in Chapter 11, “Intrinsic procedures,” on page 475.

PRINT

Writes to standard output.

Syntax

The syntax of the `PRINT` statement can take one of two forms:

- Formatted and list-directed syntax:

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

- Namelist-directed syntax:

  `PRINT` *name*

format

is one of the following:

- An asterisk (*), specifying list-directed I/O.
- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification.

name

is the name of a namelist group, as previously defined by a `NAMELIST` statement . Using the namelist-directed syntax, the `PRINT` statement sends data in the namelist group to standard output. To direct output to a connected file, you must use the `WRITE` statement and include the `NML=` specifier.

output-list

is a comma-separated list of data items for output. The data items can include expressions and implied-`DO` lists.

Description

The `PRINT` statement transfers data from memory to standard output. (Unit 6 is preconnected to the HP-UX standard output.) The `PRINT` statement can be used to perform formatted, list-directed, and namelist-directed I/O only.

To direct output to a connected file, use the `WRITE` statement.

Examples

The examples in this section illustrate different uses of the PRINT statement.

Formatted output

The following statement writes the contents of the variables num and des to standard output, using the format specification in the FORMAT statement at label 10:

```
PRINT 10, num, des
```

List-directed output

The following statement uses list-directed formatting to print the literal string x= and the value of the variable x:

```
PRINT *, 'x=', x
```

Embedded format specification

The following statement uses an embedded format specification to print the same output:

```
PRINT '(A2, F8.2)', 'x=', x
```

Namelist-directed output

The following statement prints all variables in the namelist group coord, using namelist-directed formatting:

```
PRINT coord
```

Related statements

FORMAT and WRITE

Related concepts

For related information, see the following:

- “List-directed I/O” on page 179
- “Embedded format specification” on page 237
- “Implied-DO loop” on page 194

PRIVATE (statement and attribute)

Prevents access to module entities by use association.

Syntax

The syntax of a type declaration statement with the PRIVATE attribute is:

type, *attrib-list* :: *access-id-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.).

attrib-list

is a comma-separated list of attributes including PRIVATE and optionally those attributes compatible with it, namely:

ALLOCATABLE	INTRINSIC	SAVE
DIMENSION	PARAMETER	TARGET
EXTERNAL	POINTER	

access-id-list

is a comma-separated list of one or more of the following:

- *constant-name*
- *variable-name*
- *procedure-name*
- *defined-type-name*
- *namelist-group-name*
- OPERATOR (*operator*)
- ASSIGNMENT (=)

The syntax of the PRIVATE statement is:

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

Description

The PRIVATE attribute may appear only in the specification part of a module. The default accessibility in a module is PUBLIC; it can be changed to PRIVATE using a statement without a list. However, only one PRIVATE accessibility statement without a list is permitted in a module.

The PRIVATE attribute in a type statement or in an accessibility statement restricts the accessibility of entities such as module variables, type definitions, functions, and named constants. USE statements may restrict accessibility further.

A derived type may contain a PRIVATE attribute or an internal PRIVATE statement, if it is defined in a module. The internal PRIVATE statement in a type definition makes the components unavailable outside the module even though the type itself might be available.

The PRIVATE statement may also be used to restrict access to subroutines, generic specifiers, and namelist groups.

The PRIVATE specification for a generic name, operator, or assignment does not apply to any specific name unless the specific name is the same as the generic name.

Examples

```
MODULE fourier
   REAL :: x, y, z  ! PUBLIC (default)
   COMPLEX, PRIVATE :: fft  ! PRIVATE, accessible only in module
   TYPE (structure_name), PRIVATE ::  structure_a, structure_b
   ! a, b and c are accessible only within this module
   PRIVATE  a, b, c
   ! r, s, and t are accessible outside the module
   PUBLIC  r, s, t
END MODULE fourier

MODULE place
   PRIVATE   ! Change default accessibility to PRIVATE
   INTERFACE OPERATOR (.st.)
      MODULE PROCEDURE xst
   END INTERFACE

   ! make .st. public; everything else is private
   PUBLIC OPERATOR (.st.)
   LOGICAL, DIMENSION (100) :: lt
   CHARACTER(20) :: name
   INTEGER ix, iy
END MODULE place
```

Related statements

PUBLIC and USE

Related concepts

For related information, see the following:

- "Type declaration for intrinsic types" on page 24
- "Modules" on page 161

PROGRAM

Identifies the main program unit.

Syntax

`PROGRAM` *name*

name

is the name of the program.

Description

The optional `PROGRAM` statement assigns a name to the main program unit. *name* does not have to match the main program's filename. However, if the corresponding `END PROGRAM` statement specifies a name, it must match *name*.

If the `PROGRAM` statement is specified, it must be the first statement in the main program unit.

Examples

```
! A program with a name
PROGRAM main_program
PRINT *, 'This program doesn't do much.'
END PROGRAM main_program
```

Related statements

`END`

Related concepts

For information about the main program unit, see "Main program" on page 125.

PUBLIC (statement and attribute)

Enables access to module entities by use association.

Syntax

The syntax of a type declaration statement with the `PUBLIC` attribute is:

type, *attrib-list* :: *access-id-list*

type

is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE` (*name*), etc.).

attrib-list

is a comma-separated list of attributes including `PUBLIC` and optionally those attributes compatible with it, namely:

ALLOCATABLE	INTRINSIC	SAVE
DIMENSION	PARAMETER	TARGET
EXTERNAL	POINTER	VOLATILE

access-id-list

is a comma-separated list of one or more of the following:

- *constant-name*
- *variable-name*
- *procedure-name*
- *defined-type-name*
- *namelist-group-name*
- `OPERATOR` (*operator*)
- `ASSIGNMENT ( = )`

The syntax of the PUBLIC statement is:

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

Description

The PUBLIC attribute may appear only in the specification part of a module. The default accessibility in a module is PUBLIC; it can be reaffirmed using a PUBLIC statement without a list. However, only one PUBLIC accessibility statement without a list is permit ted in a module.

The PUBLIC attribute in a type statement or in an accessibility statement permits access to entities such as module variables, type definitions, functions, and named constants. USE statements may control accessibility further.

A derived type may contain a PUBLIC attribute or an internal PUBLIC statement, if it is defined in a module.

The PUBLIC statement may also be used to permit access to sub routines, generic specifiers, and namelist groups.

The PUBLIC specification for a generic name, operator, or assignment does not apply to any specific name unless the specific name is the same as the generic name.

Examples

```
MODULE fourier
   PUBLIC      ! PUBLIC unless explicitly PRIVATE
   COMPLEX, PRIVATE :: fft   ! fft accessible only in module
   PRIVATE  a, b, c   ! accessible only in module
   PUBLIC   r, s, t    ! accessible outside the module
END MODULE fourier

MODULE place
   PRIVATE   ! Change default accessibility to PRIVATE
   INTERFACE OPERATOR (.st.)
      MODULE PROCEDURE xst
   END INTERFACE

   ! Make .st. public; everything else is private
   PUBLIC OPERATOR (.st.)
   LOGICAL, DIMENSION (100) :: lt
   CHARACTER(20) :: name
   INTEGER ix, iy
END MODULE PLACE
```

Related statements

PRIVATE and USE

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Modules” on page 161

READ

Inputs data from external and internal files.

Syntax

The syntax of the `READ` statement can take one of the following forms:

- Long form (for use when reading from a connected file):

  `READ` (*io-specifier-list*) [*input-list*]

- Short form (for use when reading from standard input):

  `READ` *format* [, *input-list*]

- Short namelist-directed form (for use when reading from standard input into a namelist group):

  `READ` *name*

io-specifier-list

is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*unit*

specifies the unit connected to the input file. *unit* can be one of the following:

- The name of a character variable, indicating an internal file
- An integer expression that evaluates to the unit connected to an external file
- An asterisk, indicating a pre-connection to unit 5 (standard input)

If the optional keyword `UNIT=` is omitted, *unit* must be the first item in *io-specifier-list*.

`[FMT=]`*format*

specifies the format specification for formatting the data. *format* can be one of the following:

- An asterisk (`*`), specifying list-directed I/O.
- The label of a `FORMAT` statement containing the format specification.

- An integer variable that has been assigned the label of a `FORMAT` statement.
- A character expression that provides the format specification.

If the optional keyword `FMT=` is omitted, *format* must be the second item in *io-specifier-list*.

NOTE The `NML=` and `FMT=` specifier may not both appear in the same *io-specifier-list*.

`[NML=]`*name*

specifies the name of a namelist group for namelist-directed input. *name* must have been defined in a `NAMELIST` statement. If the optional keyword `NML=` is omitted, *name* must be the second item in the list. The first item must be the unit specifier without the optional keyword `UNIT=`.

The `NML=` and `FMT=` specifier may not both appear in the same *io-specifier-list*.

`ADVANCE=`*character-expression*

specifies whether to use advancing I/O for this statement. *character-expression* can be one of the following arguments:

`'YES'`	Use advancing formatted sequential I/O (default).
`'NO'`	Use nonadvancing formatted sequential I/O.

If the `ADVANCE=` specifier appears in *io-specifier-list*, *unit* must be connected to an external file opened for formatted sequential I/O. Also, `ADVANCE='NO'` must be specified if the `EOR=` or `SIZE=` specifier appear in the list. Nonadvancing I/O is incompatible with list-directed and namelist I/O.

`END=`*stmt-label*

specifies the label of the executable statement to which control passes if an end-of-file record is encountered. This specifier is only valid for reading files opened for sequential access.

`EOR=`*stmt-label*

specifies the label of the executable statement to which control passes if an end-of-record condition is encountered. This specifier may appear in *io-specifier-list* only if `ADVANCE='NO'` also appears in the list.

`IOSTAT=`*integer-variable*

returns the I/O status after the statement executes. If the statement successfully executes, *integer-variable* is set to zero. If an end-of-file record is encountered without an error condition, it is set to a negative integer. If an error occurs, *integer-variable* is set to a positive integer that indicates which error occurred.

`REC=`*integer-expression*

specifies the number of the record to be read from a file connected for direct access. This specifier cannot appear in *io-specifier-list* with the `NML=`, `ADVANCE=`, `SIZE=`, and `EOR=` specifiers, nor with `FMT=*` (for list-directed I/O).

`SIZE=`*integer-variable*

returns the number of characters that have been read by this `READ` statement. This specifier may appear in *io-specifier-list* only if `ADVANCE='NO'` also appears in the list.

input-list

is a comma-separated list of data items for input. The data items can include variables and implied-`DO` lists.

format

is one of the following:

- An asterisk (`*`), specifying list-directed I/O.
- The label of a `FORMAT` statement containing the format specification.
- An integer variable that has been assigned the label of a `FORMAT` statement.
- An embedded format specification.

name

is the name of a namelist group, as previously defined by a `NAMELIST` statement. Using the namelist-directed syntax, the `READ` statement takes its input from standard input. To read from a connected file, you must use the `NML=` specifier with the full syntax form, as described below.

Description

The `READ` statement transfers data from an external or internal file to internal storage. An external file can be opened for sequential access or direct access. If it is opened for sequential access, the `READ` statement can perform the following types of I/O:

- Formatted
- Unformatted
- List-directed
- Namelist-directed

If the file is opened for direct access, the `READ` statement can perform formatted or unformatted I/O.

`READ` statements operating on internal files can perform formatted or list-directed I/O.

Examples

The following examples illustrate different uses of the READ statement.

Formatted sequential I/O

The following READ statement reads 10 formatted records from a file opened for sequential access, using an implied-DO list to read the data into the array x_array. If the end-of-file record is encountered before the array is filled, execution control passes to the statement at label 99.

```
READ (41, '(F10.2)', END=99) (x_array(i),i=1,10)
```

Nonadvancing I/O

The following READ statement takes its input from a file that was opened for sequential access and is connected to unit 9. It uses nonadvancing I/O to read an integer into the variable key. If the statement encounters the end-of-record condition before it can complete execution, control will pass to the executable statement at label 100. After the statement executes, the number of characters that have been read will be stored in cnt.

```
INTEGER :: key
READ (UNIT=9, '(I4)', ADVANCE='NO', SIZE=cnt, EOR=100) key
```

Internal file

The following statement inputs a string of characters from the internal file cfile, uses an embedded format specification to perform format conversion, and stores the results in the variables i and x:

```
READ (cfile, FMT='(I5, F10.5)') i, x
```

Namelist-directed I/O

Each of the four READ statements in the next example uses a different style of syntax to do exactly the same thing:

```
NAMELIST /nl/ a, b, c
READ (UNIT=5, NML=nl)  ! 5 = standard input
READ (5, nl)
READ (*, NML=nl)  ! * = standard input
READ nl  ! assume standard input
```

List-directed I/O

The following statement takes its data from standard input, storing the converted value in int_var. The format conversion is based on the type of int_var.

```
READ *, int_var
```

If you knew the format, you could substitute for the asterisk one of the following:

- The label of the FORMAT statement with the format specification, as in the following:

```
READ 100, int_var
100 FORMAT(I4)
```

- An embedded format specification, as in the following:

```
READ '(I4)', int_var
```

Unformatted direct-access I/O

The following statement takes its input from the file connected to unit 31. The REC= specifier indicates that the file has been opened for direct access and that this statement will read the record whose number is stored in the variable rec_num. If an I/O error occurs during the execution of the statement, an error number will be stored in ios, and execution control will branch to the executable statement at label 99.

```
READ (31, REC=rec_num, ERR=99, IOSTAT=ios) a, b
```

Related statements

CLOSE, OPEN, and WRITE.

Related concepts

For more about I/O concepts, including information about files and different types of I/O, see Chapter 8, “I/O and file handling,” on page 171. This chapter also lists example programs that use I/O. For information about I/O formatting, see Chapter 9, “I/O formatting,” on page 205.

REAL

Declares entities of type real.

Syntax

`REAL [`*kind-spec*`] [[, `*attrib-list*`] ::] `*entity-list*

kind-spec

is the kind type parameter that specifies the range and precision of the entities in *entity-list*. *kind-spec* takes the form:

`([KIND=]`*kind-param*`)`

where *kind-param* can be a named constant or a constant expression that has the integer value of 4, 8, or 16. The size of the default type is 4.

As an extension, *kind-spec* can take the form:

`*` *len-param*

where *len-param* is the integer 4, 8, or 16 (default = 4).

attrib-list

is a list of one or more of the following attributes, separated by commas:

ALLOCATABLE	INTRINSIC	PRIVATE
DIMENSION	OPTIONAL	PUBLIC
EXTERNAL	PARAMETER	SAVE
INTENT	POINTER	TARGET

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where *name* is the name of a variable or function, *array-spec* is a comma-separated list of dimension bounds, and *initialization-expr* is a real constant expression. If *initialization-expr* is present, *entity-list* must be preceded by the double colon.

Description

The REAL statement is used to declare the length and properties of data that approximate the mathematical real numbers. A kind parameter (if present) indicates the representation method.

The REAL statement is constrained by the rules for all type declaration statements, including the requirement that it precede all executable statements.

As a portability extension, HP Fortran 90 allows the following syntax for specifying the length of an entity:

name [**len*] [(*array-spec*)] [= *initialization-expr*]

If (*array-spec*) is specified, **len* may appear on either side of (*array-spec*). If *name* appears with **len*, it overrides the length specified by REAL**size*.

Examples

The following are valid declarations:

```
REAL, TARGET :: x, y
REAL(KIND=16) :: z
REAL(4), PARAMETER :: pi=3.14
! initialize an array, using an array constructor
REAL, DIMENSION(4) :: rvec=(/ 1.1,2.2,3.3,4.4 /)
! use the slash notation (an HP extension) to initialize
REAL x/2.87/, y/93.34/, z/13.99/  ! note, no double colon
! the following declarations are equivalent; the second uses the
! HP length specification extension
REAL (KIND = 8) x
REAL*4 x*8
```

Related statements

DOUBLE PRECISION

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Implicit typing” on page 28
- “Array declarations” on page 54
- “Array constructors” on page 71
- “Expressions” on page 80
- “KIND(X)” on page 537

RECORD (extension)

Declares a record of a previously defined structure.

Syntax

```
RECORD /struct-name/rec-name [, rec-name]...
        [/struct-name/rec-name [, rec-name ] ]...
```

struct-name

is the name of a structure declared in a previous structure definition.

rec-name

is a record name.

Description

HP Fortran 90 supports the `RECORD` statement as a compatibility extension. New programs should use the derived type, a standard feature of Fortran 90. For more information about derived types, see “Derived types” on page 39 and “TYPE (definition)” on page 457.

The `RECORD` statement declares a record variable of a structure that has been previously defined by a `STRUCTURE` statement. A record variable can consist of multiple data items, called *fields*. The `STRUCTURE` statement is described in “STRUCTURE (extension)” on page 437.

Referencing record fields

The syntax for referencing a field in a record depends on whether the field itself is another record (a composite reference) or not (a simple reference). Composite references have the following syntax:

rec-name [. *substruct-fieldname*]...

Simple references have the following syntax:

rec-name [. *substruct-fieldname*]... *simple-fieldname*

rec-name

is the name of the record in which a composite or simple field is being referenced.

substruct-field-name

is the name of a nested structure or nested record field name, if applicable.

simple-field-name

is the name of a lowest-level field, defined with a type declaration statement. As indicated by the syntax, the field could be part of a nested structure or nested record.

Given the following structure definition and record declarations:

```
STRUCTURE /abc/
  REAL  a, b, c(5)
  STRUCTURE /xyz/ xyz, xyzs(5)
    INTEGER     x, y, z(3)
  END STRUCTURE
END STRUCTURE

RECORD /abc/ abc, abcs(100)
RECORD /xyz/ xyz
```

the following are composite references:

```
abc     !composite record references
abcs(1)
xyz
abcs(idx)

abc.xyz !composite field references
abc.xyzs(3)
```

and the following are simple references:

```
abc.a
abc.c(1)
xyz.x
xyz.z(1)
abc.xyz.x
abcs(idx).xyz.y(1)
abcs(2).xyzs(3).z(1)
```

Composite references can be either to an entire record or to a record field that is itself a structure or record.

Rules for record field

Arrays of records can be created as follows:

```
RECORD /student/ students(1000)
```

or

```
RECORD /student/ students
DIMENSION students (1000)
```

In either case a 1000-record array called `students` of structure `student` is declared.

Records can be placed in common blocks. The following code places the `students` array (declared above) in the common block `frosh`, along with variables `a`, `b`, and `c`:

```
COMMON /frosh/ a, b, c, students
```

Simple field references can appear wherever a variable can appear. The following assigns values to the fields of record `r` of structure `struct`:

```
STRUCTURE /struct/
  INTEGER       i
  REAL          a
END STRUCTURE

RECORD /struct/ r
r.i = r.i + 1
r.a = FLOAT(r.i) - 2.7
```

Composite assignment is allowed for two records or two composite fields of the same structure—that is, the record declaration statements for both records must have specified the same *struct-name*. For example, the following is legal:

```
STRUCTURE /string/
  BYTE len
  CHARACTER*1 str(254)
END STRUCTURE
RECORD /string/ str1, str2
str1 = str2
```

The following example is also valid and uses composite assignment to assign the value of the record `edate` of structure `date` to a field of the same structure (`when`) in the record `event`:

```
STRUCTURE /event/
  CHARACTER*20  desc
  STRUCTURE /date/ when
    BYTE  month, day
    INTEGER*2 year
  END STRUCTURE
END STRUCTURE

RECORD /date/ edate
RECORD /event/ event
edate.month = 1
edate.day   = 6edate.year  = 62
event.desc = 'Party for Joanne'
! composite assignment of record to field
!   of record--both have same structure
event.when = edate
```

Even though the following records are of identical structures—that is, the fields of both structures have the same type, size, and format—the code is invalid because the structures have a different name:

```
STRUCTURE /intarray/
  BYTE          elem_count
  INTEGER       arr(100)
END STRUCTURE

STRUCTURE /iarray/
  BYTE          elem_count
  INTEGER       arr(100)
END STRUCTURE

RECORD /intarray/ iarray1
RECORD /iarray/ iarray2
! The next assignment won't work.  The two
!   records are not of the same structure.
iarray1 = iarray2     ! Invalid
```

When performing I/O on structures and records, composite record and field references can appear only in unformatted I/O statements. They are not allowed in formatted, list-directed, or namelist-directed I/O statements. However, simple field references can appear in all types of I/O statements. For information about I/O, see Chapter 9, “I/O formatting,” on page 205.

A record name or composite field reference can appear as either a formal or an actual argument to a subroutine or function. Formal and actual arguments must have the same size as well as the same number, type, and order of fields.

Composite record and field arguments to subroutines and functions are passed by reference, just like other HP Fortran 90 arguments.

Adjustable arrays are allowed in `RECORD` statements that declare formal arguments.

Do not name a field with any of the following:

- Logical constants, `.TRUE.` and `.FALSE.`
- Logical operators, such as `.OR.`, `.AND.`, and `.NOT.`
- Relational operators, such as `.EQ.`, `.LT.`, and `.NEQV.`
- The name of a defined operator

Related statements

`STRUCTURE` and `TYPE`

Related concepts

For related information, see the following:

- “Derived types” on page 39
- “Allocatable arrays” on page 59
- “Arguments” on page 139
- “Procedures” on page 123

RETURN

Returns control from a subprogram.

Syntax

`RETURN [`*scalar-integer-expression*`]`

scalar-integer-expression

is an optional scalar integer expression that is evaluated when the `RETURN` statement is executed. It determines which alternate return is used.

Description

A `RETURN` statement can appear only in a subprogram.

An expression may appear in a `RETURN` statement only if alternate returns (one or more asterisks) are specified as dummy arguments in the relevant `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement of the subprogram. An expression with a value *i* in the range will return to the *i*th asterisk argument (specified as `*`*label*) in the actual argument list. A normal return is executed if *i* is not in the range 1 to *n*, where *n* is the number of dummy argument alternate returns specified.

Examples

```
SUBROUTINE calc (y, z)
! Subroutine calc checks the range of y.  If
! it exceeds the permitted range, it calls
! an error handler and stops the program
  IF (y > ymax) GO TO 303
  RETURN
! It returns to the caller of calc if the
! calculation proceeds to normal completion.
303 CALL err (3, "OUT OF RANGE")
  STOP 303
END
```

Related statements

`SUBROUTINE` and `FUNCTION`

Related concepts

For more information about returning from a procedure call, see “Returning from a procedure reference” on page 132.

REWIND

Positions file at its initial point.

Syntax

The syntax of the `REWIND` statement can take one of the following forms:

- Short form:

  integer-expression

- Long form:

  `REWIND (`*io-specifier-list*`)`

integer-expression

is the unit connected to a sequential file or device.

io-specifier-list

is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*unit*

specifies the unit connected to an external file opened for sequential access. *unit* must be an integer expression that evaluates to a number greater than 0. If the optional keyword `UNIT=` is omitted, *unit* must be the first item in *io-specifier-list*.

`ERR=`*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

`IOSTAT=`*integer-variable*

returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

Description

The `REWIND` statement repositions the file connected to the specified unit at the start of the first record. If the file is already at its starting point or if the unit is not connected to a file, the `REWIND` statement has no effect.

Examples

The following example of the REWIND statement repositions the file connected to unit 10 to its initial point:

```
REWIND 10
```

The next example repositions to its initial point the file connected to unit 21. If an error occurs during the execution of the statement, control passes to the statement at label 99, and the error code is returned in `ios`:

```
REWIND (21, ERR=99, IOSTAT=ios)
```

Related statements

BACKSPACE, ENDFILE, and OPEN

Related concepts

For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171. This chapter also lists example programs that use I/O.

SAVE (statement and attribute)

Stores variables in static memory.

Syntax

A type declaration statement with the SAVE attribute is:

type , *attrib-list* :: *save-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.).

attrib-list

A comma-separated list of attributes including SAVE and optionally those attributes compatible with it, namely:

ALLOCATABLE	PRIVATE	TARGET
DIMENSION	PUBLIC	VOLATILE
POINTER	STATIC	

save-list

is a comma-separated list of names of objects to save.

The syntax of the SAVE statement is:

SAVE [[::] *save-list*]

Description

The SAVE statement and attribute cause objects in a subroutine or function to be stored in static memory, instead of being dynamically allocated whenever the procedure is invoked (the default case). A saved object retains its value and definition, association, and allocation status between invocations of the program unit in which the saved object is declared.

If *save-list* is omitted, everything in the scoping unit that can be saved is saved. No other explicit occurrences of the SAVE attribute or the SAVE statement are allowed.

The names of the following may appear in *save-list*:

- Scalar variables
- Arrays
- Named common blocks
- Derived type objects
- Records

If the name of a common block appears in *save-list*, it must be delimited by slashes (for example, `/my_block/`); all variables in the named common block are saved. If a common block is saved in one program unit, it must be saved in all program units (except main) where it appears.

HP Fortran 90 always saves all common blocks.

The following must not appear in *save-list*:

- Formal argument names
- Procedure names
- Selected items in a common block
- Variables declared with the `AUTOMATIC` statement or attribute
- Function results
- Automatic data objects (such as automatic arrays, allocatable arrays, automatic character strings, and Fortran 90 pointers)

Initializing a variable in a DATA statement or in a type declaration statement implies that the variable has the SAVE attribute, unless the variable is in a named common block in a block data subprogram.

A SAVE statement in a main program unit has no effect.

Examples

The SAVE statement in the following example saves the variables `a`, `b`, and `c`, as well as the variables in the common block `dot`:

```
SUBROUTINE matrix
SAVE a, b, c, /dot/
RETURN
```

The SAVE statement in the next example saves the values of all of the variables in the subroutine `fixit`:

```
SUBROUTINE fixit
SAVE
RETURN
```

Related statements

`AUTOMATIC` and `STATIC`

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- “Recursive reference” on page 131
- Information about automatic and static variables, in the *HP Fortran 90 Programmer's Guide*

SELECT CASE

Begins CASE construct.

Syntax

`[`*construct-name* `:] SELECT CASE (`*case-expr*`)`

construct-name

is the name given to the CASE construct.

case-expr

is a scalar expression of type integer, character, or logical.

Description

The SELECT CASE statement, the first statement of a CASE construct, causes *case-expr* to be evaluated, resulting in the case index. The CASE construct uses the case index to determine which of its statement blocks to execute.

If *construct-name* is specified, it must also appear in the END SELECT statement.

Examples

For an example of the SELECT CASE statement, see “CASE” on page 265.

Related statements

CASE and END (construct)

Related concepts

For information about the CASE construct, see “CASE construct” on page 105.

SEQUENCE

Imposes storage sequence on components of derived type object.

Syntax

```
SEQUENCE
```

Description

The `SEQUENCE` statement can appear once within any derived type definition; its presence specifies that a **storage sequence** on the components that is the same as their definition order. A derived type that includes the `SEQUENCE` statement in its definition is known as a **sequence derived type**. Sequence derived types are used:

- To allow objects of sequence derived type to be storage associated with the `COMMON` and `EQUIVALENCE` statements.
- To allow actual and dummy arguments to have the same type without use or host association. The corresponding actual and dummy arguments of derived types are of the same derived type if the derived-type objects refer to the same type definition. Alternatively, they are of the same type if all of the following are true:
  - They refer to different type definitions with the same name.
  - They have the `SEQUENCE` statement in their definitions.
  - The components have the same names and types and are in the same order.
  - None of the components is of a private type or of a type that has private access.

The following restrictions apply to the use of the `SEQUENCE` statement:

- No more than one `SEQUENCE` statement may appear in the definition of a derived type.
- If a derived type definition includes the `SEQUENCE` statement, each component that is of derived type must also include the `SEQUENCE` statement.

Examples

```
TYPE weather
! weather is a sequence derived type with two
!  character components and two integer components
   SEQUENCE
   CHARACTER(LEN=32) place
   INTEGER high_temp, low_temp
   CHARACTER(LEN=16) conditions
END TYPE weather
```

Related statements

`TYPE`, `COMMON`, and `EQUIVALENCE`

Related concepts

For information about sequence derived types, see “Sequence derived type” on page 41.

STATIC (statement, attribute, extension)

Gives variables and arrays static storage.

Syntax

The syntax of a type declaration statement with the STATIC attribute is:

type, *attribute-list* :: *entity-list*

type

is a valid type specification (INTEGER, REAL, LOGICAL, CHARACTER, TYPE (*name*), etc.), as described in Chapter 3, “Data types and data objects,” on page 21.

attribute-list

is a comma-separated list of attributes including STATIC and optionally those attributes compatible with it, namely:

ALLOCATABLE	PRIVATE	VOLATILE
DIMENSION	SAVE	
POINTER	TARGET	

entity-list

is a comma-separated list of variables and arrays.

The syntax of the STATIC statement is:

STATIC [::] *entity-list*

Description

The STATIC statement and attribute is an HP Fortran 90 extension. Variables possessing the STATIC attribute retain their storage location for the duration of the program. A STATIC variable declared within a procedure will therefore retain its value between calls of the procedure.

The STATIC statement and attribute has the same functionality as the SAVE statement and attribute; it is provided for compatibility with other vendors’ Fortran 90.

Examples

```
SUBROUTINE work_out(first_call)
  LOGICAL first_call
  INTEGER, STATIC :: ncalls

  IF (first_call) ncalls = 0
  ncalls = ncalls + 1    ! record how often work_out is called
  ...
END SUBROUTINE work_out
```

Related statements

AUTOMATIC and SAVE

Related concepts

For related information, see the following:

- “Type declaration for intrinsic types” on page 24
- Information about automatic and static variables, in the *HP Fortran 90 Programmer's Guide*

STOP

Terminates program execution.

Syntax

STOP [*stop-code*]

stop-code

is a character constant, a named constant, or a list of up to 5 digits.

Description

The STOP statement terminates program execution and optionally prints a message to standard error or standard list.

STOP also sends a message to standard error, dependent on whether digits, characters, or nothing was specified with the STOP statement:

- If digits are specified, the message “STOP *digits*” is written to standard error.
- If a character expression is specified, the message “STOP *character-expression*” is written.
- If nothing appears after STOP, nothing is written.

Examples

```
IF (b .LT. c) STOP 'BAD VALUE!'
```

Related statements

PAUSE

Related concepts

For information about flow control statements, see “Flow control statements” on page 112.

STRUCTURE (extension)

Defines a named structure.

Syntax

```
STRUCTURE /struct-name/
   field-def
   ...
END STRUCTURE
```

struct-name

is the structure's name, delimited by slashes. *struct-name* can be used later to declare a record.

field-def

is a field definition.

Description

HP Fortran 90 supports the STRUCTURE statement as a compatibility extension. New programs should use the derived type, a standard feature of Fortran 90; derived types provide the same functionality as named structures. For more information about derived types, see "Derived types" on page 39 and "TYPE (declaration)" on page 454.

The STRUCTURE statement defines the type, size, and layout of a structure's fields, and assigns a name to the structure. Once a structure is defined, you can declare records of that structure using the RECORD statement and can manipulate the record's fields.

A structure definition pertains only to the program unit in which it is defined. For example, you cannot define a structure in the main program unit and then declare a record of that structure in a subprogram unit. Instead, the structure must be explicitly defined again in the subprogram unit.

field-def can be any of the following:

- A type declaration statement
- A nested structure definition
- A nested record declaration
- A union definition

Each type of field definition is described in the remaining sections.

Field definition as type declaration

At the simplest level, *field-def* can be a type declaration statement. As such, *field-def* has the same syntax as a standard Fortran 90 type declaration statement, except that the only attribute that can be specified is the DIMENSION attribute. A variable defined with a type declaration statement is called a *field*.

The following code uses simple type declaration statements to define a structure named `date` with three fields: `month` and `day` of type `BYTE`, and `year` of type `INTEGER(KIND=2)`:

```
STRUCTURE /date/
  BYTE :: month, day
  INTEGER(KIND=2) :: year
END STRUCTURE
```

A type declaration statement in a structure definition can optionally define initial values for the fields. For example:

```
STRUCTURE /xyz/
  REAL :: x = 1.0, y = 2.0, z = 3.0
END STRUCTURE
```

Thereafter, any record declared of structure `xyz` will have its `x`, `y`, and `z` fields initially set to 1.0, 2.0, and 3.0 respectively. Consider the following:

```
RECORD /xyz/ xyz
PRINT *, xyz.x, xyz.y, xyz.z
```

Even though no values have been assigned to the fields of `xyz` with an assignment statement, the above code will display:

```
 1.0 2.0 3.0
```

Implicit typing is not allowed in a structure definition. For example, the following code would cause a compile error:

```
STRUCTURE /dimensions/
  x, y, z    ! illegal
END STRUCTURE
```

A correct way to code this would be:

```
STRUCTURE /dimensions/
  REAL(KIND=8) :: x, y, z    ! legal
END STRUCTURE
```

A field type declaration statement can also define an array, as in the following:

```
STRUCTURE /foo_bar/
  INTEGER foo(10)
END STRUCTURE
```

or, using Fortran 90 syntax:

```
STRUCTURE /foo_bar/
  REAL, DIMENSION(30, 50) :: bar
END STRUCTURE
```

The array's dimensions must in any case appear in the type statement. The `DIMENSION` statement (but not the `DIMENSION` attribute) is illegal in a structure definition. The following code defines the structure, `string`, which uses a type declaration statement to define an array field `str` of type `CHARACTER(LEN=1)`, containing 254 elements:

```
STRUCTURE /string/
  CHARACTER(LEN=1) :: str(254)! Contains string
  INTEGER :: length          ! string's length
END STRUCTURE
```

As mentioned, the `DIMENSION` statement cannot be used in a structure definition. For example, the following code would cause a compile-time error:

```
STRUCTURE /real_array/
  REAL :: rarray
  DIMENSION arr(100)       ! illegal example
END STRUCTURE
```

A correct way to code this would be:

```
STRUCTURE /real_array/
  REAL :: rarray(100)
END STRUCTURE
```

or

```
STRUCTURE /real_array/
  REAL, DIMENSION(100) :: arr
END STRUCTURE
```

Assumed-size and adjustable arrays are also illegal in structure definitions. For example, the following is illegal:

```
STRUCTURE /assumed_size/  ! illegal example
  CHARACTER*(*) :: carray
END STRUCTURE
```

The following is also illegal:

```
STRUCTURE /adj_array/   ! illegal example
  INTEGER :: size
  REAL :: iarray(size)
END STRUCTURE
```

For alignment purposes, HP Fortran 90 provides the `%FILL` field name. It enables the programmer to pad a record to ensure proper alignment. The padding does not have a name and is therefore not accessible. For example, the following structure, `sixbytes`, creates a 6-byte structure, of which 4 bytes are inaccessible filler bytes:

```
STRUCTURE /sixbytes/
  INTEGER(KIND=2) :: twobytes
  CHARACTER(LEN=4) :: %FILL
END STRUCTURE
```

`%FILL` can be of any type and may appear more than once in a structure.

`%FILL` should not be needed in normal usage. The compiler automatically adds padding to ensure proper alignment.

Nested structures

A *field-def* can itself be a structure definition, known as a nested structure. The syntax of a nested structure definition is:

```
STRUCTURE /struct-name/struct-field-list
  field-def
  ...
END STRUCTURE
```

struct-name

is the structure's name (delimited by slashes), which can be used later to declare a record.

struct-field-list

is a comma-separated list of one or more names of nested structure field names.

field-def

can be one of the following regular field definitions (defined in the same way as an unnested structure field):

- A type declaration statement
- Another nested structure
- A nested record
- A union definition

NOTE Note that a structure definition allows multiple levels of nesting.

A nested structure definition is the same as an unnested structure definition, with two exceptions:

- /*struct-name*/ is optional in a nested structure.
- A nested structure definition must include a list of one or more structure field names (*struct-field-list*).

If /*struct-name*/ is present in a nested structure definition, the structure *struct-name* can also be used in subsequent record declarations. For example, the following code defines a structure named `person`, which contains a nested structure named `name`. The structure's field name is `nm` and contains three `CHARACTER*10` fields: `last`, `first`, and `mid`.

```
STRUCTURE /person/
  INTEGER :: person_id
  ! Define the nested structure 'name' with the field name 'nm'.
  STRUCTURE /name/ nm
    CHARACTER(LEN=10) :: last, first, mid
  END STRUCTURE
END STRUCTURE
```

Given this definition, the following code defines the record `p` of structure `person` and the record `n` of structure `name`:

```
RECORD /person/p
RECORD /name/n
```

If /*struct-name*/ is not present, then the structure can be used only in this declaration. For example, we could redefine the `person` structure so that the nested structure no longer has a name:

```
STRUCTURE /person/
  INTEGER :: person_id
  STRUCTURE nm
    CHARACTER(LEN=10) :: last, first, mid
  END STRUCTURE
END STRUCTURE
```

There is no way to declare a separate record of the nested structure because it has no name. Note, however, that the nested structure still has a field name, `nm`. The field name is required.

To declare an array of nested structures, simply specify a dimension declarator with the structure's field name. For example, the following structure definition contains a nested, 3-element array of structures with field name `phones` of structure `phone`:

```
STRUCTURE /person/
  INTEGER :: person_id
  ! Define the nested structure 'name' with the field name 'nm'.
  STRUCTURE /name/ nm
    CHARACTER(LEN=10) :: last, first, mid
  END STRUCTURE

! Nested array of structures.
  STRUCTURE /phone/ phones(3)
    INTEGER(KIND=2) :: area_code
    INTEGER :: number
  END STRUCTURE
END STRUCTURE
```

Nested records

A *field-def* can be a record declaration, known as a nested record. See "RECORD (extension)" on page 420 for information about record declarations.) A nested record declaration must use a structure that has already been defined. The following code first defines the structure `date`. It then declares the structure `event`, which contains the nested record `when` of structure `date`:

```
STRUCTURE /date/
  BYTE :: month, day
  INTEGER :: year
END STRUCTURE
STRUCTURE /event/
  CHARACTER :: what, where
  RECORD /date/ when
END STRUCTURE
```

A structure definition can also declare an array of nested records. For example, the following code defines the structure `calendar`, which contains a 100-element array of records of structure `event`:

```
STRUCTURE /calendar/
  ! number of events
  INTEGER(KIND=2) :: event_count
  RECORD /event/  events(100)  ! array of event records
END STRUCTURE
```

Unions

A *field-def* can be a union—a form of nested structure in which two or more map blocks share memory space. The `UNION` and `MAP` statements together define a union. The syntax of a union definition is:

```
UNION
    map-block
    map-block
    ...
END UNION
```

where map-block is defined by a `MAP` statement and one or more field definitions. All map blocks within the enclosing UNION statement share the same memory space in a record. The syntax for defining a map block is:

```
MAP
    field-def
    ...
END MAP
```

where *field-def* can be one of the following:

- A type declaration statement
- Another nested structure
- A nested record
- A union definition

Note that a structure definition allows multiple levels of nesting.

For programmers who are familiar with C or Pascal, HP Fortran 90 unions are similar to unions in C and variant records in Pascal. HP Fortran 90 unions differ from C unions in that they must be defined inside a structure definition.

The structure below contains a union with two map blocks. The first contains the integer field `int`; the second contains the real field `float`.

```
STRUCTURE /var/
  INTEGER :: type    ! 1=INTEGER, 2=REAL
  UNION
    MAP
      INTEGER :: int
    END MAP
    MAP
      REAL :: float
    END MAP
  END UNION
END STRUCTURE
```

To declare a record of this structure named `v`, use the following `RECORD` statement:

```
RECORD /var/ v
```

The declaration of the record `v` reserves 8 bytes of storage: 4 bytes for the `type` field and 4 bytes to be shared by `int` and `float`. If you use the `int` field to access the 4 bytes, they will be interpreted as an integer; if you use the `float` field, they will be interpreted as a real.

It is the programmer's responsibility to ensure that appropriate values are assigned to each field in a union. For instance, given the previous declaration of `v`, the following assignments make sense:

```
v.type =1  ! set the type to integer
! access the storage shared by 'int' and 'float' as an integer
v.int = 3
```

In contrast, the following code would yield unexpected results, although it would compile without errors:

```
v.type = 1  ! set the type to integer
! the next statement contradicts the previous statement
v.float = 3.14
```

Once a value is assigned to a map block, all other map blocks become undefined. The reason is that all map blocks share memory space within a union; therefore, the values of one map block may become altered if you assign a value to a field in another map block. Consider the following definition of a structure called `struct` and the declaration of a record called `rec`:

```
STRUCTURE /struct/
  UNION
    MAP
      CHARACTER*8 :: s
    END MAP
    MAP
      CHARACTER*1 :: c(8)
    END MAP
  END UNION
END STRUCTURE

RECORD /struct/ rec
```

If we made the following assignment to the s field:

```
rec.s = 'ABCDEFGH'
```

and then executed the next two PRINT statements:

```
PRINT *, rec.s
PRINT *, rec.c
```

the output would be:

```
ABCDEFGH
ABCDEFGH
```

Now, if we set values in the `c` field and display both fields again

```
rec.c(1) = '1'
rec.c(8) = '8'
PRINT *, rec.s
PRINT *, rec.c
```

the output would be:

```
1BCDEFG8
1BCDEFG8
```

Note how the `s` field has changed, even though it was not directly assigned any new values. This is a result of the `s` and `c` field sharing the same storage space in the union. Although this is valid coding—that is, it will not cause a compiler or runtime error—it may cause unexpected results.

However, you can also use shared memory mapping to your benefit. The fact that map blocks share space within a union makes unions useful for equivalencing data within a record. For example, the following structure could be used to mask off individual bytes in a 4-byte word:

```
STRUCTURE /wordmask/
  UNION
    MAP
      INTEGER(KIND=4) :: word
    END MAP
    MAP
      BYTE :: byte0, byte1, byte2, byte3
    END MAP
  END UNION
END STRUCTURE RECORD /wordmask/ maskrec
```

If we assign a value to the `word` field of `maskrec`, we can then get the individual values of all four bytes in `maskrec` by looking at the fields `byte0`, `byte1`, `byte2`, and `byte3`. To see how the integer variable `word` maps onto the byte variables `byte0`, `byte1`, `byte2`, and `byte3`, use the following statements:

```
    maskrec.word = 32767
    WRITE(*, fmt=100) 'word = ', maskrec.word
    WRITE(*, 200) 'byte 0 = ', maskrec.byte0
    WRITE(*, 200) 'byte 1 = ', maskrec.byte1
    WRITE(*, 200) 'byte 2 = ', maskrec.byte2
    WRITE(*, 200) “byte 3 = ', maskrec.byte3
100 FORMAT(A, Z8.8)
200 FORMAT(A, Z2.2)
```

This code displays the following output:

```
word = 00007FFF
byte 0 = 00
byte 1 = 00
byte 2 = 7F
byte 3 = FF
```

Such code, depending as it does on a specific word size, is inherently nonportable.

Related statements `RECORD` and `TYPE`

Related concepts Derived types are described in “Derived types” on page 39.

SUBROUTINE

Begins the definition of a subroutine subprogram.

Syntax

`[RECURSIVE] SUBROUTINE` *subr-name* `[([`*dummy-arg-list*`])]`

subr-name

is the name of a subroutine.

dummy-arg-list

is a comma-separated list of zero or more of *dummy-arg-name* or the asterisk character (`*`).

As indicated by the syntax, the parentheses surrounding the dummy arguments may be omitted if there are no dummy arguments.

Description

The `SUBROUTINE` statement is the first statement of a subroutine subprogram.

The following rules and restrictions apply to subroutines:

- A subroutine is either an external, module, or internal subprogram.
- If a subroutine calls itself directly or indirectly, the word `RECURSIVE` must appear in the `SUBROUTINE` statement. If the keyword `RECURSIVE` is specified, the subroutine interface is explicit within the subprogram.
- The keyword `SUBROUTINE` must appear on the `END` statement if the subroutine is a module or internal procedure.
- An asterisk in a subroutine dummy argument list designates an alternate return.
- The interface of an internal subroutine is explicit in its host. The interface of a module subroutine is explicit within the module, and if it is public, it is explicit in all program units using the module. The interface of an external subroutine is implicit, but may be made explicit by the use of an interface block.

Examples

Consider the following subroutines:

```
! A subroutine definition with two arguments.
SUBROUTINE exchange (x, y)
   temp = x; x = y; y = temp
END SUBROUTINE exchange

SUBROUTINE altitude (*, long, lat)
   ! asterisk (*) indicates alternate return
   IMPLICIT NONE
   INTEGER, OPTIONAL :: long, lat
   RETURN 1
END SUBROUTINE altitude
```

The preceding subroutines may be referenced with the CALL statement, as in the following program:

```
PROGRAM reject
   CALL exchange (a,t)        ! A subroutine reference
   ! subroutine reference, including an alternate return label,
   !   missing optional argument, and an argument keyword
   CALL altitude (*90, lat = 49)
END PROGRAM reject
```

Following are some other examples of subroutine statements:

```
SUBROUTINE pressure_surface  ! No arguments
SUBROUTINE taffy ()          ! Also no arguments
RECURSIVE SUBROUTINE fact (n, x)
```

Related statements

CALL, END, ENTRY, FUNCTION, and RETURN

Related concepts

For related information, see the following:

- “External procedures” on page 128
- “Arguments” on page 139

TARGET (statement and attribute)

Allows variables and arrays to be pointer targets.

Syntax

The syntax of a type declaration statement with the `TARGET` attribute is:

type, *attrib-list* :: *entity-list*

type

is a valid type specification (`INTEGER`, `REAL`, `LOGICAL`, `CHARACTER`, `TYPE (`*name*`)`, etc.).

attrib-list

is a comma-separated list of attributes including `TARGET` and optionally those attributes compatible with it, namely:

ALLOCATABLE	OPTIONAL	SAVE
DIMENSION	PRIVATE	
INTENT	PUBLIC	

entity-list

is a comma-separated list of entities. Each entity is of the form:

array-name [(*deferred-shape-spec-list*)]

If (*deferred-shape-spec-list*) is omitted, it must be specified in another declaration statement.

array-name

is the name of an array being given the attribute `ALLOCATABLE`.

deferred-shape-spec-list

is a comma-separated list of colons, each colon representing one dimension. Thus the rank of the array is equal to the number of colons specified.

The syntax of the TARGET statement is:

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

array-spec

explicit-shape-spec is	[*lower-bound* :] *upper-bound*
assumed-shape-spec is	[*lower-bound*] :
deferred-shape-spec is	:
assumed-size-spec is	[*explicit-shape-spec-list* ,] [*lower-bound* :] *

That is, an *assumed-size-spec* is an *explicit-shape-spec-list* with the final upper bound given as *.

Description

The TARGET attribute or statement specifies that *name* is a target that may be pointed at by a pointer. A target may be either a scalar or an array.

The TARGET attribute allows the compiler to generate efficient code because only those objects specified with the TARGET or POINTER attribute can be dynamically aliased.

If the target in a pointer assignment is a variable, then one of the following must be true:

- It must have the TARGET attribute.
- It must be the component of a derived-type, the element of an array variable, or the substring of a character variable that has the TARGET attribute.
- It must have the POINTER attribute.

If the target of a pointer assignment is an array section, the array must have either the TARGET or the POINTER attribute.

Examples

```
! p is a pointer array
INTEGER, POINTER, DIMENSION(:,:) :: p
! declare t as an array with the TARGET attribute
INTEGER, TARGET :: t(10, 20, 30)
! make p point to a rank-2 section of t
```

```
p => t(10,1:10,2:5)

REAL, POINTER :: nootka(:), talk(:)
REAL, ALLOCATABLE, TARGET :: x(:)
ALLOCATE (x(1:100), STAT = is)
nootka => x(51:100)
! Pointer assignment statements
talk => x(1:50)

REAL r, p1, p2
TARGET r
POINTER p1, p2
r = 4.7
! make both p1 and p2 aliases of r
p1 => r
p2 => p1
...
ALLOCATE (p1)
p1 = 9.4
```

Related statements

POINTER, ALLOCATE, DEALLOCATE, and NULLIFY

Related concepts

For related information, see the following:

- “Pointers” on page 47
- “Pointer assignment” on page 97
- The description of the ASSOCIATED intrinsic in Chapter 11, “Intrinsic procedures,” on page 475.

TASK COMMON (extension)

Declares a common block to be local to a thread during parallel execution.

NOTE A program that uses the `TASK COMMON` statement should be compiled with the `+Oparallel` or `+parallel` option; otherwise, the compiler treats the `TASK COMMON` statement as a `COMMON` statement.

Syntax

```
TASK COMMON /cbn/nlist[,/cbn/nlist...]
```

cbn

is a symbolic name for a common block that is declared in a TASK COMMON statement. Unnamed common blocks are not allowed in a TASK COMMON statement.

nlist

is a list of variable names, array names, and array declarators. These variables cannot appear in a DATA statement, but otherwise can be used like other variables in common storage.

Description

The `TASK COMMON` statement is an extension to the Fortran 90 standard and is provided for compatibility with programs that use the Cray `TASK COMMON` feature. `TASK COMMON` blocks can only be declared in functions and subroutines.

A program should already be running multiple threads before calling a subroutine that contains a `TASK COMMON` block.

When used in a program executing multiple threads, the `TASK COMMON` statement declares all variables in a common block as local to a thread (also called a *task*). If multiple threads execute code that uses the same `TASK COMMON` block, each thread has a private copy of the block.

All occurrences of the `TASK COMMON` block must be declared with the `TASK COMMON` statement; a common block cannot be declared in both a `COMMON` statement and a `TASK COMMON` statement.

Related statements

`COMMON`

Related concepts

For related information, see the following:

- "Type declaration for intrinsic types" on page 24
- "Implicit typing" on page 28
- "Array declarations" on page 54
- "Array constructors" on page 71
- "Expressions" on page 80

TYPE (declaration)

Declares a variable of derived type.

Syntax

`TYPE (`*type-name*`) [[,` *attrib-list*`] ::]` *entity-list*

type-name

is the name of a previously defined derived type.

attrib-list

is a comma-separated list of one or more of the following attributes:

`ALLOCATABLE`	`INTRINSIC`	`PRIVATE`
`DIMENSION`	`OPTIONAL`	`PUBLIC`
`EXTERNAL`	`PARAMETER`	`SAVE`
`INTENT`	`POINTER`	`TARGET`

If *attrib-list* is present, it must be followed by the double colon. For information about individual attributes, see the corresponding statement in this chapter.

entity-list

is a list of entities, separated by commas. Each entity takes the form:

name [(*array-spec*)] [= *initialization-expr*]

where:

name

is the name of a variable or function

array-spec

is a comma-separated list of dimension bounds *initialization-expr*

is a structure constructor

initialization-expr

is present

entity-list

must be preceded by the double colon.

Description

The TYPE declaration statement specifies the type and attributes of derived-type objects. A derived-type object may be an array, which may be deferred shape (pointer or allocatable), assumed shape (dummy argument), or assumed size (dummy argument).

Assignment is intrinsically defined for each derived type but may be redefined by the user. Operators appropriate to a derived type may be defined by procedures with the appropriate interfaces.

When a derived-type object is used as a procedure argument, the types of the associated actual and dummy arguments must be the same. For sequence derived types different physical type definitions may be used for the actual and dummy arguments, as long as both type definitions specify identical type names, components, and component order. For nonsequenced types the same physical type definition must be used, typically accessed via host or use association, for both the actual and dummy arguments.

Examples

```
! Weather is a simple derived type with two
!   character components and two integer components.
TYPE Weather
   CHARACTER(LEN=32) Place
   INTEGER High_temp, Low_temp
   CHARACTER(LEN=16) Conditions
END TYPE Weather

TYPE (Weather) July(num_ws, 31)
! A two-dimensional Weather array for July
July(:,:) % Low_temp = -40
! Initialize all low temps in July
TYPE Polar
! Polar is a derived type with two real components that cannot be
! directly accessed in Polar objects outside the module
   PRIVATE
   REAL rho, theta
END TYPE Polar

! Point is a derived type with three components, one of which is
! itself of derived type
```

```
TYPE Point
   REAL x, y
   TYPE (Polar) p
END TYPE Point

TYPE (Polar) r, q(500)
! Two variables of type Polar
TYPE (Point) a, b, t(100,100)
! Three variables of type Point
b = Point(0.,0.,Polar(0.,0.))
! Use of nested structure constructors.
```

Related statements `INTERFACE`, `PRIVATE`, `PUBLIC`, `SEQUENCE`, and `TYPE` (definition)

Related concepts For information about derived types, see “Derived types” on page 39.

TYPE (definition)

The first statement of a derived type definition.

Syntax

TYPE [[, *access-spec*] ::] *derived-type-name*

access-spec
: is the keyword PUBLIC or PRIVATE.

derived-type-name
: is a legal Fortran 90 name.

Description

The TYPE statement introduces the definition of a derived type. A derived type name may be any legal Fortran 90 name, as long as it is not the same as an intrinsic type name or another local name (except component names and actual argument keyword names) in that scoping unit.

A derived type may contain an access specification (PUBLIC or PRIVATE attribute) or an internal PRIVATE statement only if it is in a module.

Examples

```
! This is a simple example of a derived type
!    with two components, high and low.
TYPE temp_range
   INTEGER high, low
END TYPE temp_range

! This type uses the previous definition for one of its
! components
TYPE temp_record
   CHARACTER(LEN=40) city
   TYPE (temp_range) extremes(1950:2050)
END TYPE temp_record
! This type has a pointer component to provide links to other
! objects of the same type, thus providing linked lists.
TYPE linked_list
   REAL value
   TYPE(linked_list),POINTER :: next
END TYPE linked_list
! This is a public type whose components are private; defined
! operations provide all functionality.
TYPE, PUBLIC :: set; PRIVATE
   INTEGER cardinality
   INTEGER element ( max_set_size )
END TYPE set
! Declare scalar and array of type set.
TYPE (set) :: baker, fox(1:size(hh))
```

Related statements `INTERFACE`, `PRIVATE`, `PUBLIC`, `SEQUENCE`, and `TYPE` (declaration)

Related concepts For information about derived types, see "Derived types" on page 39.

TYPE (I/O) (extension)

Writes to standard output.

Description

The `TYPE` statement is a synonym for the `PRINT` statement and has the same functionality and syntax. It is provided as an HP extension for compatibility with earlier versions of Fortran. For more information, see "PRINT" on page 402.

UNION (extension)

Defines a union within a structure.

Syntax

```
UNION
   map-block
   map-block
   ...
END UNION
```

map-block

is one or more of the following:

- A type declaration statement
- Another nested structure
- A nested record
- A union definition

Description

The UNION statement is an HP Fortran 90 extension that is used with the MAP statement to define a union within a structure. For detailed information about the MAP and UNION statements, see “STRUCTURE (extension)” on page 437.

USE

Provides controlled access to module entities.

Syntax

A `USE` statement has one of the following forms:

- `USE` *module-name* `[,` *rename-list* `]`
- `USE` *module-name*`, ONLY :` *access-list*

rename-list
: is a comma-separated list of *rename*

rename
: is *local-name* `=>` *module-entity-name*

access-list
: is a comma-separated list of the following:
  - `[`*local-name* `=>]` *module-entity-name*
  - `OPERATOR (`*operator*`)`
  - `ASSIGNMENT (=)`

Description

The `USE` statement provides access to a module's public specifications and definitions. These include declared variables, named constants, derived-type definitions, procedure interfaces, procedures, generic identifiers, and namelist groups. The method of access is called *use association*. Such access may be limited by an `ONLY` clause on the `USE` statement, or the accessed entities may be renamed.

All `USE` statements must appear after the program unit header statement and before any other statements. More than one `USE` statement may be present, including more than one referring to the same module.

Modules may contain `USE` statements referring to other modules; however, references must not directly or indirectly be recursive.

The local-name in a renaming operation is not declared: it assumes the attributes of the module entity being renamed.

The first two forms of the USE statement make available by use association all publicly accessible entities in the module, except that the USE statement may rename some module entities. The third form makes available only those entities specified in *access-list*, with possible renaming of some module entities.

Entities made accessible by a USE statement include public entities from other modules referenced by USE statements within the referenced module.

The same name or specifier may be made accessible by means of two or more USE statements. Such an entity must not be referenced in the scoping unit containing the USE statements, except where specific procedures can be distinguished by the overload rules. A rename or ONLY clause may be used to restrict access to one name or to rename one entity so that both are accessible.

Examples

```
MODULE rat_arith
   TYPE rat
     INTEGER  n, d
   END TYPE

   ! Make all entities public except zero.
   TYPE(rat), PRIVATE, PARAMETER :: zero = rat(0,1)
   TYPE(rat), PUBLIC, PARAMETER  :: one = rat(1,1)
   TYPE(rat)  r1, r2
   NAMELIST  /nml_rat/  r1, r2
   INTERFACE  OPERATOR( + )
     MODULE PROCEDURE rat_plus_rat, int_plus_rat
   END INTERFACE

CONTAINS
   FUNCTION rat_plus_rat(l, r)
   END FUNCTION
END MODULE

PROGRAM Mine
   ! From the module rat_arith, access only the entities rat,
   ! one, r1, r2, nml_rat but use the name one_rat for the
   ! rational value one.
   USE rat_arith, ONLY: rat, one_rat => one, r1, r2, nml_rat

   ! The OPERATOR + for rationals and the procedures rat_plus_rat
   ! and int_plus_rat are not available because of the ONLY
clause
   READ *, r2; r1 = one_rat
   WRITE( *, NML = nml_rat)
END PROGRAM
```

Related statements `MODULE`

Related concepts For information about modules, see “Modules” on page 161.

VIRTUAL (extension)

Declares an array.

Syntax

```
VIRTUAL array-declarator-list
```

array-declarator-list

is a comma-separated list of array declarators.

Description

The `VIRTUAL` statement is an HP extension in HP Fortran 90 for compatibility with earlier versions of Fortran. It is an alternative to the `DIMENSION` statement. `VIRTUAL` cannot be used as an attribute in type declaration statements.

Examples

```
VIRTUAL A(10), B(1:5,2:6)
```

Related statements

`DIMENSION`

Related concepts

Arrays are discussed in Chapter 4, “Arrays,” on page 51.

VOLATILE (extension)

Provides for data sharing between asynchronous processes.

Syntax

`VOLATILE [::]` *object-name-list*

object-name-list

is a comma-separated list of the following:

- *variable-name*
- *array-name*
- *common-block-name*

Description

It is only necessary to declare an object as `VOLATILE` when its value may be altered by an independent asynchronous process or event (for example, a signal handler). All optimization processes are inhibited for objects with the `VOLATILE` attribute. Data objects declared as `VOLATILE` are addressable by otherwise independent processes.

If an array or common block is declared as `VOLATILE`, then all of the array elements or common block variables become `VOLATILE`. Similarly, use of `EQUIVALENCE` with a `VOLATILE` object implies that any associated object is also volatile.

Examples

```
INTEGER alarm, trem
EXTERNAL wakeup
COMMON/FLAGS/ialarm
VOLATILE ialarm
trem = ALARM(60,wakeup)  ! Set an alarm to execute in 60 seconds
wakeup
IALARM = 0
DO
    IF (ialarm.NE.0) EXIT
END DO
SUBROUTINE wakeup
  COMMON/flags/ialarm
  VOLATILE ialarm
  ialarm=1
END
```

WHERE (statement and construct)

Performs masked array assignments.

Syntax

`WHERE (`*array-logical-expr*`) [`*array-assignment-statement*`]`

If the optional array-assignment clause is present, the `WHERE` statement is syntactically complete and does not require the `END WHERE` statement.

If the array-assignment clause is not present, the `WHERE` statement is the first statement of a `WHERE` construct. The syntax of the `WHERE` construct is:

```
WHERE (array-logical-expr)
   array-assignment-statement
   ...
[ ELSEWHERE
   array-assignment-statement
   ... ]
END WHERE
```

array-logical-expr

is a logical array expression.

array-assignment-statement

is an array assignment statement.

Description

Certain array elements can be selected by a mask and assigned in array-assignment statements using the `WHERE` statement or `WHERE` construct. *array-logical-expr* establishes the mask.

For any elemental operation in the array assignments, only the elements selected by the mask participate in the computation. The elemental operations include the usual intrinsic operations and the elemental intrinsic functions such as `ABS`. Masked array assignments are useful when certain elemental operations involving arrays need to be avoided because of program exceptions.

The following rules and restrictions apply:

- The shape of the result of *array-logical-expr* and the arrays in each *array-assignment-statement* must be the same; they may be of size zero.

- *array-assignment-statement* must be an intrinsic array assignment statement; no defined assignment statements are permitted.

- Each elemental operation in *array-assignment-statement* is masked by the array logical expression.
- The elements of the arrays that are used in the WHERE part (the assignments after the WHERE keyword) are those corresponding to the true elements of the array logical expression. The elements of the arrays that are used in the ELSEWHERE part (the assignments after the ELSEWHERE keyword and before the END WHERE keywords) are those corresponding to the false elements of the array logical expression.
- Each *array-assignment-statement* executes in the order in which it appears in both the WHERE and ELSEWHERE part of the WHERE construct.
- In a WHERE construct, only the WHERE statement may be a branch target statement.

Examples

```
REAL, DIMENSION(150) :: a, recip_a
REAL(DOUBLE), DIMENSION(10,20,30) :: b, sqrt_b
! Assign 1.0/a to recip_a only where a is nonzero
WHERE( a /= 0.0 ) recip_a = 1.0 / a

WHERE( b .GE. 0.0 )
   ! Assign to sqrt_b only where b is nonnegative
   sqrt_b = SQRT(b)
ELSEWHERE   ! Set sqrt_b to 0.0 where b is -ve.
   sqrt_b = 0.0
END WHERE

INTEGER, DIMENSION(no_of_tests, student):: score
CHARACTER, DIMENSION(no_of_tests, student)  :: letter_grade
! Assign letter grades for numeric scores
WHERE( score >= 92 )      letter_grade = 'A'
WHERE( score >= 82 .AND. score <= 91 ) letter_grade = 'B'
WHERE( score >= 72 .AND. score <= 81 ) letter_grade = 'C'
WHERE( score >= 62 .AND. score <= 71 ) letter_grade = 'D'
WHERE( score >=  0 .AND. score <= 61 ) letter_grade = 'E'
```

In the next example, the arrays `values`, `delta`, and `count` must all be of the same shape:

```
WHERE (ABS(values) .LT. 10.0)
   values = ABS(values) + delta
   count = count + 1
ELSEWHERE
   values = 0
   count = count + 1
ENDWHERE
```

The first two assignment statements are processed for elements corresponding to true elements of the mask. The second two assignment statements are processed for elements corresponding to false elements of the mask. Unlike the ELSE clause of an IF statement, the assignment statements in both the WHERE and ELSEWHERE parts are processed.

Note the different behavior of the calls to ABS. In evaluating the mask expression, the entire VALUES array is passed to ABS, producing an array result whose elements are then compared to 10. In the assignment statement, however, ABS is only invoked for those particular elements of VALUES corresponding to true elements of the mask. Also, note the mixed use of arrays and scalars in the assignment statement expressions.

The mask expression must have the same shape as the arrays in the assignment statements, but it might involve completely separate arrays. In the following example, A, B, and C can be independent of D and E, as long as they are all conformable:

```
WHERE (a+b .EQ. c) d = SIN(e)
```

The following example illustrates why the order of processing is important for dependency reasons:

```
REAL a(100)
REAL b(100)
EQUIVALENCE b, a
WHERE(a(1:20:1) .GT. 0) a(20:1:-1) = -1.0
WHERE(a(61:100:2) .LT. 1)  b(20:1:-1) = a(1:20:1) * 100.0
```

In the first WHERE statement, changing elements of a in the assignment might be thought to affect the mask expression. However, because the mask is evaluated before the assignment is processed, the behavior of this statement is well defined. A similar situation arises in the second WHERE statement. Assignment values to elements of the assignment variable b alter the elements of the assignment expression `a * 100.0`. Because the assignment expression is evaluated for all true elements of the mask before any transfer of values to B, the behavior is again well defined.

It is important to note that assignment statements in a WHERE construct are processed sequentially. In the next example, the second assignment is not processed until the first is completely finished. This means that the values of b used in the second assignment have been modified by the first statement:

```
WHERE (SQRT(ABS(a)) .gt. 3.0)
   b = SIN(a)
   c = SQRT(b)
ENDWHERE
```

Related statements `END` (construct) and `ELSEWHERE`

Related concepts For related information, see the following:

- The discussion of arrays in Chapter 4, “Arrays,” on page 51
- “Masked array assignment” on page 99

WRITE

Outputs data to external and internal files.

Syntax

`WRITE` (*io-specifier-list*) [*output-list*]

output-list

is a list of comma-separated data items for output. The data items can include expressions and implied-`DO`.

io-specifier-list

is a list of the following comma-separated I/O specifiers:

`[UNIT=]`*unit*

specifies the unit connected to the output file. *unit* can be one of the following:

- The name of a character variable, indicating an internal file
- An integer expression that evaluates to the unit connected to an external file
- An asterisk, indicating the preconnected unit 6 (standard output)

If the optional keyword `UNIT=` is omitted, *unit* must be the first item in *io-specifier-list*. This is the only specifier required in *io-specifier-list*.

`[FMT=]` *format*

specifies the format specification for formatting the data. *format* can be one of the following:

- An asterisk (`*`), specifying list-directed I/O
- The label of a `FORMAT` statement containing the format specification
- An integer variable that has been assigned the label of a `FORMAT` statement
- An embedded format specification

If the optional keyword `FMT=` is omitted, *format* must be the second item in *io-specifier-list*.

NOTE The `NML=` and `FMT=` specifier may not both appear in the same *io-specifier-list*.

`[NML=]`*name*

specifies the name of a namelist group for namelist-directed output. *name* must have been defined in a `NAMELIST` statement. If the optional keyword `NML=` is omitted, *name* must be the second item in the list. The first item must be the unit specifier without the optional keyword `UNIT=`.

The `NML=` and `FMT=` specifier may not both appear in the same *io-specifier-list*.

`ADVANCE=`*character-expression*

specifies whether to use advancing I/O for this statement. *character-expression* can be one of the following arguments:

`'YES'`	Use advancing formatted sequential I/O default.
`'NO'`	Use nonadvancing formatted sequential I/O.

If the `ADVANCE=` specifier appears in *io-specifier-list*, *unit* must be connected to an external file opened for formatted sequential I/O. Nonadvancing I/O is incompatible with list-directed and namelist I/O.

`ERR=`*stmt-label*

specifies the label of the executable statement to which control passes if an error occurs during statement execution.

`IOSTAT=`*integer-variable*

returns the I/O status after the statement executes. If the statement executes successfully, *integer-variable* is set to zero. If an error occurs, it is set to a positive integer that indicates which error occurred.

`REC=`*integer-expression*

specifies the number of the record to be written to the file connected for direct access. This specifier cannot appear in *io-specifier-list* with the `NML=` and `ADVANCE=` specifiers, nor with `FMT=*` (for list-directed I/O).

Description

The `WRITE` statement transfers data from internal storage to an external or internal file. An external file can be opened for sequential access or direct access I/O. If it is opened for sequential access, the `WRITE` statement can perform the following types of I/O:

- Formatted
- Unformatted
- List-directed
- Namelist-directed

If the file is opened for direct access, the `WRITE` statement can perform formatted or unformatted I/O.

`WRITE` statements operating on internal files can perform formatted or list-directed I/O.

For detailed information about files and different types of I/O, see Chapter 8, “I/O and file handling,” on page 171.

Examples

The examples in this section illustrate different uses of the `WRITE` statement.

Nonadvancing I/O

```
CHARACTER(LEN=17) :: prompt = 'Enter a number: '
WRITE (6, '(A)', ADVANCE='NO') prompt
```

The `WRITE` statement outputs to the file connected to unit 6, which is preconnected to standard output. The `ADVANCE='NO'` specifier indicates the following:

- The file has been opened for formatted sequential I/O.
- The statement uses nonadvancing I/O to read an integer formatted as four characters into the variable `prompt`.

The effect of the nonadvancing `WRITE` is to output the character string in `prompt` to standard output without a terminating newline. This means that anything subsequently entered by the user will appear on the same line.

Internal file

```
CHARACTER(LEN=80) :: cfile
WRITE (cfile, '(I5, F10.5)') i, x
```

The statement writes a string of characters into the internal file `cfile`, using the embedded format specification to perform the format conversion.

Namelist-directed I/O

In the next example, each of the four `WRITE` statements following the `NAMELIST` statement uses a different style of syntax to do exactly the same thing:

```
NAMELIST /nl/ a, b, c
WRITE (UNIT=6, NML=nl) ! 6 = standard output
WRITE (6, nl)
WRITE (*, NML=nl)     ! * = standard output
WRITE nl              ! assume standard output
```

List-directed I/O

```
WRITE (6, *) int_var
```

This statement converts the value of `int_var` to character format and outputs the character string to standard output. The format conversion is based on the type of `int_var`. If you knew the format, you could substitute for the asterisk one of the following:

- The label of the FORMAT statement with the format specification, as in:

```
WRITE (6, 100) int_var
100 FORMAT(I4)
```

- An embedded format specification itself, as in:

```
WRITE (6, '(I4)') int_var
```

Unformatted direct-access I/O

```
WRITE (31, REC=rec_num, ERR=99, IOSTAT=ios) a, b
```

This statement outputs to the file connected to unit 31. The REC= specifier indicates that the file has been opened for direct access and that this statement will output to the record whose number is stored in the variable rec_num. If an I/O error occurs during the execution of the statement, an error number will be stored in ios, and execution control will branch to the executable statement at label 99.

Related statements

CLOSE, OPEN, PRINT, and READ

Related concepts

For information about I/O concepts, see Chapter 8, “I/O and file handling,” on page 171, which also lists example programs that use I/O. For information about I/O formatting, see Chapter 9, “I/O formatting,” on page 205.

11 Intrinsic procedures

Intrinsic procedures are built-in functions and subroutines that are available by default to every Fortran 90 program and procedure. This chapter describes the intrinsic procedures provided by HP Fortran 90. All intrinsic procedures defined by the Fortran 90 Standard are supported in HP Fortran 90.

The following topics are described in this chapter:

- Basic terms and concepts
- Nonstandard intrinsic procedures
- Data representation models
- Functional categories of intrinsic procedures
- Intrinsic procedure specifications

NOTE

HP Fortran 90 intrinsic procedures are provided in the libraries `/opt/fortran90/lib/libF90.a` and `/usr/lib/libcl.a`. `/usr/lib/libcl.2` is used instead of libcl.a if using shared libraries (the default).

Basic terms and concepts

The following sections describe the terms and concepts that are used in this chapter to describe intrinsic procedures.

Availability of intrinsics

An intrinsic procedure is available in every Fortran 90 program unit except when an intrinsic and a user-defined procedure (or a library procedure) have the same name, and the user-defined procedure:

- Has the EXTERNAL attribute; see “EXTERNAL (statement and attribute)” on page 328 for more information. Library routines are declared in the user program with the EXTERNAL attribute so that they will be called instead of intrinsics that have the same name.
- Has an explicit interface; see “Procedure interface” on page 151 for a description. A statement function has an explicit interface and therefore, if it has the same name as an intrinsic, will be recognized instead of the intrinsic.

  Both a user-defined procedure and an intrinsic may have the same name when the user-defined procedure is used to extend a generic intrinsic and the argument types differ. See “Generic procedures” on page 154 for a description of this.

Subroutine and function intrinsics

Intrinsic procedures are available as functions and subroutines. In general, they behave the same as user-defined subroutines and functions. Intrinsic subroutines are invoked by the CALL statement and can return values through arguments passed to the intrinsic. Intrinsic functions can be referenced as part of an expression or in a statement that expects a value.

All interface intrinsic subroutines and functions have an explicit interface.

Generic and specific function names

The names of intrinsic functions can be either generic or specific. The name is generic—for example, ABS—if it permits arguments of different types. A name is specific—for example, IABS—if it permits arguments of one data type only.

A specific intrinsic function can be passed as an argument if it has the `INTRINSIC` attribute. A generic intrinsic function can have the `INTRINSIC` attribute if it is also the specific name, as in the case of the `SIN` intrinsic. See “Procedure dummy argument” on page 142 and the description of “`INTRINSIC` (statement and attribute)” on page 363.

NOTE Some compile-line options—for example, `+autodbl`—change the default data type sizes and can cause different or invalid intrinsic procedure references.

Classes of intrinsics

Intrinsic procedures are classified as:

- Elemental intrinsics
- Transformational functions
- Inquiry functions

The following sections describe each class. The descriptions in “Intrinsic procedure specifications” on page 487 identify the class of each intrinsic.

Elemental intrinsics

An intrinsic procedure is elemental if it is specified as having scalar arguments but will actual arguments that are arrays. Calling an elemental intrinsic with an array argument causes the function to perform the scalar operation on each element of the array. `MVBITS` is the only elemental subroutine. All other intrinsic subroutines are nonelemental.

An elemental function that is called with all scalar dummy arguments delivers a scalar result. Calling an elemental function with conformable array arguments, however, results in a conformable array result. If both array and scalar arguments are specified to an elemental function, each scalar is treated as an array in which all elements have the scalar value. The “scalar array” is conformable with the array arguments.

Transformational functions

Transformational intrinsic functions are nonelemental. Such functions require at least one array argument and return either a scalar or array result based on actual arguments that cannot be evaluated elementally. Often, an array result will be of a different shape than the argument(s). For example, `SUM` returns a scalar result that represents the sum of all the elements of the array argument.

Inquiry functions

Inquiry intrinsic functions return information based on the properties of the principal argument—its value is irrelevant, and the argument need not be defined. For example, the `SIZE` inquiry function can be used to return the extent of an array along one dimension or the total number of elements in the array.

Optimized intrinsic functions

The following intrinsics are available in millicode versions, which are optimized for performance. To get access the millicode intrinsics, you must optimize at level 2 or higher, or compile with the `+Olibcalls` option. See the *Fortran 90 Programmer's Guide* for information on this.

`acos`	`cos`	`pow`
`asin`	`exp`	`sin`
`atan`	`log`	`tan`
`atan2`	`log10`	

Nonstandard intrinsic procedures

HP Fortran 90 supports all intrinsic procedures defined by the Fortran 90 Standard. In addition, it supports the nonstandard intrinsic procedures listed in Table 53 on page 485. Like the standard intrinsics, the nonstandard intrinsics are part of the HP Fortran 90 language: their recognition is not enabled by compile-line options, and their generic nature, types, and dummy argument attributes are known to the compiler.

The nonstandard intrinsics provide:

- Additional functionality not defined in the Standard
- Compatibility with other Fortran 90 implementations
- Specific routines for data types beyond those in the Standard

Both standard and nonstandard intrinsics are described in “Intrinsic procedure specifications” on page 487.

Data representation models

The Fortran 90 Standard specifies data representation models that suggest how data are represented in the computer and how computations are performed on the data. The computations performed by some Fortran 90 intrinsic functions are described in terms of these models.

There are three data representation models in Fortran 90:

- “The Bit Model” on page 481
- “The Integer Number System Model” on page 482
- “The Real Number System Model” on page 482

In any given implementation, the model parameters are chosen to match the implementation as closely as possible. However, an exact match is not required, and the model does not impose any particular arithmetic on the implementation.

Data representation model intrinsics

Several intrinsic functions provide information about the three data representation models. These intrinsics are listed in Table 52.

Table 52 Intrinsic functions and data representation models

Intrinsic function	Description
“BIT_SIZE(I)” on page 500	Number of bits in an integer of the kind of I (I is an object, not a kind number)
“DIGITS(X)” on page 510	Base digits of precision in integer or real model for X
“EPSILON(X)” on page 515	Small value compared to 1 in real model for X
“EXPONENT(X)” on page 516	Real model exponent value for X
“FRACTION(X)” on page 517	Real model fraction value for X
“HUGE(X)” on page 519	Largest model number in integer or real model for X
“MAXEXPONENT(X)” on page 547	Maximum exponent value in real model for X

Intrinsic function	Description
“MINEXPONENT(X)” on page 551	Minimum exponent value in real model for X
“NEAREST(X, S)” on page 556	Nearest processor real value
“PRECISION(X)” on page 559	Decimal precision in real model for X
“RADIX(X)” on page 562	Base (radix) in integer or real model for X
“RANGE(X)” on page 564	Decimal exponent range in integer or real model for X
“RRSPACING(X)” on page 567	1/(relative spacing near X)
“SCALE(X, I)” on page 568	X with real model exponent changed by I
“SET_EXPONENT(X, I)” on page 571	Set the real model exponent of X to I
“SPACING(X)” on page 575	Absolute spacing near X
“TINY(X)” on page 581	Smallest number in real model for X

The Bit Model

The bit model interprets a nonnegative scalar data object *a* of type integer as a sequence of binary digits (bits), based upon the model:

$$a = \sum_{k=0}^{n-1} b_k 2^k$$

where *n* is the number of bits, given by the intrinsic function BIT_SIZE and each *b* has a bit value of 0 or 1. The bits are numbered from right to left beginning with 0.

The Integer Number System Model

The integer number system is modeled by:

$$i = s \sum_{k=0}^{q-1} d_k r^k$$

where

i	is the integer value.
s	is the sign (+1 or –1).
r	is the radix given by the intrinsic function `RADIX` (always 2 for HP systems).
q	is the number of digits (integer greater than 0), given by the intrinsic function `DIGITS`.
d	is the *k*th digit and is an integer 0 <= *d*< *r*. The digits are numbered left to right, beginning with 1.

The Real Number System Model

The real number system is modeled by:

$$x = s b^e \sum_{k=1}^{p} f_k b^{-k}$$

where

x	is the real value.
s	is the sign (+1 or –1).
b	is the base (real radix) and is an integer greater than 1, given by the intrinsic function `RADIX` (always 2 for HP systems).

e	is an integer between some minimum value (*lmin*) and maximum value (*lmax*), given by the intrinsic functions `MINEXPONENT` and `MAXEXPONENT`.
p	is the number of mantissa digits and is an integer greater than 1, given by the intrinsic function `DIGITS`.
f_k	is the *k*th digit and is an integer $0 <= f_k < b$, but f_1 may be zero only if all the f_k are zero. The digits are numbered left to right, beginning with 1.

Functional categories of intrinsic procedures

This section categorizes HP Fortran 90 intrinsic procedures based on their functionality. The procedures are divided into the following categories:

- Array construction, array inquiry, array location, array manipulation, array reduction, array reshape
- Bit inquiry, bit manipulation
- Character computation, character inquiry
- Floating-point manipulation, mathematical computation, matrix multiply, numeric computation, numeric inquiry, and vector multiply
- Kind
- Logical
- Nonstandard intrinsic procedures
- Pointer inquiry
- Presence inquiry
- Pseudorandom number
- Time
- Transfer

A listing of intrinsic procedures, ordered alphabetically by category, appears in “Intrinsic procedures by category” on page 485. More complete information on the individual intrinsic procedures is provided in “Intrinsic procedure specifications” on page 487.

Table 53 Intrinsic procedures by category

Category	Intrinsic routines
Array construction	MERGE, PACK, SPREAD, UNPACK
Array inquiry	ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND
Array location	MAXLOC, MINLOC
Array manipulation	CSHIFT, EOSHIFT, TRANSPOSE
Array reduction	ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM
Array reshape	RESHAPE
Bit inquiry	BIT_SIZE
Bit manipulation	BTEST, IAND, IBCLR, IBITS, IBSET, IEOR, IOR, ISHFT, ISHFTC, MVBITS, NOT
Character computation	ACHAR, ADJUSTL, ADJUSTR, CHAR, IACHAR, ICHAR, INDEX, LEN_TRIM, LGE, LGT, LLE, LLT, REPEAT, SCAN, TRIM, VERIFY
Character inquiry	LEN
Floating-point manipulation	EXPONENT, FRACTION, NEAREST, RRSPACING, SCALE, SET_EXPONENT, SPACING
Kind	KIND, SELECT_INT_KIND, SELECTED_REAL_KIND
Logical	LOGICAL
Mathematical computation	ACOS, ASIN, ATAN, ATAN2, COS, COSH, EXP, LOG, LOG10, SIN, SINH, SQRT, TAN, TANH
Matrix multiply	MATMUL

Category	Intrinsic routines
Nonstandard intrinsic procedures	ABORT, ACOSD, ACOSH, AND, ASIND, ASINH, ATAN2D, ATAND, ATANH, BADDRESS, COSD, DATE, DCMPLX, DFLOAT, DNUM, DREAL, EXIT, FLUSH, FNUM, FREE, FSET, FSTREAM, GETARG, GETENV, GRAN, HFIX, IACHAR, IADDR, IARGC, IDATE, IDIM, IGETARG, IJINT, IMAG, INT1, INT2, INT4, INT8, INUM, IOMSG, IQINT, IRAND, IRANP, ISIGN, ISNAN, IXOR, JNUM, LOC, LSHFT, LSHIFT, MALLOC, MCLOCK, OR, QEXT, QFLOAT, QNUM, QPROD, RAN, RAND, RNUM, RSHFT, RSHIFT, SECNDS, SIND, SIZEOF, SRAND, SYSTEM, TAND, TIME, XOR, ZEXT
Numeric computation	ABS, AIMAG, AINT, ANINT, CEILING, CMPLX, CONJG, DBLE, DIM, DPROD, FLOOR, INT, MAX, MIN, MOD, MODULO, NINT, REAL, SIGN
Numeric inquiry	DIGITS, EPSILON, HUGE, MAXEXPONENTS, MINEXPONENTS, PRECISION, RADIX, RANGE, TINY
Pointer inquiry	ASSOCIATED
Optional argument inquiry	PRESENT
Pseudorandom number	RANDOM_NUMBER, RANDOM_SEED
Time	DATE_AND_TIME, SYSTEM_CLOCK
Transfer	TRANSFER
Vector multiply	DOT_PRODUCT

Intrinsic procedure specifications

The following sections describe the HP Fortran 90 intrinsic procedures. The descriptions are ordered alphabetically, by intrinsic name. All of the intrinsics are generic. This means that the type, kind, and rank of the actual arguments can differ for each reference to the same intrinsic. In many cases, the kind and type of intrinsic function results are the same as that of the principal argument. For example, the `SIN` function may be called with any kind of real argument or any kind of complex argument, and the result has the type and kind of the argument.

Intrinsic procedure references may use keyword option. The actual argument expression is preceded by the dummy argument name—the argument keyword—and the equals sign (=). The argument keywords are shown in the descriptions.

Some intrinsic procedure's arguments are optional. Optional arguments are noted as such in the following descriptions.

ABORT()

Description Close all files, terminate the program, and cause an exception to create a core file.

Class Nonstandard subroutine.

ABS(A)

Description Absolute value.

Class Elemental function.

Argument A must be of type integer, real, or complex.

Result type/ type parameters The same as A except that if A is complex, the result is real.

Result value(s)

- If A is of type integer or real, the value of the result is |A|.
- If A is complex with value (x, y), the result is equal to a processor-dependent approximation to the square root of $(x^2 + y^2)$.

Specific forms BABS, CABS, CDABS, DABS, HABS, QABS, ZABS.

ACHAR(I)

Description Returns the character in a specified position of the ASCII collating sequence. It is the inverse of the `IACHAR` function.

Class Elemental function.

Argument `I` must be of type integer.

Result type/ type parameters Character of length one with kind type parameter value `KIND('A')`.

Result value If `I` has a value in the range 0 <= `I` <= 127, the result is the character in position `I` of the ASCII collating sequence, provided the processor is capable of representing that character; otherwise, the result is processor-dependent.

If the processor is not capable of representing both uppercase and lowercase letters and `I` corresponds to a letter in a case that the processor is not capable of representing, the result is the letter in the case that the processor is capable of representing.

`ACHAR(IACHAR(C))` must have the value `C` for any character `C` capable of representation in the processor.

ACOS(X)

Description Arccosine (inverse cosine) function in radians.

Class Elemental function.

Argument `X` must be of type real with a value that satisfies the inequality `|X|` <= 1.

Result type/ type parameters Same as `X`.

Result value The result has a value equal to a processor-dependent approximation to arccos(X), expressed in radians. It lies in the range 0 <= `ACOS(X)` <= Pi.

Specific forms `DCOS`, `QACOS`.

ACOSD(X)

Description Arccosine (inverse cosine) function in degrees.

Class Elemental nonstandard function.

Argument `X` must be of type real with a value that satisfies the inequality `|X|` <= 1.

Result type/ type parameters Same as `X`.

Result value The result has a value equal to a processor-dependent approximation to arccos(X), expressed in degrees. It lies in the range 0 <= `ACOSD(X)` <= 180.

Specific forms `DACOSD`, `QACOSD`.

ACOSH(X)

Description	Hyperbolic arccosine of radians.
Class	Elemental nonstandard function.
Argument	`X` must be of type real with a value `X` >= 1.
Result type/ type parameters	Same as `X`.
Result value	The result has a value equal to a processor-dependent approximation to the hyperbolic arccosine of `X`. It lies in the range 0 <= `ACOSH(X)`.
Specific forms	`DACOSH`, `QACOSH`.

ADJUSTL(STRING)

Description	Adjust to the left, removing leading blanks and inserting trailing blanks.
Class	Elemental function.
Argument	`STRING` must be of type character.
Result type	Character of the same length and kind type parameter as `STRING`.
Result value	The value of the result is the same as `STRING` except that any leading blanks have been deleted and the same number of trailing blanks have been inserted.

ADJUSTR(STRING)

Description	Adjust to the right, removing trailing blanks and inserting leading blanks.
Class	Elemental function.
Argument	`STRING` must be of type character.
Result type	Character of the same length and kind type parameter as `STRING`.

Result value The value of the result is the same as STRING except that any trailing blanks have been deleted and the same number of leading blanks have been inserted.

AIMAG(Z)

Description Imaginary part of a complex number.

Class Elemental function.

Argument Z must be of type complex.

Result type/ type parameters Real with the same kind type parameter as Z.

Result value If Z has the value (*x*, *y*), the result has value *y*.

AINT(A, KIND)

Optional argument KIND

Description Truncation to a whole number.

Class Elemental function.

Arguments

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result type/ type parameters The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result value If |A| < 1, AINT(A) has the value 0; if A >= 1, AINT(A) has a value equal to the integer whose magnitude is the largest integer that does not exceed the magnitude of A and whose sign is the same as the sign of A.

Specific forms DDINT, DINT, QINT.

ALL(MASK, DIM)

Optional argument `DIM`

Description Determine whether all values are `.TRUE.` in `MASK` along dimension `DIM`.

Class Transformational function.

Arguments

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with value in the range 1 <= `DIM` <= n where n is the rank of `MASK`. The corresponding actual argument must not be an optional dummy argument.

Result type, type parameters, and shape The result is of type logical with the same kind type parameter as `MASK`. It is scalar if `DIM` is absent or `MASK` has rank one; otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., $d_{\mathrm{DIM-1}}$, $d_{\mathrm{DIM+1}}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of `MASK`.

Result value

Case 1	The result of `ALL(MASK)` has the value `.TRUE.` if all elements of `MASK` are `.TRUE.` or if `MASK` has size zero, and the result has value `.FALSE.` if any element of `MASK` is `.FALSE.`.
Case 2	If `MASK` has rank one, `ALL(MASK, DIM)` has a value equal to that of `ALL(MASK)`. Otherwise, the value of element (s_1, s_2, ..., $s_{\mathrm{DIM-1}}$, $s_{\mathrm{DIM+1}}$, ..., s_n) of `ALL(MASK, DIM)` is equal to `ALL(MASK` (s_1, s_2, ..., $s_{\mathrm{DIM-1}}$, :, $s_{\mathrm{DIM+1}}$, ..., s_n)).

ALLOCATED(ARRAY)

Description Indicate whether or not an allocatable array is currently allocated.

Class Inquiry function.

Argument ARRAY must be an allocatable array.

Result type, type parameters, and shape Default logical scalar.

Result value The result has the value .TRUE. if ARRAY is currently allocated and has the value .FALSE. if ARRAY is not currently allocated. The result is undefined if the allocation status of the array is undefined.

AND(I, J)

Description Logical AND.

Class Elemental nonstandard function.

Arguments

I must be of type integer.

J must be of type integer with the same kind type parameter as I.

Result type/ type parameters Same as I.

Result value The result has the value obtained by performing a logical AND on I and J bit-by-bit according to Table 54.

Table 54 **Truth table for AND intrinsic**

I	J	AND(I, J)
1	1	1

I	J	AND(I, J)
1	0	0
0	1	0
0	0	1

The model for interpreting an integer value as a sequence of bits is described in “The Bit Model” on page 481.

ANINT(A, KIND)

Optional argument KIND

Description Nearest whole number.

Class Elemental function.

Arguments

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result type/ type parameters The result is of type real. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of A.

Result value If A>0, ANINT(A) has the value AINT(A+0.5); if A <= 0, ANINT(A) has the value AINT(A–0.5).

Specific forms DNINT, QNINT.

ANY(MASK, DIM)

Optional argument `DIM`

Description Determine whether any value is `.TRUE.` in `MASK` along dimension `DIM`.

Class Transformational function.

Arguments

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range 1 <= `DIM` <= *n*, where *n* is the rank of `MASK`. The corresponding actual argument must not be an optional dummy argument.

Result type, type parameters, and shape The result is of type logical with the same kind type parameter as `MASK`. It is scalar if `DIM` is absent or `MASK` has rank one; otherwise, the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of `MASK`.

Result value

Case 1	The result of `ANY(MASK)` has the value `.TRUE.` if any element of `MASK` is `.TRUE.` and has the value `.FALSE.` if no elements are `.TRUE.` or if `MASK` has size zero.
Case 2	If `MASK` has rank one, `ANY(MASK,DIM)` has a value equal to that of `ANY(MASK)`. Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of `ANY(MASK, DIM)` is equal to ANY(MASK(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)).

ASIN(X)

Description Arcsine (inverse sine) function in radians.

Class Elemental function.

Argument X must be of type real. Its value must satisfy the inequality |X| >= 1.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in radians. It lies in the range -Pi/2 <= ASIN(X) <= Pi/2.

Specific forms DASIN, QASIN.

ASIND(X)

Description Arcsine (inverse sine) function in degrees.

Class Elemental nonstandard function.

Argument X must be of type real. Its value must satisfy the inequality |X| <= 1.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to arcsin(X), expressed in degrees. It lies in the range
-90 <= ASIN(X) <= 90.

Specific forms DASIND, QASIND.

ASINH(X)

Description Hyperbolic arcsine of radians.

Class Elemental nonstandard function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to the hyperbolic arcsine of X.

Specific forms DASINH, QASINH.

ASSOCIATED(POINTER, TARGET)

Optional argument TARGET

Description Returns the association status of its pointer argument or indicates the pointer is associated with the target.

Class Inquiry function.

Arguments

POINTER	must be a pointer and may be of any type. Its pointer association status must not be undefined.
TARGET (optional)	must be a pointer or target. If it is a pointer, its pointer association status must not be undefined.

Result type The result is scalar of type default logical.

Result value

Case 1	If TARGET is absent, the result is .TRUE. if POINTER is currently associated with a target and .FALSE. if it is not.
Case 2	If TARGET is present and is a target, the result is .TRUE. if POINTER is currently associated with TARGET and .FALSE. if it is not.

Case 3	If TARGET is present and is a pointer, the result is .TRUE. if both POINTER and TARGET are currently associated with the same target, and is .FALSE. otherwise. If either POINTER or TARGET is disassociated, the result is .FALSE..

ATAN(X)

Description Arctangent (inverse tangent) function in radians.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to arctan(X), expressed in radians, that lies in the range -Pi/2 <= ATAN(X) <= Pi/2.

Specific forms DATAN, QATAN.

ATAN2(Y, X)

Description Arctangent (inverse tangent) function in radians. The result is the principal value of the argument of the nonzero complex number (X, Y).

Class Elemental function.

Arguments

Y	must be of type real.
X	must be of the same type and kind type parameter as Y. If Y has the value zero, X must not have the value zero.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in radians.

The result lies in the range -Pi <= `ATAN2(Y, X)` <= Pi and is equal to a processor-dependent approximation to a value of arctan(Y/X) if X is not 0.

If Y>0, the result is positive. If Y=0, the result is zero if X>0 and the result is Pi if X<0. If Y<0, the result is negative. If X=0, the absolute value of the result is Pi/2.

Specific forms `DATAN2`, `QATAN2`.

ATAN2D(Y, X)

Description Arctangent (inverse tangent) function in degrees.

Class Elemental nonstandard function.

Arguments

Y	must be of type real.
X	must be of the same type and kind type parameter as Y.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to the principal value of the argument of the complex number (X, Y), expressed in degrees, that lies in the range
-90 < `ATAN2D(Y,X)` < 90.

Specific forms `DATAN2D`, `QATAN2D`.

ATAND(X)

Description Arctangent (inverse tangent) function in degrees.

Class Elemental nonstandard function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to arctan(X), expressed in degrees, that lies in the range
-90 < `ATAND(X)` < 90.

Specific forms `DATAND`, `QATAND`.

ATANH(X)

Description Hyperbolic arctangent of radians.

Class Elemental nonstandard function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to the hyperbolic arctangent of X.

Specific forms `DATANH`, `QATANH`.

BADDRESS(X)

Description Return the address of X.

Class Inquiry nonstandard function.

Argument X may be of any type.

Result type The result is of type default integer.

BIT_SIZE(I)

Description Returns the number of bits *n*, defined by the model described in “The Bit Model” on page 481, for integers with the kind parameter of the argument.

Class Inquiry function.

Argument I must be of type integer.

Result type, type parameters, and shape Scalar integer with the same kind type parameter as I.

Result value The result has the value of the number of bits *n* in the model integer, defined for bit manipulation contexts in "The Bit Model" on page 481, for integers with the kind parameter of the argument.

BTEST(I, POS)

Description Tests a bit of an integer value.

Class Elemental function.

Arguments

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and be less than BIT_SIZE(I).

Result type The result is of type default logical.

Result value The result has the value .TRUE. if bit POS of I has the value 1 and has the value .FALSE. if bit POS of I has the value 0. The model for the interpretation of an integer value as a sequence of bits is described in "The Bit Model" on page 481.

Specific forms BBTEST, BITEST, BJTEST, BKTEST, HTEST.

CEILING(A)

Description Returns the least integer greater than or equal to its argument.

Class Elemental function.

Argument A must be of type real.

Result type/ type parameters Default integer.

Result value

The result has a value equal to the least integer greater than or equal to A. The result is undefined if the processor cannot represent this value in the default integer type.

CHAR(I, KIND)

Optional argument

KIND

Description

Returns the character in a given position of the processor collating sequence associated with the specified kind type parameter. It is the inverse of the function ICHAR.

Class

Elemental function.

Arguments

I	must be of type integer with a value in the range 0 <= I <= *n*-1, where *n* is the number of characters in the collating sequence associated with the specified kind type parameter.
KIND (optional)	must be a scalar integer initialization expression.

Result type/ type parameters

Character of length one. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default character type.

Result value

The result is the character in position I of the collating sequence associated with the specified kind type parameter.

`ICHAR(CHAR(I, KIND(C)))` must have the value I for 0 <= I <= *n*-1 and `CHAR(ICHAR(C), KIND(C))` must have the value C for any character C capable of representation in the processor.

CMPLX(X, Y, KIND)

Optional arguments

Y, KIND

Description

Convert to complex type.

Class

Elemental function.

Arguments

X	must be of type integer, real, or complex.

Y (optional)	must be of type integer or real. It must not be present if X is of type complex.
KIND (optional)	must be a scalar integer initialization expression.

Result type/ type parameters

The result is of type complex. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default real type.

Result value

- If Y is absent and X is not complex, it is as if Y were present with the value zero.
- If Y is absent and X is complex, it is as if Y were present with the value AIMAG(X).

CMPLX(X,Y,KIND) has the complex value whose real part is REAL(X,KIND) and whose imaginary part is REAL(Y,KIND).

CONJG(Z)

Description

Conjugate of a complex number.

Class

Elemental function.

Argument

Z must be of type complex.

Result type/ type parameters

Same as Z.

Result value

If Z has the value (x, y), the result has the value (x, -y).

Specific forms

DCONJG.

COS(X)

Description

Cosine function in radians.

Class

Elemental function.

Argument

X must be of type real or complex.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to cos(X). If X is of type real, it is regarded as a value in radians. If X is of type complex, its real part is regarded as a value in radians.

Specific forms CCOS, CDCOS, DCOS, QCOS, ZCOS.

COSD(X)

Description Cosine function in degrees.

Class Elemental nonstandard function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to cos(X).

Specific forms DCOSD, QCOSD.

COSH(X)

Description Hyperbolic cosine function.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to cosh(X).

Specific forms DCOSH, QCOSH.

COUNT(MASK, DIM)

Optional argument DIM

Description Count the number of .TRUE. elements of MASK along dimension DIM.

Class Transformational function.

Arguments

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range 1 <= DIM <= *n*, where *n* is the rank of MASK. The corresponding actual argument must not be an optional dummy argument.

Result type, type parameters, and shape The result is of type default integer. It is scalar if DIM is absent or MASK has rank one; otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of MASK.

Result value

Case 1	The result of COUNT(MASK) has a value equal to the number of .TRUE. elements of MASK or has the value zero if MASK has size zero.
Case 2	If MASK has rank one, COUNT(MASK, DIM) has a value equal to that of COUNT(MASK). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of COUNT(MASK, DIM) is equal to COUNT(MASK(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)).

Specific forms KCOUNT.

CSHIFT(ARRAY, SHIFT, DIM)

Optional argument DIM

Description Perform a circular shift on an array expression of rank one, or perform circular shifts on all the complete rank one sections along a given dimension of an array expression of rank two or greater.

Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

Class Transformational function.

Arguments

ARRAY	may be of any type. It must not be scalar.
SHIFT	must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank n-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.
DIM (optional)	must be a scalar and of type integer with a value in the range 1 <= DIM <= n, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

Result type, type parameters, and shape The result is of the type and type parameters of ARRAY, and has the shape of ARRAY.

Result value

Case 1	If ARRAY has rank one, element i of the result is ARRAY(1 + MODULO(i + SHIFT – 1, SIZE(ARRAY))).
Case 2	If ARRAY has rank greater than one, section (s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) of the result has a value equal to CSHIFT(ARRAY(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n), sh, 1), where sh is SHIFT or SHIFT(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n).

Specific forms KCSHIFT.

DATE(DATESTR)

Description	Return current system date.
Class	Nonstandard subroutine.
Argument	`DATESTR` must be of type character. It must be a character string of length 9 or more.

DATE_AND_TIME(DATE, TIME, ZONE, VALUES)

Optional arguments `DATE`, `TIME`, `ZONE`, `VALUES`

Description Returns data on the real-time clock and date in a form compatible with the representations defined in ISO 8601:1988 ("Data elements and interchange formats — Information interchange — Representation of dates and times").

Class Subroutine.

Arguments

`DATE` (optional) must be scalar and of type default character, and must be of length at least 8 in order to contain the complete value. It is an `INTENT(OUT)` argument. Its leftmost 8 characters are set to a value of the form *CCYYMMDD*, where *CC* is the century, *YY* the year within the century, *MM* the month within the year, and *DD* the day within the month. If there is no date available, they are set to blank.

`TIME` (optional) must be scalar and of type default character, and must be of length at least 10 in order to contain the complete value. It is an `INTENT(OUT)` argument. Its leftmost 10 characters are set to a value of the form *hhmmss.sss*, where *hh* is the hour of the day, *mm* is the minutes of the hour, and *ss.sss* is the seconds and milliseconds of the minute. If there is no clock available, they are set to blank.

`ZONE` (optional) must be scalar and of type default character, and must be of length at least 5 in order to contain the complete value. It is an `INTENT(OUT)` argument. Its leftmost 5 characters are set to a value of the form (+/-)*hhmm*, where *hh* and *mm* are the time difference with respect to Coordinated Universal Time (UTC) in hours and parts of an hour expressed in minutes, respectively. If there is no clock available, they are set to blank.

VALUES (optional)
: must be of type default integer and of rank one. It is an `INTENT(OUT)` argument. Its size must be at least 8. The values returned in VALUES are as follows:

`VALUES(1)`	the year (for example, 1990), or –`HUGE(0)` if there is no date available;
`VALUES(2)`	the month of the year, or –`HUGE(0)` if there is no date available;
`VALUES(3)`	the day of the month, or –`HUGE(0)` if there is no date available;
`VALUES(4)`	the time difference with respect to Coordinated Universal Time (UTC) in minutes, or –`HUGE(0)` if this information is not available;
`VALUES(5)`	the hour of the day, in the range of 0 to 23, or –`HUGE(0)` if there is no clock;
`VALUES(6)`	the minutes of the hour, in the range 0 to 59, or –`HUGE(0)` if there is no clock;
`VALUES(7)`	the seconds of the minute, in the range 0 to 60, or –`HUGE(0)` if there is no clock;
`VALUES(8)`	the milliseconds of the second, in the range 0 to 999, or –`HUGE(0)` if there is no clock.

The HUGE intrinsic function is described in “HUGE(X)” on page 519.

DBLE(A)

Description Convert to double precision real type.

Class Elemental function.

Argument A must be of type integer, real, or complex.

Result type/ type parameters Double precision real.

Result value

Case 1	If A is of type double precision real, DBLE(A) = A.
Case 2	If A is of type integer or real, the result is as much precision of the significant part of A as a double precision real datum can contain.
Case 3	If A is of type complex, the result is as much precision of the significant part of the real part of A as a double precision real datum can contain.

Specific forms DBLEQ.

DCMPLX(X,Y)

Optional argument Y

Description Convert to double precision complex type.

Class Elemental nonstandard function.

Arguments

X	must be of type integer, real, or complex.
Y	must not be supplied if X is of type complex; otherwise is optional and must be of the same type and kind type parameter as X.

Result type/ type parameters Double precision complex.

DFLOAT(A)

Description Convert to double precision type.

Class Elemental nonstandard function.

Argument A must be of type integer.

Result type/ type parameters Double precision.

Specific forms `DFLOTI`, `DFLOTJ`, `DFLOTK`.

DIGITS(X)

Description Returns the number of significant digits in the model representing numbers of the same type and kind type parameter as the argument.

Class Inquiry function.

Argument X must be of type integer or real. It may be scalar or array valued.

Result type, type parameters, and shape Default integer scalar.

Result value The result has the value *q* if X is of type integer and *p* if X is of type real, where *q* and *p* are as defined in “Data representation models” on page 480 for the model representing numbers of the same type and kind type parameter as X.

DIM(X, Y)

Description The difference X–Y if it is positive; otherwise zero.

Class Elemental function.

Arguments

X	must be of type integer or real.
Y	must be of the same type and kind type parameter as X.

Result type/ type parameters Same as X.

Result value The value of the result is X–Y if X > Y and zero otherwise.

Specific forms BDIM, DDIM, HDIM, QDIM.

DNUM(I)

Description Convert to double precision.

Class Elemental nonstandard function.

Argument I must be of type character.

Result type Double precision.

DOT_PRODUCT(VECTOR_A, VECTOR_B)

Description Performs dot-product multiplication of numeric or logical vectors.

Class Transformational function.

Arguments

VECTOR_A	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.
VECTOR_B	must be of numeric type if VECTOR_A is of numeric type or of type logical if VECTOR_A is of type logical. It must be array valued and of rank one. It must be of the same size as VECTOR_A.

Result type, type parameters, and shape

The result is scalar.

If the arguments are of numeric type, the type and kind type parameter of the result are those of the expression `VECTOR_A * VECTOR_B` determined by the types of the arguments.

If the arguments are of type logical, the result is of type logical with the kind type parameter of the expression `VECTOR_A .AND. VECTOR_B`.

Result value

Case 1	If `VECTOR_A` is of type integer or real, the result has the value `SUM(VECTOR_A*VECTOR_B)`. If the vectors have size zero, the result has the value zero.
Case 2	If `VECTOR_A` is of type complex, the result has the value `SUM(CONJG(VECTOR_A)*VECTOR_B)`. If the vectors have size zero, the result has the value zero.
Case 3	If `VECTOR_A` is of type logical, the result has the value `ANY(VECTOR_A .AND. VECTOR_B)`. If the vectors have size zero, the result has the value `.FALSE.`.

DPROD(X, Y)

Description

Double precision real product.

Class

Elemental function.

Arguments

`X`	must be of type default real.
`Y`	must be of type default real.

Result type/ type parameters

Double precision real.

Result value

The result has a value equal to a processor-dependent approximation to the product of `X` and `Y`.

DREAL(A)

Description Convert to double precision.

Class Elemental nonstandard function.

Argument A must be of type integer, real, or complex.

Result Double precision.

EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)

Optional arguments BOUNDARY, DIM

Description Perform an end-off shift on an array expression of rank one or perform end-off shifts on all the complete rank-one sections along a given dimension of an array expression of rank two or greater.

Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end.

Different sections may have different boundary values and may be shifted by different amounts and in different directions (positive for left shifts, negative for right shifts).

Class Transformational function.

Arguments

ARRAY
: may be of any type. It must not be scalar.

SHIFT
: must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank n-1 and of shape (d_1, d_2, ..., $d_{\text{DIM-1}}$, $d_{\text{DIM+1}}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

BOUNDARY (optional)
: must be of the same type and type parameters as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either scalar or of rank n-1 and of shape (d_1, d_2, ..., $d_{\text{DIM-1}}$, $d_{\text{DIM+1}}$, ..., d_n). BOUNDARY may be omitted for the data types listed in Table 55, which lists the default values of BOUNDARY for each data type.

Table 55 **Default values for the BOUNDARY argument**

Data type of ARRAY	Default value of BOUNDARY
Integer	0
Real	0.0
Complex	(0.0, 0.0)
Logical	`.FALSE.`
Character (*len*)	*len* blanks

`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. If `DIM` is omitted, it is as if it were present with the value 1.

Result type, type parameters, and shape

The result has the type, type parameters, and shape of ARRAY.

Result value

Element (s_1, s_1, ..., s_n) of the result has the value ARRAY (s_1, s_2, ..., s_{DIM-1}, s_{DIM} + *sh*, s_{DIM+1}, ..., s_n) where *sh* is SHIFT or SHIFT(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) provided the inequality `LBOUND(ARRAY, DIM)` <= s_{DIM} + *sh* <= `UBOUND(ARRAY, DIM)` holds and is otherwise BOUNDARY or BOUNDARY(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n).

Specific forms

`KEOSHIFT.`

EPSILON(X)

Description

Returns a positive model number that is almost negligible compared to unity in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function.

Argument

X must be of type real. It may be scalar or array valued.

Result type, type parameters, and shape

Scalar of the same type and kind type parameter as X.

Result value

The result has the value b^{1-p} where b and p are as defined in "The Real Number System Model" on page 482 for the model representing numbers of the same type and kind type parameter as X.

EXIT(STATUS)

Optional argument

STATUS

Description

Close all files and terminate the program.

Class

Nonstandard subroutine.

Argument

STATUS must be of type integer.

If STATUS is supplied, the calling program exits with a return code status of STATUS. Otherwise the return code status is indeterminate.

In `csh` the `$status` environment variable holds the return code for the last executed command. In `ksh`, the `$?` environment variable holds the return code.

EXP(X)

Description Exponential.

Class Elemental function.

Argument X must be of type real or complex.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to e^{X}. If X is of type complex, its imaginary part is regarded as a value in radians.

Specific forms `CEXP`, `CDEXP`, `DEXP`, `QEXP`, `ZEXP`.

EXPONENT(X)

Description Returns the exponent part of the argument when represented as a model number.

Class Elemental function.

Argument X must be of type real.

Result type Default integer.

Result value The result has a value equal to the exponent *e* of the model representation (see “The Real Number System Model” on page 482) for the value of X, provided X is nonzero and *e* is within the range for default integers. The result is undefined if the processor cannot represent *e* in the default integer type. `EXPONENT(X)` has the value zero if X is zero.

FLOOR(A)

Description Returns the greatest integer less than or equal to its argument.

Class Elemental function.

Argument A must be of type real.

Result type/ type parameters Default integer.

Result value The result has a value equal to the greatest integer less than or equal to A. The result is undefined if the processor cannot represent this value in the default integer type.

FLUSH(LUNIT)

Description Flush pending I/O on a logical unit.

Class Nonstandard subroutine.

FNUM(UNIT)

Description Get an operating system file descriptor.

Class Inquiry nonstandard function.

FRACTION(X)

Description Returns the fractional part of the model representation of the argument value.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has the value X * b^{-e}, where b and e are as defined in "The Real Number System Model" on page 482. If X has the value zero, the result has the value zero.

FREE(P)

Description Free a block of memory.

Class Nonstandard subroutine.

FSET(UNIT, NEWFD, OLDFD)

Description Attach a system file descriptor to a logical unit.

Class Nonstandard subroutine.

FSTREAM(UNIT)

Description Retrieve a C language FILE stream pointer.

Class Inquiry nonstandard function.

GETARG(N, STRING)

Description Get the arguments passed to the program.

Class Nonstandard subroutine.

Arguments

`N` must be of type integer. `N` specifies which command-line argument is requested. When `N`=1, it returns the program name. When `N`=0, it returns all blanks.

`STRING` must be a character variable. It is assigned the requested command-line argument, padded with blanks on the end. If the requested argument is longer than `STRING`, a truncated version is assigned to `STRING`.

GETENV(VAR, VALUE)

Description Return the value of a system environment variable.

Class Nonstandard subroutine.

Arguments VAR and VALUE are of type character. VAR specifies the environment variable name. The character variable VALUE is assigned the environment variable's value. VALUE must be declared large enough to hold the value. If the environment variable is not defined VALUE is set to all blanks.

GRAN()

Description Generate Gaussian normal random numbers.

Class Elemental nonstandard function.

Result REAL(4). The numbers generated by GRAN have a mean of 0.0, a standard deviation of 1.0, and a range of approximately -5.0 through +5.0.

HFIX(A)

Description Convert to INTEGER(2) type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, double precision, or complex.

Result INTEGER(2) type.

HUGE(X)

Description Returns the largest number in the model representing numbers of the same type and kind type parameter as the argument.

Class Inquiry function.

Argument X must be of type integer or real. It may be scalar or array valued.

Result type, type parameters, and shape

Scalar of the same type and kind type parameter as X.

Result value

The result has the value r^q - 1 if X is of type integer and

$(1 - b^{-p})\ b$ ** e_{max}

if X is of type real, where r, q, b, p, and e_{max} are as defined in "The Real Number System Model" on page 482.

IACHAR(C)

Description

Returns the position of a character in the ASCII collating sequence.

Class

Elemental function.

Argument

C must be of type default character and of length one.

Result type/ type parameters

Default integer.

Result value

If C is in the collating sequence defined by the codes specified in ISO 646:1983 ("Information technology — ISO 7-bit coded character set for information interchange"), the result is the position of C in that sequence and satisfies the inequality (0 <= IACHAR(C) <= 127).

A processor-dependent value is returned if C is not in the ASCII collating sequence. The results are consistent with the LGE, LGT, LLE, and LLT lexical comparison functions. For example, if LLE(C, D) is .TRUE., IACHAR(C) .LE. IACHAR(D) is .TRUE. where C and D are any two characters representable by the processor.

IADDR(X)

Description

Return the address of X.

Class

Inquiry nonstandard function.

Argument

X may be of any type.

Result type

The result is of type default integer.

See "BADDRESS(X)" on page 500 for examples.

IAND(I, J)

Description Performs a bitwise logical AND.

Class Elemental function.

Arguments

`I` must be of type integer.

`J` must be of type integer with the same kind type parameter as `I`.

Result type/ type parameters Same as `I`.

Result value The result has the value obtained by combining `I` and `J` bit-by-bit according to Table 56.

Table 56 Truth table for IAND intrinsic

I	J	IAND(I, J)
1	1	1
1	0	0
0	1	0
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in "The Bit Model" on page 481.

Specific forms `BIAND`, `HIAND`, `IIAND`, `JIAND`, `KIAND`.

IARGC()

Description Get the number of arguments passed to the program.

Class Elemental nonstandard function.

Result type Integer.

Result value If no arguments are passed to the program, IARGC returns zero. Otherwise IARGC returns a count of the arguments that follow the program name on the command line.

IBCLR(I, POS)

Description Clears a bit to zero.

Class Elemental function.

Arguments

I	must be of type integer.
POS	must be of type integer. It must be nonnegative and less than BIT_SIZE(I).

Result type/ type parameters Same as I.

Result value The result has the value of the sequence of bits of I, except that bit POS of I is set to zero. The model for the interpretation of an integer value as a sequence of bits is in “The Bit Model” on page 481.

Specific forms BBCLR, HBCLR, IIBCLR, JIBCLR, KIBCLR.

IBITS(I, POS, LEN)

Description	Extracts a sequence of bits.	
Class	Elemental function.	
Arguments	`I`	must be of type integer.
	`POS`	must be of type integer. It must be nonnegative and `POS + LEN` must be less than or equal to `BIT_SIZE(I)`.
	`LEN`	must be of type integer and nonnegative.
Result type/ type parameters	Same as `I`.	
Result value	The result has the value of the sequence of `LEN` bits in `I` beginning at bit `POS`, right-adjusted and with all other bits zero. The model for the interpretation of an integer value as a sequence of bits is in “The Bit Model” on page 481.	
Specific forms	`BBITS`, `HBITS`, `IIBITS`, `JIBITS`, `KIBITS`.	

IBSET(I, POS)

Description	Sets a bit to one.	
Class	Elemental function.	
Arguments	`I`	must be of type integer.
	`POS`	must be of type integer. It must be nonnegative and less than `BIT_SIZE(I)`.
Result type/ type parameters	Same as `I`.	
Result value	The result has the value of the sequence of bits of `I`, except that bit `POS` of `I` is set to one. The model for the interpretation of an integer value as a sequence of bits is in “The Bit Model” on page 481.	
Specific forms	`HBSET`, `IIBSET`, `JIBSET`, `KIBSET`.	

ICHAR(C)

Description Returns the position of a character in the processor collating sequence associated with the kind type parameter of the character.

Class Elemental function.

Argument `C` must be of type character and of length one. Its value must be that of a character capable of representation in the processor.

Result type/ type parameters Default integer.

Result value The result is the position of `C` in the processor collating sequence associated with the kind type parameter of `C` and is in the range 0 <= `IACHAR(C)` <*n*-1, where *n* is the number of characters in the collating sequence.

For any characters `C` and `D` capable of representation in the processor, `C.LE.D` is `.TRUE.` if and only if `ICHAR(C) .LE. ICHAR(D)` is `.TRUE.`, and `C.EQ.D` is `.TRUE.` if and only if `ICHAR(C).EQ. ICHAR(D)` is `.TRUE.`.

IDATE(MONTH, DAY, YEAR)

Description Return the month, day, and year of current system.

Class Nonstandard subroutine.

Arguments `MONTH`, `DAY`, and `YEAR` must be of type integer.

IDIM(X, Y)

Description Integer positive difference.

Class Nonstandard function.

Arguments

X	must be of type integer.
Y	must be of type integer with the same kind type parameter as X.

Result type/ type parameters Integer of same kind type parameter as X.

Result value If X > Y, IDIM(X, Y) is X-Y. If X <= Y, IDIM(X, Y) is zero.

Specific forms IIDIM, JIDIM. KIDIM.

IEOR(I, J)

Description Performs a bitwise exclusive OR.

Class Elemental function.

Arguments

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result type/ type parameters Same as I.

Result value The result has the value obtained by combining I and J bit-by-bit according to Table 57.

Table 57 **Truth table for IEOR intrinsic**

I	J	IEOR(I, J)
1	1	0
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in “The Bit Model” on page 481.

Specific forms BIEOR, HIEOR, IIEOR, JIEOR, KIEOR.

IGETARG(N, STR, STRLEN)

Description Get command-line argument.

Class Inquiry nonstandard function.

Arguments

N — must be of type integer. N specifies which command-line argument is requested. When N=0, it returns the program name.

STR — must be a character variable. It is assigned first STRLEN characters of the requested command-line argument, padded with blanks on the end. If the requested argument is longer than STR, a truncated version is assigned to STR.

STRLEN — must be of type integer. STRLEN specifies the number of characters of argument N to assign to STR.

Result value IGETARG returns an integer value, either -1 if the requested argument was not found, or a positive integer that indicates the number of characters copied from the command line to STR.

IJINT(A)

Description Convert to `INTEGER(2)` type.

Class Elemental nonstandard function.

Argument `A` must be of type `INTEGER(4)`.

Result `INTEGER(2)` type.

IMAG(A)

Description Imaginary part of complex number.

Class Elemental nonstandard function.

Argument `A` must be of type complex or double complex.

Result Real if `A` is complex. Double precision if `A` is double complex.

INDEX(STRING, SUBSTRING, BACK)

Optional argument `BACK`

Description Returns the starting position of a substring within a string.

Class Elemental function.

Arguments

`STRING`	must be of type character.
`SUBSTRING`	must be of type character with the same kind type parameter as `STRING`.
`BACK` (optional)	must be of type logical.

Result type/ type parameters Default integer.

Result value

Case 1	If `BACK` is absent or present with the value `.FALSE.`, the result is the minimum positive value of `I` such that `STRING(I : I + LEN(SUBSTRING) – 1) = SUBSTRING` or zero if there is no such value.

	Zero is returned if `LEN(STRING)` < `LEN(SUBSTRING)` and one is returned if `LEN(SUBSTRING)` = 0.
Case 2	If `BACK` is present with the value `.TRUE.`, the result is the maximum value of `I` less than or equal to `LEN(STRING)` – `LEN(SUBSTRING)` + 1 such that `STRING(I : I + LEN(SUBSTRING) - 1)` = `SUBSTRING` or zero if there is no such value.
	Zero is returned if `LEN(STRING)` < `LEN(SUBSTRING)` and `LEN(STRING)` + 1 is returned if `LEN(SUBSTRING)` = 0.

Specific forms `KINDEX`.

INT(A, KIND)

Optional argument `KIND`

Description Convert to integer type.

Class Elemental function.

Arguments

`A`	must be of type integer, real, or complex.
`KIND` (optional)	must be a scalar integer initialization expression.

Result type/ type parameters Integer. If `KIND` is present, the kind type parameter is that specified by `KIND`; otherwise, the kind type parameter is that of default integer type.

Result value

Case 1	If `A` is of type integer, `INT(A)` = `A`.
Case 2	If `A` is of type real, there are two cases: if |`A`| < 1, `INT(A)` has the value 0; if |`A`| >= 1, `INT(A)` is the integer whose magnitude is the largest integer that does not exceed the magnitude of `A` and whose sign is the same as the sign of `A`.
Case 3	If `A` is of type complex, `INT(A)` is the value obtained by applying the above rules (for reals) to the real part of `A`. The result is undefined if the processor cannot represent the result in the specified integer type.

Specific forms `IFIX`, `IIFIX`, `IINT`, `JIFIX`, `JINT`, `KIFIX`, `KINT`.

INT1(A)

Description Convert to `INTEGER(1)` type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, or complex.

Result `INTEGER(1)` type. If A is complex, `INT1(A)` is equal to the truncated real portion of A.

INT2(A)

Description Convert to `INTEGER(2)` type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, or complex.

Result `INTEGER(2)` type. If A is complex, `INT2(A)` is equal to the truncated real portion of A.

INT4(A)

Description Convert to `INTEGER(4)` type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, or complex.

Result `INTEGER(4)` type. If A is complex, `INT4(A)` is equal to the truncated real portion of A.

INT8(A)

Description Convert to INTEGER(8) type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, or complex.

Result INTEGER(8) type. If A is complex, INT8(A) is equal to the truncated real portion of A.

Specific forms IDINT.

INUM(I)

Description Convert character to INTEGER(2) type.

Class Elemental nonstandard function.

Argument I must be of type character.

Result INTEGER(2) type.

IOMSG(N, MSG)

Description Print the text for an I/O message.

Class Nonstandard subroutine.

IOR(I, J)

Description Performs a bitwise inclusive OR.

Class Elemental function.

Arguments

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result type/ type parameters Same as `I`.

Result value The result has the value obtained by combining `I` and `J` bit-by-bit according to Table 58.

Table 58 **Truth table for IOR intrinsic**

I	J	IOR(I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for the interpretation of an integer value as a sequence of bits is in “The Bit Model” on page 481.

Specific forms `BIOR`, `HIOR`, `IIOR`, `JIOR`, `KIOR`

IQINT(A)

Description Convert to integer type.

Class Elemental nonstandard function.

Argument `A` must be of type `REAL(16)`.

Result Integer type.

Specific forms `IIQINT`, `JIQINT`, `KIQINT`.

IRAND()

Description Generate pseudorandom numbers.

Class Elemental nonstandard function.

Result type/ type parameters `INTEGER(4)` type.

Result value `RAND` generates numbers in the range 0 through 2^{15}-1.

NOTE For details about restarting the pseudorandom number generator used by `IRAND` and `RAND`, see “SRAND(ISEED)” on page 577.

IRANP(X)

Description Generate Poisson-distributed random numbers.

Class Elemental nonstandard function.

Argument `X` must be of `REAL(4)` type and must be in the range 0.0 through 87.33. For better performance, it is recommended that `X` be less than 50.0 (see “Result value” below).

Result type/ type parameters `INTEGER(4)` type.

Result value `IRANP` returns an error code of -1 if X <= 0.0.

`IRANP` returns an error code of -2 if X > 87.33.

`IRANP` calculates exponentially distributed random numbers until the product is less than `exp(-X)`. The random number returned by `IRANP` is the number of exponentials needed, minus 1. `IRANP` makes an average of `X`+1 calls to `RAND`, so it is recommended that `X` be less than 50.

ISHFT(I, SHIFT)

Description Performs a logical shift.

Class Elemental function.

Arguments

`I`	must be of type integer.
`SHIFT`	must be of type integer. The absolute value of `SHIFT` must be less than or equal to `BIT_SIZE(I)`.

Result type/ type parameters Same as `I`.

Result value The result has the value obtained by shifting the bits of `I` by `SHIFT` positions.

If `SHIFT` is positive, the shift is to the left; if `SHIFT` is negative, the shift is to the right; and if `SHIFT` is zero, no shift is performed. Bits shifted out from the left or from the right, as appropriate, are lost. Zeros are shifted in from the opposite end.

The model for the interpretation of an integer value as a sequence of bits is described in “The Bit Model” on page 481.

Specific forms `BSHFT`, `HSHFT`, `IISHFT`, `JISHFT`, `KISHFT`.

ISHFTC(I, SHIFT, SIZE)

Optional argument `SIZE`

Description Performs a circular shift of the rightmost bits.

Class Elemental function.

Arguments

`I`	must be of type integer.
`SHIFT`	must be of type integer. The absolute value of `SHIFT` must be less than or equal to `SIZE`.
`SIZE` (optional)	must be of type integer. The value of `SIZE` must be positive and must not exceed `BIT_SIZE(I)`. If `SIZE` is absent, it is as if it were present with the value of `BIT_SIZE(I)`.

Result type/ type parameters
Same as `I`.

Result value
The result has the value obtained by shifting the `SIZE` rightmost bits of `I` circularly by `SHIFT` positions.

If `SHIFT` is positive, the shift is to the left; if `SHIFT` is negative, the shift is to the right; and if `SHIFT` is zero, no shift is performed. No bits are lost. The unshifted bits are unaltered.

The model for the interpretation of an integer value as a sequence of bits is described in “The Bit Model” on page 481.

Specific forms
`HSHFTC`, `ISHFTC`, `JISHFTC`, `KISHFTC`.

ISIGN(A, B)

Description
Absolute value of `A` times the sign of `B`.

Class
Elemental nonstandard function.

Arguments

`A`	must be of type integer.
`B`	must be of type integer with the same kind type parameter as `A`.

Result type/ type parameters
Same as `A`.

Result value
The value of the result is `|A|` if `B>=` 0 and `-|A|` if `B` < 0.

ISNAN(X)

Description
Determine if a value is NaN (not a number).

Class
Elemental nonstandard function.

Argument
`X` must be of type real.

Result type
Logical.

IXOR(I, J)

Description Exclusive `OR`.

Class Elemental nonstandard function.

Arguments

`I` must be of type integer.

`J` must be of type integer with the same kind type parameter as `I`.

Result type/ type parameters Same as `I`.

Result value The result has the value obtained by performing an exclusive `OR` on `I` and `J` bit-by-bit according to Table 59.

Table 59 Truth table for IXOR intrinsic

I	J	IXOR(I, J)
1	1	0
1	0	1
0	1	1
0	0	0

The model for interpreting an integer value as a sequence of bits is described in “The Bit Model” on page 481.

Specific forms `BIXOR`, `HIXOR`, `IIXOR`, `JIXOR`.

IZEXT(A)

Description Zero extend.

Class Generic elemental nonstandard function.

Argument `A` must be of type `INTEGER(1)`, `INTEGER(2)`, `LOGICAL(1)`, or `LOGICAL(2)`.

Result type/ type parameters The result is of type `INTEGER(2)`.

Result `IZEXT` converts `A` to `INTEGER(2)` by sign-extending zeroes instead of the actual sign bit.

JNUM(I)

Description Convert character to integer type.

Class Elemental nonstandard function.

Argument `I` must be of type character.

Result Integer type.

JZEXT(A)

Description Zero extend.

Class Generic elemental nonstandard function.

Argument `A` must be of type `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, `LOGICAL(1)`, `LOGICAL(2)`, or `LOGICAL(4)`.

Result type/ type parameters The result is of type `INTEGER(4)`.

Result `JZEXT` converts `A` to `INTEGER(4)` by sign-extending zeroes instead of the actual sign bit.

KIND(X)

Description Returns the value of the kind type parameter of X.

Class Inquiry function.

Argument X may be of any intrinsic type.

Result type, type parameters, and shape Default integer scalar.

Result value The result has a value equal to the kind type parameter value of X.

KZEXT(A)

Description Zero extend.

Class Generic elemental nonstandard function.

Argument A must be of type `INTEGER(1)`, `INTEGER(2)`, `INTEGER(4)`, `INTEGER(8)`, `LOGICAL(1)`, `LOGICAL(2)`, `LOGICAL(4)`, or `LOGICAL(8)`.

Result type/ type parameters The result is of type `INTEGER(8)`.

Result KZEXT converts A to `INTEGER(8)` by sign-extending zeroes instead of the actual sign bit.

LBOUND(ARRAY, DIM)

Optional argument DIM

Description Returns all the lower bounds or a specified lower bound of an array.

Class Inquiry function.

Arguments

ARRAY: may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated.

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.

Result type, type parameters, and shape 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.

Result value

Case 1: For an array section or for an array expression other than a whole array or array structure component, LBOUND(ARRAY, DIM) has the value 1. For a whole array or array structure component, LBOUND(ARRAY, DIM) has the value:

- equal to the lower bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have extent zero or if ARRAY is an assumed-size array of rank DIM

or

- one (1), otherwise.

Case 2: LBOUND(ARRAY) has a value whose *i*th component is equal to LBOUND(ARRAY, *i*), for *i*= 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Specific forms KLBOUND.

LEN(STRING)

Description Returns the length of a character entity.

Class Inquiry function.

Argument `STRING` must be of type character. It may be scalar or array valued.

Result type, type parameters, and shape Default integer scalar.

Result value The result has a value equal to the number of characters in `STRING` if it is scalar or in an element of `STRING` if it is array valued.

Specific forms `KLEN`.

LEN_TRIM(STRING)

Description Returns the length of the character argument without counting trailing blank characters.

Class Elemental function.

Argument `STRING` must be of type character.

Result type/ type parameter Default integer.

Result value The result has a value equal to the number of characters remaining after any trailing blanks in `STRING` are removed. If the argument contains no nonblank characters, the result is zero.

Specific forms `KLEN_TRIM`.

LGE(STRING_A, STRING_B)

Description

Tests whether a string is lexically greater than or equal to another string, based on the ASCII collating sequence.

Class

Elemental function.

Arguments

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result type/ type parameters

Default logical.

Result value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor dependent.

The result is `.TRUE.` if the strings are equal or if `STRING_A` follows `STRING_B` in the ASCII collating sequence; otherwise, the result is `.FALSE.` Note that the result is `.TRUE.` if both `STRING_A` and `STRING_B` are of zero length.

LGT(STRING_A, STRING_B)

Description

Tests whether a string is lexically greater than another string, based on the ASCII collating sequence.

Class

Elemental function.

Arguments

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result type/ type parameters

Default logical.

Result value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor-dependent.

The result is .TRUE. if STRING_A follows STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .FALSE. if both STRING_A and STRING_B are of zero length.

LLE(STRING_A, STRING_B)

Description

Tests whether a string is lexically less than or equal to another string, based on the ASCII collating sequence.

Class

Elemental function.

Arguments

STRING_A	must be of type default character.
STRING_B	must be of type default character.

Result type/ type parameters

Default logical.

Result value

If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor dependent.

The result is .TRUE. if the strings are equal or if STRING_A precedes STRING_B in the ASCII collating sequence; otherwise, the result is .FALSE.. Note that the result is .TRUE. if both STRING_A and STRING_B are of zero length.

LLT(STRING_A, STRING_B)

Description Tests whether a string is lexically less than another string, based on the ASCII collating sequence.

Class Elemental function.

Arguments

`STRING_A`	must be of type default character.
`STRING_B`	must be of type default character.

Result type/ type parameters Default logical.

Result value If the strings are of unequal length, the comparison is made as if the shorter string were extended on the right with blanks to the length of the longer string.

If either string contains a character not in the ASCII character set, the result is processor-dependent.

The result is `.TRUE.` if `STRING_A` precedes `STRING_B` in the ASCII collating sequence; otherwise, the result is `.FALSE.`. Note that the result is `.FALSE.` if both `STRING_A` and `STRING_B` are of zero length.

LOC(X)

Description Return the address of the argument.

Class Inquiry nonstandard function.

LOG(X)

Description Natural logarithm.

Class Elemental function.

Argument X must be of type real or complex. If X is real, its value must be greater than zero. If X is complex, its value must not be zero.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to $\log_e x$. A result of type complex is the principal value with imaginary part *w* in the range -Pi < *w* <= Pi. The imaginary part of the result is Pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

Specific forms `ALOG`, `CDLOG`, `CLOG`, `DLOG`, `QLOG`, `ZLOG`.

LOG10(X)

Description Common logarithm.

Class Elemental function.

Argument X must be of type real. The value of X must be greater than zero.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to $\log_{10}$X.

Specific forms `ALOG10`, `DLOG10`, `QLOG10`.

LOGICAL(L, KIND)

Optional argument `KIND`

Description Converts between kinds of logical.

Class Elemental function.

Arguments

`L`	must be of type logical.
`KIND` (optional)	must be a scalar integer initialization expression.

Result type/ type parameters Logical. If `KIND` is present, the kind type parameter is that specified by `KIND`; otherwise, the kind type parameter is that of default logical.

Result value The value is that of `L`.

LSHFT(I, SHIFT)

Description Left shift.

Class Elemental nonstandard function.

LSHIFT(I, SHIFT)

Description Left shift.

Class Elemental nonstandard function.

MALLOC(SIZE)

Description Allocate a block of memory.

Class Transformational nonstandard function.

MATMUL(MATRIX_A, MATRIX_B)

Description

Performs matrix multiplication of numeric or logical matrices.

Class

Transformational function.

Arguments

`MATRIX_A`	must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.
`MATRIX_B`	must be of numeric type if `MATRIX_A` is of numeric type and of logical type if `MATRIX_A` is of logical type. It must be array valued and of rank one or two. If `MATRIX_A` has rank one, `MATRIX_B` must have rank two. If `MATRIX_B` has rank one, `MATRIX_A` must have rank two. The size of the first (or only) dimension of `MATRIX_B` must equal the size of the last (or only) dimension of `MATRIX_A`.

Result type, type parameters, and shape

If the arguments are of numeric type, the type and kind type parameter of the result are determined by the types of `MATRIX_A` and `MATRIX_B`.

If the arguments are of type logical, the result is of type logical with the kind type parameter of the arguments.

The shape of the result depends on the shapes of the arguments as follows:

Case 1	If `MATRIX_A` has shape [*n*, *m*] and `MATRIX_B` has shape [*m*, *k*], the result has shape [*n*, *k*].
Case 2	If `MATRIX_A` has shape [*m*] and `MATRIX_B` has shape [*m*, *k*], the result has shape [*k*].
Case 3	If `MATRIX_A` has shape [*n*, *m*] and `MATRIX_B` has shape [*m*], the result has shape [*n*].

Result value

Case 1	Element (*i*, *j*) of the result has the value `SUM(MATRIX_A(`*i*`,:) * MATRIX_B(:, `*j*`))` if the arguments are of numeric type and has the value `ANY(MATRIX_A(`*i*`, :) .AND. MATRIX_B(:, `*j*`))` if the arguments are of logical type.

Case 2	Element (*j*) of the result has the value `SUM(MATRIX_A(:) * MATRIX_B(:, j))` if the arguments are of numeric type and has the value `ANY(MATRIX_A(:) .AND. MATRIX_B(:, j))` if the arguments are of logical type.
Case 3	Element (*i*) of the result has the value `SUM(MATRIX_A(i, :) * MATRIX_B(:))` if the arguments are of numeric type and has the value `ANY(MATRIX_A(i, :) .AND. MATRIX_B(:))` if the arguments are of logical type.

MAX(A1, A2, A3, ...)

Optional arguments
A3, ...

Description
Maximum value.

Class
Elemental function.

Arguments
The arguments must all have the same type which must be integer or real, and they must all have the same kind type parameter.

Result type/ type parameters
Same as the arguments.

Result value
The value of the result is that of the largest argument.

Specific forms
`AIMAX0`, `AJMAX0`, `AKMAX0`, `AMAX0`, `AMAX1`, `DMAX1`, `IMAX0`, `IMAX1`, `JMAX0`, `JMAX1`, `KMAX0`, `KMAX1`, `MAX0`, `MAX1`, `QMAX1`.

MAXEXPONENT(X)

Description Returns the maximum exponent in the model representing numbers of the same type and kind type parameter as the argument.

Class Inquiry function.

Argument X must be of type real. It may be scalar or array valued.

Result type, type parameters, and shape Default integer scalar.

Result value The result has the value e_{max}, as defined in “The Real Number System Model” on page 482.

Specific forms KMAXLOC.

MAXLOC(ARRAY, MASK)

Optional argument MASK

Description Returns the location of the first element of ARRAY having the maximum value of the elements identified by MASK.

Class Transformational function.

Arguments

ARRAY	must be of type integer or real. It must not be scalar.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result type, type parameters, and shape The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result value

Case 1: If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the maximum value of all of the elements of ARRAY.

The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY.

If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor-dependent.

Case 2 — If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a .TRUE. element of MASK, whose value equals the maximum value of all such elements of ARRAY.

The ith subscript returned lies in the range 1 to e_i, where e_i is the extent of the ith dimension of ARRAY.

If more than one such element has the maximum value, the element whose subscripts are returned is the first such element taken in array element order.

If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value .FALSE.), the value of the result is processor-dependent.

In both cases, an element of the result is undefined if the processor cannot represent the value as a default integer.

MAXVAL(ARRAY, DIM, MASK)

Optional arguments — DIM, MASK

Description — Maximum value of the elements of ARRAY along dimension DIM that correspond to the .TRUE. elements of MASK.

Class — Transformational function.

Arguments

ARRAY — must be of type integer or real. It must not be scalar.

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.

MASK (optional) — must be of type logical and must be conformable with ARRAY.

Result type, type parameters, and shape

The result is of the same type and kind type parameter as ARRAY.

It is scalar if `DIM` is absent or ARRAY has rank one; otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result value

Case 1: The result of `MAXVAL(ARRAY)` has a value equal to the maximum value of all the elements of ARRAY or has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case 2: The result of `MAXVAL(ARRAY, MASK = MASK)` has a value equal to the maximum value of the elements of ARRAY corresponding to `.TRUE.` elements of MASK or has the value of the negative number of the largest magnitude supported by the processor for numbers of the same type and kind type parameter as ARRAY if there are no `.TRUE.` elements.

Case 3: If ARRAY has rank one, `MAXVAL(ARRAY, DIM [,MASK])` has a value equal to that of `MAXVAL(ARRAY [,MASK = MASK])`. Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of `MAXVAL(ARRAY, DIM [,MASK])` is equal to the following:

`MAXVAL(ARRAY(`s_1`, `s_2`, ..., `s_{DIM-1}`, :, `s_{DIM+1}`, ..., `s_n`) [, MASK = MASK(`s_1`, `s_2`, ..., `s_{DIM-1}`, :, `s_{DIM+1}`, ..., `s_n`) ] )`

MCLOCK()

Description

Return time accounting for a program.

Class

Inquiry nonstandard function.

Result type

Integer.

Result value

The value returned, in units of microseconds, is the sum of the current process's user time and the user and system time of all its child processes.

MERGE(TSOURCE, FSOURCE, MASK)

Description Choose alternative value according to the value of a mask.

Class Elemental function.

Arguments

TSOURCE	may be of any type.
FSOURCE	must be of the same type and type parameters as TSOURCE.
MASK	must be of type logical.

Result type/ type parameters Same as TSOURCE.

Result value The result is TSOURCE if MASK is .TRUE. and FSOURCE otherwise.

MIN(A1, A2, A3, ...)

Optional arguments A3, ...

Description Minimum value.

Class Elemental function.

Arguments The arguments must all be of the same type, which must be integer or real, and they must all have the same kind type parameter.

Result type/ type parameters Same as the arguments.

Result value The value of the result is that of the smallest argument.

Specific forms AIMIN0, AJMIN0, AKMIN0, AMIN0, AMIN1, DMIN1, IMIN0, IMIN1, JMIN0, JMIN1, KMIN0, KMIN1, MIN0, MIN1, QMIN1.

MINEXPONENT(X)

Description
Returns the minimum exponent in the model representing numbers of the same type and kind type parameter as the argument.

Class
Inquiry function.

Argument
X must be of type real. It may be scalar or array valued.

Result type, type parameters, and shape
Default integer scalar.

Result value
The result has the value e_{min}, as defined in "The Real Number System Model" on page 482.

MINLOC(ARRAY, MASK)

Optional argument
MASK

Description
Returns the location of the first element of ARRAY having the minimum value of the elements identified by MASK.

Class
Transformational function.

Arguments

ARRAY	must be of type integer or real. It must not be scalar.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result type, type parameters, and shape
The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result value

Case 1: If MASK is absent, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY whose value equals the minimum value of all the elements of ARRAY.

The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY.

If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If ARRAY has size zero, the value of the result is processor-dependent.

Case 2 — If MASK is present, the result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a .TRUE. element of MASK, whose value equals the minimum value of all such elements of ARRAY.

The *i*th subscript returned lies in the range 1 to e_i, where e_i is the extent of the *i*th dimension of ARRAY. If more than one such element has the minimum value, the element whose subscripts are returned is the first such element taken in array element order.

If ARRAY has size zero or every element of MASK has the value .FALSE., the value of the result is processor-dependent.

In both cases, an element of the result is undefined if the processor cannot represent the value as a default integer.

Specific forms KMINLOC.

MINVAL(ARRAY, DIM, MASK)

Optional argument DIM, MASK

Description Minimum value of all the elements of ARRAY along dimension DIM corresponding to .TRUE. elements of MASK.

Class Transformational function.

Arguments

ARRAY	must be of type integer or real. It must not be scalar.
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.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result type, type parameters, and shape The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., $d_{\text{DIM-1}}$, $d_{\text{DIM+1}}$, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result value

Case 1	The result of MINVAL(ARRAY) has a value equal to the minimum value of all the elements of ARRAY or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.
Case 2	The result of MINVAL(ARRAY, MASK = MASK) has a value equal to the minimum value of the elements of ARRAY corresponding to .TRUE. elements of MASK or has the value of the positive number of the largest magnitude supported by the processor for numbers of the same type and kind type parameter as ARRAY if there are no .TRUE. elements.
Case 3	If ARRAY has rank one, MINVAL(ARRAY, DIM [,MASK]) has a value equal to that of MINVAL(ARRAY [,MASK = MASK]). Otherwise, the value of element (s_1, s_2, ..., $s_{\text{DIM-1}}$, $s_{\text{DIM+1}}$, ..., s_n) of MINVAL(ARRAY, DIM [,MASK]) is equal to the following:

`MINVAL(ARRAY(`s_1`, `s_2`, ..., `s_{DIM-1}`, :, `s_{DIM+1}`, ..., `s_n`) [, MASK= MASK(`s_1`, `s_2`, ..., `s_{DIM-1}`, :, `s_{DIM+1}`, ..., `s_n`) ] )`

MOD(A, P)

Description Remainder function.

Class Elemental function.

Arguments

A	must be of type integer or real.
P	must be of the same type and kind type parameter as A.

Result type/ type parameters Same as A.

Result value If P is not 0, the value of the result is `A - INT(A/P) * P`. If P=0, the result is processor-dependent.

Specific forms AMOD, BMOD, DMOD, HMOD, IMOD, JMOD, KMOD, QMOD.

MODULO(A, P)

Description Modulo function.

Class Elemental function.

Arguments

A	must be of type integer or real.
P	must be of the same type and kind type parameter as A.

Result type/ type parameters Same as A.

Result value

Case 1: A is of type integer. If P is not 0, `MODULO(A, P)` has the value R such that `A = Q * P + R`, where Q is an integer, the inequalities 0 <= R<P hold if P>0, and P<R <= 0 hold if P<0. If P=0, the result is processor-dependent.

Case 2	`A` is of type real. If `P` is not 0, the value of the result is `A -FLOOR (A / P) * P`. If `P`=0, the result is processor-dependent.

MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)

Description Copies a sequence of bits from one data object to another.

Class Elemental subroutine.

Arguments

`FROM` must be of type integer. It is an `INTENT(IN)` argument.

`FROMPOS` must be of type integer and nonnegative. It is an `INTENT(IN)` argument. `FROMPOS` + `LEN` must be less than or equal to `BIT_SIZE(FROM)`. The model for the interpretation of an integer value as a sequence of bits is described in “The Bit Model” on page 481.

`LEN` must be of type integer and nonnegative. It is an `INTENT(IN)` argument.

`TO` must be a variable of type integer with the same kind type parameter value as `FROM` and may be the same variable as `FROM`. It is an `INTENT(INOUT)` argument.

`TO` is set by copying the sequence of bits of length `LEN`, starting at position `FROMPOS` of `FROM` to position `TOPOS` of `TO`. No other bits of `TO` are altered. On return, the `LEN` bits of `TO` starting at `TOPOS` are equal to the value that the `LEN` bits of `FROM` starting at `FROMPOS` had on entry.

The model for the interpretation of an integer value as a sequence of bits is described in “The Bit Model” on page 481.

`TOPOS` must be of type integer and nonnegative. It is an `INTENT(IN)` argument. `TOPOS` + `LEN` must be less than or equal to `BIT_SIZE(TO)`.

Specific forms `BMVBITS`, `HMVBITS`.

NEAREST(X, S)

Description Returns the nearest different machine representable number in a given direction.

Class Elemental function.

Arguments

X	must be of type real.
S	must be of type real and not equal to zero.

Result type/ type parameters Same as X.

Result value The result has a value equal to the machine representable number distinct from X and nearest to it in the direction of the infinity with the same sign as S.

NINT(A, KIND)

Optional argument KIND

Description Nearest integer.

Class Elemental function.

Arguments

A	must be of type real.
KIND (optional)	must be a scalar integer initialization expression.

Result type/ type parameters Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of default integer type.

Result value If A>0, NINT(A) has the value INT(A+0.5); if A <= 0, NINT(A) has the value INT(A–0.5). The result is undefined if the processor cannot represent the result in the specified integer type.

Specific forms IDNINT, IIDNNT, IIQNNT, ININT, IQNINT, JIDNNT, JIQNNT, JNINT, KIDNNT, KIQNNT, KNINT.

NOT(I)

Description Performs a bitwise logical complement.

Class Elemental function.

Argument `I` must be of type integer.

Result type/ type parameters Same as `I`.

Result value The result has the value obtained by complementing `I` bit-by-bit according to the following truth table:

Table 60 Truth table for NOT intrinsic

I	NOT(I)
1	0
0	1

The model for the interpretation of an integer value as a sequence of bits is described in “The Bit Model” on page 481.

Specific forms `BNOT`, `HNOT`, `INOT`, `JNOT`, `KNOT`.

OR(I, J)

Description Bitwise logical OR.

Class Elemental nonstandard function.

Arguments

`I` must be of type integer.

`J` must be of type integer with the same kind type parameter as `I`.

Result type/ type parameters Same as `I`.

Result value

The result has the value obtained by performing an OR on `I` and `J` bit-by-bit according to the following truth table:

Table 61 **Truth table for OR intrinsic**

I	J	OR(I, J)
1	1	1
1	0	1
0	1	1
0	0	0

The model for interpreting an integer value as a sequence of bits is described in “The Bit Model” on page 481.

PACK(ARRAY, MASK, VECTOR)

Optional argument

`VECTOR`

Description

Pack an array into an array of rank one under the control of a mask.

Class

Transformational function.

Arguments

`ARRAY` may be of any type. It must not be scalar.

`MASK` must be of type logical and must be conformable with `ARRAY`.

`VECTOR` (optional) must be of the same type and type parameters as `ARRAY` and must have rank one. `VECTOR` must have at least as many elements as there are `.TRUE.` elements in `MASK`. If `MASK` is scalar with the value `.TRUE.`, `VECTOR` must have at least as many elements as there are in `ARRAY`.

Result type, type parameters, and shape

The result is an array of rank one with the same type and type parameters as `ARRAY`. If `VECTOR` is present, the result size is that of `VECTOR`; otherwise, the result size is the number *t* of `.TRUE.` elements in `MASK` unless `MASK` is scalar with the value `.TRUE.`, in which case the result size is the size of `ARRAY`.

Result value

Element *i* of the result is the element of ARRAY that corresponds to the *i*th .TRUE. element of MASK, taking elements in array element order, for *i*= 1, 2, ..., *t*. If VECTOR is present and has size *n*>*t*, element *i* of the result has the value VECTOR (*i*), for *i*= *t*+1, ..., *n*.

Specific forms

KPACK.

PRECISION(X)

Description

Returns the decimal precision in the model representing real numbers with the same kind type parameter as the argument.

Class

Inquiry function.

Argument

X must be of type real or complex. It may be scalar or array valued.

Result type, type parameters, and shape

Default integer scalar.

Result value

The result has the value INT((*p*-1) * LOG10(*b*))+k. The values of *b* and *p* are as defined in “The Real Number System Model” on page 482 for the model representing real numbers with the same kind type parameter as X. The value of *k* is 1 if *b* is an integral power of 10 and 0 otherwise.

PRESENT(A)

Description

Determine whether an optional argument is present.

Class

Inquiry function.

Argument

A must be the name of an optional dummy argument that is accessible in the procedure in which the PRESENT function reference appears.

Result type/ type parameters

Default logical scalar.

Result value

The result has the value .TRUE. if A is present and otherwise has the value .FALSE.

PRODUCT(ARRAY, DIM, MASK)

Optional arguments

DIM, MASK

Description

Product of all the elements of ARRAY along dimension DIM corresponding to the .TRUE. elements of MASK.

Class

Transformational function.

Arguments

ARRAY	must be of type integer, real, or complex. It must not be scalar.
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.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result type, type parameters, and shape

The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank *n*-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result value

Case 1	The result of PRODUCT(ARRAY) has a value equal to a processor-dependent approximation to the product of all the elements of ARRAY or has the value one if ARRAY has size zero.
Case 2	The result of PRODUCT(ARRAY, MASK = msk) has a value equal to a processor-dependent approximation to the product of the elements of ARRAY corresponding to the .TRUE. elements of msk or has the value one if there are no .TRUE. elements.
Case 3	If ARRAY has rank one, PRODUCT(ARRAY, DIM [,msk]) has a value equal to that of PRODUCT(ARRAY [,MASK = msk]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of PRODUCT(ARRAY, DIM [,msk]) is equal to the following:

```
PRODUCT(ARRAY(s1, s2, ..., sDIM-1, :, sDIM+1, ..., sn) &
   [, MASK = msk(s1, s2, ..., sDIM-1, :, sDIM+1, ..., sn)])
```

QEXT(A)

Description Convert to `REAL(16)` type.

Class Elemental nonstandard function.

Argument A must be of type integer, real, double precision, or complex.

Result `REAL(16)`.

Specific forms `QEXTD`.

QFLOAT(A)

Description Convert to `REAL(16)` type.

Class Elemental nonstandard function.

Argument A must be of type integer or `REAL(4)`.

Result `REAL(16)`.

Specific forms `QFLOATI`, `QFLOTI`, `QFLOTJ`, `QFLOTK`.

QNUM(I)

Description Convert character to `REAL(16)` type.

Class Elemental nonstandard function.

Argument `I` must be of type character.

Result `REAL(16)` type.

QPROD(X, Y)

Description Double precision product.

Class Elemental nonstandard function.

Arguments X and Y must be of type double precision.

Result REAL(16) type.

RADIX(X)

Description Returns the base of the model representing numbers of the same type and kind type parameter as the argument.

Class Inquiry function.

Argument X must be of type integer or real. It may be scalar or array valued.

Result type, type parameters, and shape Default integer scalar.

Result value The result has the value *r* if X is of type integer and the value *b* if X is of type real, where *r* and *b* are as defined in "The Real Number System Model" on page 482.

RAN(ISEED)

Description Multiplicative congruent random number generator.

Class Elemental nonstandard function.

Argument ISEED must be an INTEGER(4) variable or array element. RAN stores a number in ISEED to be used by the next call to RAN.

ISEED should initially be set to an odd number, preferably very large; see the following example.

Result type/ type parameters REAL(4) type.

NOTE To ensure different random values for each run of a program, ISEED should be set to a different value each time the program is run. One way to implement this would be to have the user enter the seed at the start of the program. Another way would be to compute a value from the current year, day, and month (returned by IDATE) and the number of seconds since midnight (returned by SECNDS).

RAND()

Description Generate successive pseudorandom numbers uniformly distributed in the range of 0.0 to 1.0.

Class Elemental nonstandard function.

Result type/ type parameters `REAL(4)` type.

NOTE For details about restarting the pseudorandom number generator used by `IRAND` and `RAND`, see “SRAND(ISEED)” on page 577 section.

RANDOM_NUMBER(HARVEST)

Description Returns one pseudorandom number or an array of pseudorandom numbers from the uniform distribution over the range $0 <= x < 1$.

Class Subroutine.

Argument `HARVEST` must be of type real. It is an `INTENT(OUT)` argument. It may be a scalar or an array variable. It is set to contain pseudorandom numbers from the uniform distribution in the interval $0 <= x < 1$.

RANDOM_SEED(SIZE, PUT, GET)

Optional arguments `SIZE`, `PUT`, `GET`

Description Restarts or queries the pseudorandom number generator used by `RANDOM_NUMBER`.

Class Subroutine.

Arguments There must either be exactly one or no arguments present.

`SIZE` (optional)	must be scalar and of type default integer. It is an `INTENT(OUT)` argument. It is set to the number *N* of integers that the processor uses to hold the value of the seed.
`PUT` (optional)	must be a default integer array of rank one and size >= *N*. It is an `INTENT(IN)` argument. It is used by the processor to set the seed value.
`GET` (optional)	must be a default integer array of rank one and size >= *N*. It is an `INTENT(OUT)` argument. It is set by the processor to the current value of the seed. If no argument is present, the processor sets the seed to a processor-dependent value.

RANGE(X)

Description

Returns the decimal exponent range in the model representing integer or real numbers with the same kind type parameter as the argument.

Class

Inquiry function.

Argument

`X` must be of type integer, real, or complex. It may be scalar or array valued.

Result type, type parameters, and shape

Default integer scalar.

Result value

Case 1	For an integer argument, the result has the value `INT(LOG10(`*huge*`))`, where *huge* is the largest positive integer in the model representing integer numbers with same kind type parameter as `X`. See “The Integer Number System Model” on page 482 for more information.
Case 2	For a real or complex argument, the result has the value `INT(MIN(LOG10(`*huge*`), -LOG10(`*tiny*`)))`, where *huge* and *tiny* are the largest and smallest positive numbers in the model representing real numbers with the same value for the kind type parameter as `X`. See “The Real Number System Model” on page 482 for more information.

Example `RANGE(X)` has the value 38 for real `X`, whose model is described in “The Real Number System Model” on page 482, because in this case *huge*= $(1 - 2^{-24}) * 2^{127}$ and *tiny* = 2^{-127}.

Specific forms `SNGL`, `SNGLQ`.

REAL(A, KIND)

Optional argument `KIND`

Description Convert to real type.

Class Elemental function.

Arguments

`A`	must be of type integer, real, or complex.
`KIND` (optional)	must be a scalar integer initialization expression.

Result type/ type parameters Real.

Case 1	If `A` is of type integer or real and `KIND` is present, the kind type parameter is that specified by `KIND`.
	If `A` is of type integer or real and `KIND` is not present, the kind type parameter is the processor-dependent kind type parameter for the default real type.
Case 2	If `A` is of type complex and `KIND` is present, the kind type parameter is that specified by `KIND`.
	If `A` is of type complex and `KIND` is not present, the kind type parameter is the kind type parameter of `A`.

Result value

Case 1	If `A` is of type integer or real, the result is equal to a processor-dependent approximation to `A`.
Case 2	If `A` is of type complex, the result is equal to a processor-dependent approximation to the real part of `A`.

Specific forms `FLOAT`, `FLOATI`, `FLOATJ`, `FLOATK`.

REPEAT(STRING, NCOPIES)

Description	Concatenate several copies of a string.	
Class	Transformational function.	
Arguments	`STRING`	must be scalar and of type character.
	`NCOPIES`	must be scalar and of type integer. Its value must not be negative.
Result type, type parameters, and shape	Character scalar of length `NCOPIES` times that of `STRING`, with the same kind type parameter as `STRING`.	
Result value	The value of the result is the concatenation of `NCOPIES` copies of `STRING`.	
Specific forms	`KREPEAT`.	

RESHAPE(SOURCE, SHAPE, PAD, ORDER)

Optional arguments	`PAD`, `ORDER`	
Description	Constructs an array of a specified shape from the elements of a given array.	
Class	Transformational function.	
Arguments	`SOURCE`	may be of any type. It must be array valued. If `PAD` is absent or of size zero, the size of `SOURCE` must be greater than or equal to `PRODUCT(SHAPE)`. The size of the result is the product of the values of the elements of `SHAPE`.
	`SHAPE`	must be of type integer, rank one, and constant size. Its size must be positive and less than 8. It must not have an element whose value is negative.
	`PAD` (optional)	must be of the same type and type parameters as `SOURCE`. `PAD` must be array valued.

ORDER (optional) must be of type integer, must have the same shape as SHAPE, and its value must be a permutation of [1, 2, ..., n], where n is the size of SHAPE. If absent, it is as if it were present with value [1, 2, ..., n].

Result type, type parameters, and shape

The result is an array of shape SHAPE (that is, SHAPE(RESHAPE(SOURCE, SHAPE, PAD, ORDER)) is equal to SHAPE) with the same type and type parameters as SOURCE.

Result value

The elements of the result, taken in permuted subscript order ORDER(1), ..., ORDER(n), are those of SOURCE in normal array element order followed if necessary by those of PAD in array element order, followed if necessary by additional copies of PAD in array element order.

Specific forms

KRESHAPE.

RNUM(I)

Description

Convert character to real type.

Class

Elemental nonstandard function.

Argument

I must be of type character.

Result

Default real type.

RRSPACING(X)

Description

Returns the reciprocal of the relative spacing of model numbers near the argument value.

Class

Elemental function.

Argument

X must be of type real.

Result type/ type parameters

Same as X.

Result value

The result has the value $|X * b^{-e}| * b^{p}$, where b, e, and p are as defined in “The Real Number System Model” on page 482.

RSHFT(I, SHIFT)

Description Bitwise right shift.

Class Elemental nonstandard function.

RSHIFT(I, SHIFT)

Description Bitwise right shift.

Class Elemental nonstandard function.

SCALE(X, I)

Description Returns X * b^{I} where *b* is the base in the model representation of X. See “The Real Number System Model” on page 482 for a description of this.

Class Elemental function.

Arguments

`X`	must be of type real.
`I`	must be of type integer.

Result type/ type parameters Same as X.

Result value The result has the value X * b^{I}, where *b* is defined in “The Real Number System Model” on page 482, provided this result is within range; if not, the result is processor dependent.

SCAN(STRING, SET, BACK)

Optional argument BACK

Description Scan a string for any one of the characters in a set of characters.

Class Elemental function.

Arguments

`STRING`	must be of type character.

	SET	must be of type character with the same kind type parameter as STRING.
	BACK (optional)	must be of type logical.
Result type/ type parameters	Default integer.	
Result value	Case 1	If BACK is absent or is present with the value .FALSE. and if STRING contains at least one character that is in SET, the value of the result is the position of the leftmost character of STRING that is in SET.
	Case 2	If BACK is present with the value .TRUE. and if STRING contains at least one character that is in SET, the value of the result is the position of the rightmost character of STRING that is in SET.
	Case 3	The value of the result is zero if no character of STRING is in SET or if the length of STRING or SET is zero.

SECNDS(X)

Description

Return the number of seconds that have elapsed since midnight, less the value of the argument.

Class

Elemental nonstandard function.

Argument

X must be of type REAL(4).

Result type/ type parameters

REAL(4).

NOTE

SECNDS is accurate to one one-hundredth of a second (0.01 second). The SECNDS routine is useful for computing elapsed time for a code’s execution.

SELECTED_INT_KIND(R)

Description

Returns a value of the kind type parameter of an integer data type that represents all integer values *n* with $-10^R < n < 10^R$.

Class

Transformational function.

Argument

R must be scalar and of type integer.

Result type, type parameters, and shape

Default integer scalar.

Result value

The result has a value equal to the value of the kind type parameter of an integer data type that represents all values *n* in the range of values *n* with $-10^R < n < 10^R$, or if no such kind type parameter is available on the processor, the result is –1.

If more than one kind type parameter meets the criteria, the value returned is the one with the smallest decimal exponent range, unless there are several such values, in which case the smallest of these kind values is returned.

SELECTED_REAL_KIND(P, R)

Optional arguments

P, R

Description

Returns a value of the kind type parameter of a real data type with decimal precision of at least P digits and a decimal exponent range of at least R.

Class

Transformational function.

Arguments

At least one argument must be present.

P (optional)	must be scalar and of type integer.
R (optional)	must be scalar and of type integer.

Result type, type parameters, and shape

Default integer scalar.

Result value

The result has a value equal to a value of the kind type parameter of a real data type with decimal precision, as returned by the function PRECISION, of at least P digits and a decimal exponent range, as returned by the function RANGE, of at least R.

If no such kind type parameter is available on the processor, the result is –1 if the precision is not available, –2 if the exponent range is not available, and –3 if neither is available.

If more than one kind type parameter value meets the criteria, the value returned is the one with the smallest decimal precision, unless there are several such values, in which case the smallest of these kind values is returned.

SET_EXPONENT(X, I)

Description

Returns the model number whose exponent is I and whose fractional part is the fractional part of X.

Class

Elemental function.

Arguments

X	must be of type real.
I	must be of type integer.

Result type/ type parameters

Same as X.

Result value

The result has the value X * b^{I-e}, where b and e are as defined in “The Real Number System Model” on page 482, provided this result is within range; if not, the result is processor-dependent.

If X has value zero, the result has value zero.

SHAPE(SOURCE)

Description Returns the shape of an array or a scalar.

Class Inquiry function.

Argument SOURCE may be of any type. It may be array valued or scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. It must not be an assumed-size array.

Result type, type parameters, and shape The result is a default integer array of rank one whose size is equal to the rank of SOURCE.

Result value The value of the result is the shape of SOURCE.

Specific forms KSHAPE.

SIGN(A, B)

Description Absolute value of A times the sign of B.

Class Elemental function.

Arguments

A	must be of type integer or real.
B	must be of the same type and kind type parameter as A.

Result type/ type parameters Same as A.

Result value The value of the result is |A| if B>= 0 and –|A| if B < 0.

Specific forms BSIGN, DSIGN, HSIGN, IISIGN, JSIGN, QSIGN, KISIGN.

SIN(X)

Description	Sine function in radians.
Class	Elemental function.
Argument	X must be of type real or complex.
Result type/ type parameters	Same as X.
Result value	The result has a value equal to a processor-dependent approximation to sin(X). • If X is of type real, it is regarded as a value in radians. • If X is of type complex, its real part is regarded as a value in radians.
Specific forms	`CDSIN`, `CSIN`, `DSIN`, `QSIN`, `ZSIN`.

SIND(X)

Description	Sine function in degrees.
Class	Elemental nonstandard function.
Argument	X must be of type real.
Result type/ type parameters	Same as X.
Result value	The result has a value equal to a processor-dependent approximation to sin(X).
Specific forms	`QSIND`.

SINH(X)

Description Hyperbolic sine function.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to sinh(X).

Specific forms `QSINH`.

SIZE(ARRAY, DIM)

Optional argument `DIM`

Description Returns the extent of an array along a specified dimension or the total number of elements in the array.

Class Inquiry function.

Arguments

ARRAY	may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, `DIM` must be present with a value less than the rank of ARRAY.
`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.

Result type, type parameters, and shape Default integer scalar.

Result value The result has a value equal to the extent of dimension `DIM` of ARRAY or, if `DIM` is absent, the total number of elements of ARRAY.

Specific forms `KSIZE.`

SIZEOF(A)

Description Return the number of bytes of storage used by the argument.

Class Inquiry nonstandard function.

Argument A may be of any type (except assumed-size arrays or passed-length character arguments).

Result type Integer.

SPACING(X)

Description Returns the absolute spacing of model numbers near the argument value.

Class Elemental function.

Argument `X` must be of type real.

Result type/ type parameters Same as `X`.

Result value If `X` is not zero, the result has the value b^{e-p}, where *b*, *e*, and *p* are as defined in "The Real Number System Model" on page 482, provided this result is within range; otherwise, the result is the same as that of `TINY(X)`.

SPREAD(SOURCE, DIM, NCOPIES)

Description Replicates an array by adding a dimension. Broadcasts several copies of `SOURCE` along a specified dimension (as in forming a book from copies of a single page) and thus forms an array of rank one greater.

Class Transformational function.

Arguments

`SOURCE`	may be of any type. It may be scalar or array valued. The rank of `SOURCE` must be less than 7.

	`DIM`	must be scalar and of type integer with value in the range 1 <= `DIM` <= n + 1, where n is the rank of `SOURCE`.
	`NCOPIES`	must be scalar and of type integer.

Result type, type parameters, and shape

The result is an array of the same type and type parameters as `SOURCE` and of rank n + 1, where n is the rank of `SOURCE`.

Case 1	If `SOURCE` is scalar, the shape of the result is (`MAX(NCOPIES, 0)`).
Case 2	If `SOURCE` is array valued with shape (d_1, d_2, ..., d_n), the shape of the result is (d_1, d_2, ..., d_{DIM-1}, `MAX(NCOPIES, 0)`, d_{DIM}, ..., d_n).

Result value

Case 1	If `SOURCE` is scalar, each element of the result has a value equal to `SOURCE`.
Case 2	If `SOURCE` is array valued, the element of the result with subscripts (r_1, r_2, ..., r_{n+1}) has the value `SOURCE`(r_1, r_2, ..., r_{DIM-1}, r_{DIM+1}, ..., r_{n+1}).

SQRT(X)

Description

Square root.

Class

Elemental function.

Argument

`X` must be of type real or complex. If `X` is real, its value must be greater than or equal to zero.

Result type/ type parameters

Same as `X`.

Result value

The result has a value equal to a processor-dependent approximation to the square root of `X`.

A result of type complex is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

Specific forms

`CDSQRT`, `CSQRT`, `DSQRT`, `QSQRT`, `ZSQRT`.

SRAND(ISEED)

Description Restart the pseudorandom number generator used by `IRAND` and `RAND`.

Class Elemental nonstandard subroutine.

Argument `ISEED` must be of `INTEGER(4)` type.

The same value for `ISEED` generates the same sequence of random numbers. To vary the sequence, call `SRAND` with a different `ISEED` value each time the program is executed. The default for `ISEED` is 1.

SUM(ARRAY, DIM, MASK)

Optional arguments `DIM`, `MASK`

Description Sum all the elements of `ARRAY` along dimension `DIM` corresponding to the `.TRUE.` elements of `MASK`.

Class Transformational function.

Arguments

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`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.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result type, type parameters, and shape The result is of the same type and kind type parameter as `ARRAY`. It is scalar if `DIM` is absent of `ARRAY` has rank one; otherwise, the result is an array of rank n-1 and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of `ARRAY`.

Result value

Case 1	The result of `SUM(ARRAY)` has a value equal to a processor-dependent approximation to the sum of all the elements of `ARRAY` or has the value zero if `ARRAY` has size zero.

Case 2 — The result of `SUM(ARRAY, MASK = msk)` has a value equal to a processor-dependent approximation to the sum of the elements of `ARRAY` corresponding to the `.TRUE.` elements of `msk` or has the value zero if there are no `.TRUE.` elements.

Case 3 — If `ARRAY` has rank one, `SUM(ARRAY, DIM [,msk])` has a value equal to that of `SUM(ARRAY [,MASK = msk])`. Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of `SUM(ARRAY, DIM [,msk])` is equal to the following:

SUM(ARRAY(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) &
[, MASK=msk(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)])

SYSTEM(STR)

Description Issue a shell command from a Fortran 90 program.

Class Nonstandard subroutine.

Argument `STR` must be of type character. `SYSTEM` gives `STR` to the default shell (`/bin/sh`) as input, as if the string were entered at a terminal. When the shell has completed, the process continues.

SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

Optional arguments COUNT, COUNT_RATE, COUNT_MAX

Description Returns integer data from a real-time clock.

Class Subroutine.

Arguments

COUNT (optional)	must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to a processor-dependent value based on the current value of the processor clock or to –HUGE(0) if there is no clock. The processor-dependent value is incremented by one for each clock count until the value COUNT_MAX is reached and is reset to zero at the next count. It lies in the range 0 to COUNT_MAX if there is a clock.
COUNT_RATE (optional)	must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to the number of processor clock counts per second, or to zero if there is no clock.
COUNT_MAX (optional)	must be scalar and of type default integer. It is an INTENT(OUT) argument. It is set to the maximum value that COUNT can have, or to zero if there is no clock.

TAN(X)

Description Tangent function in radians.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to tan(X), with X regarded as a value in radians.

Specific forms `CTAN`,`DTAN`,`QTAN`,`ZTAN`.

TAND(X)

Description Tangent function in degrees.

Class Elemental nonstandard function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to tan(X).

Specific forms `DTAND`, `QTAND`.

TANH(X)

Description Hyperbolic tangent function.

Class Elemental function.

Argument X must be of type real.

Result type/ type parameters Same as X.

Result value The result has a value equal to a processor-dependent approximation to tanh(X).

Specific forms `DTANH`, `QTANH`.

TIME(TIMESTR)

Description

Return the current system time.

Class

Nonstandard subroutine.

Argument

`TIMESTR` must be of type character and must provide at least 8 bytes of storage.

Result value

`TIME` fills `TIMESTR` with an 8-byte character string of the form *hh*`:`*mm*`:`*ss* (*hh* is the current hour, *mm* the current minute, *ss* the number of seconds past the minute).

TINY(X)

Description

Returns the smallest positive number in the model representing numbers of the same type and kind type parameter as the argument.

Class

Inquiry function.

Argument

`X` must be of type real. It may be scalar or array valued.

Result type, type parameters, and shape

Scalar with the same type and kind type parameter as `X`.

Result value

The result has the value $b^{e_{min}-1}$

where b and e_{min} are as defined in "The Real Number System Model" on page 482.

TRANSFER(SOURCE, MOLD, SIZE)

Optional argument `SIZE`

Description Returns a result with a physical representation identical to that of `SOURCE` but interpreted with the type and type parameters of `MOLD`.

Class Transformational function.

Arguments

`SOURCE`	may be of any type and may be scalar or array valued.
`MOLD`	may be of any type and may be scalar or array valued.
`SIZE` (optional)	must be scalar and of type integer. The corresponding actual argument must not be an optional dummy argument.

Result type, type parameters, and shape The result is of the same type and type parameters as `MOLD`.

Case 1	If `MOLD` is a scalar and `SIZE` is absent, the result is a scalar.
Case 2	If `MOLD` is array valued and `SIZE` is absent, the result is array valued and of rank one. Its size is as small as possible such that its physical representation is not shorter than that of `SOURCE`.
Case 3	If `SIZE` is present, the result is array valued of rank one and size `SIZE`.

Result value If the physical representation of the result has the same length as that of `SOURCE`, the physical representation of the result is that of `SOURCE`.

- If the physical representation of the result is longer than that of `SOURCE`, the physical representation of the leading part is that of `SOURCE` and the remainder is undefined.
- If the physical representation of the result is shorter than that of `SOURCE`, the physical representation of the result is the leading part of `SOURCE`. If `D` and `E` are scalar variables such that the physical representation of `D` is as long as or longer than that of `E`, the value of `TRANSFER(TRANSFER(E, D), E)` must be the value of `E`.
- If `D` is an array and `E` is an array of rank one, the value of `TRANSFER(TRANSFER(E, D), E, SIZE(E))` must be the value of `E`.

TRANSPOSE(MATRIX)

Description Transpose an array of rank two.

Class Transformational function.

Result type, type parameters, and shape `MATRIX` may be of any type and must have rank two.

The result is an array of the same type and type parameters as `MATRIX` and with rank two and shape *(n, m)* where *(m, n)* is the shape of `MATRIX`.

Result value Element *(i, j)* of the result has the value `MATRIX`(*j*, *i*), *i*= 1, 2, ..., *n*; *j*= 1, 2, ..., *m*.

TRIM(STRING)

Description Returns the argument with trailing blank characters removed.

Class Transformational function.

Argument `STRING` must be of type character and must be a scalar.

Result type/ type parameters Character with the same kind type parameter value as `STRING` and with a length that is the length of `STRING` less the number of trailing blanks in `STRING`.

Result value The value of the result is the same as `STRING` except any trailing blanks are removed. If `STRING` contains no nonblank characters, the result has zero length.

UBOUND(ARRAY, DIM)

Optional argument `DIM`

Description Returns all the upper bounds of an array or a specified upper bound.

Class Inquiry function.

Arguments

ARRAY	may be of any type. It must not be scalar. It must not be a pointer that is disassociated or an allocatable array that is not allocated. If ARRAY is an assumed-size array, DIM must be present with a value less than the rank of ARRAY.
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.

Result type, type parameters, and shape

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.

Result value

Case 1	For an array section or for an array expression, other than a whole array or array structure component, 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 ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.
Case 2	UBOUND(ARRAY) has a value whose *i*th component is equal to UBOUND(ARRAY, *i*), for *i*= 1, 2, ..., *n*, where *n* is the rank of ARRAY.

Specific forms

KUBOUND.

UNPACK(VECTOR, MASK, FIELD)

Description

Unpack an array of rank one into an array under the control of a mask.

Class

Transformational function.

Arguments

VECTOR	may be of any type. It must have rank one. Its size must be at least *t* where *t* is the number of .TRUE. elements in MASK.
MASK	must be array valued and of type logical.
FIELD	must be of the same type and type parameters as VECTOR and must be conformable with MASK.

Result type, type parameters, and shape

The result is an array of the same type and type parameters as `VECTOR` and the same shape as `MASK`.

Result value

The element of the result that corresponds to the *i*th `.TRUE.` element of `MASK`, in array element order, has the value `VECTOR(`*i*`)` for *i*=1, 2, ..., *t*, where *t* is the number of `.TRUE.` values in `MASK`. Each other element has a value equal to `FIELD` if `FIELD` is scalar or to the corresponding element of `FIELD` if it is an array.

VERIFY(STRING, SET, BACK)

Optional argument

`BACK`

Description

Verify that a set of characters contains all the characters in a string by identifying the position of the first character in a string of characters that does not appear in a given set of characters.

Class

Elemental function.

Arguments

`STRING`	must be of type character.
`SET`	must be of type character with the same kind type parameter as `STRING`.
`BACK` (optional)	must be of type logical.

Result type/ type parameters

Default integer.

Result value

Case 1	If `BACK` is absent or present with the value `.FALSE.` and if `STRING` contains at least one character that is not in `SET`, the value of the result is the position of the leftmost character of `STRING` that is not in `SET`.
Case 2	If `BACK` is present with the value `.TRUE.` and if `STRING` contains at least one character that is not in `SET`, the value of the result is the position of the rightmost character of `STRING` that is not in `SET`.
Case 3	The value of the result is zero if each character in `STRING` is in `SET` or if `STRING` has zero length.

XOR(I, J)

Description Bitwise exclusive OR.

Class Elemental nonstandard function.

Arguments

I	must be of type integer.
J	must be of type integer with the same kind type parameter as I.

Result type/ type parameters Same as I.

Result value The result has the value obtained by performing an exclusive OR on I and J bit-by-bit according to Table 59.

The model for interpreting an integer value as a sequence of bits is described in “The Bit Model” on page 481.

ZEXT(A)

Description Zero extend.

Class Elemental nonstandard function.

Argument A must be of type integer or logical.

Result ZEXT converts a 1-, 2-, or 4-byte logical or integer to a 2- or 4-byte integer by sign-extending zeroes instead of the actual sign bit.

12 BLAS and libU77 libraries

This chapter describes the Basic Linear Algebra Subroutines (BLAS) and the BSD 3f (`libU77`) libraries that are shipped with HP Fortran 90.

The `libU77` library provides routines that have a Fortran 90 interface for system routines in the `libc` library. The `libU77` routines make it easier to call HP-UX system-level routines from Fortran 90 programs because they use Fortran 90 argument-passing conventions. The `libU77` routines are also compatible with other Fortran implementations that supply these routines by default.

NOTE Even though system routines use different argument-passing rules from HP Fortran 90 programs, you can call these routines from HP Fortran 90 programs by using the %VAL and %REF built-in functions to change how arguments are passed. For more information about %VAL and %REF, see “%VAL and %REF built-in functions” on page 148.

The Basic Linear Algebra Subroutine (BLAS) library consists of a set of routines that perform low-level vector and matrix operations. These routines have been tuned for maximum performance and are callable from HP Fortran 90 programs. For information about the background and significance of the BLAS library, refer to the *LAPACK User's Guide*, by E. Anderson *et al* (SIAM Press, 1992).

The following sections considerations to keep in mind when writing and compiling a program that calls routines from the BLAS or `libU77` library, and briefly describes the routines in the libraries. For information about other libraries that are shipped with HP Fortran 90—including how to create and link libraries with your programs—refer to the *HP Fortran 90 Programmer's Guide*.

Calling libU77 and BLAS routines

This section discusses considerations pertinent to writing and compiling programs that call `libU77` and BLAS routines, including:

- The compile-line options that make `libU77` and BLAS routines available to your programs
- Declaring the type of return type of library functions
- Declaring library functions with the `EXTERNAL` attribute
- BLAS and `libU77` man pages

Compile-line options

The following sections describe the compile-line options to use to access routines from the `libU77` and BLAS libraries.

+U77 option

To access `libU77` routines, compile with the `+U77` option. The entry-point name of each `libU77` routine has an appended underscore, which must also be added to the external name of any `libU77` routine that your program calls. The `+U77` option does this. For example, if your program contains the following call:

```
CALL FLUSH(unit_no)
```

compiling with `+U77` causes the compiler to generate the external name `access_`. The `+ppu` and `+uppercase` options have no effect on `libU77` external names.

-lblas option

To access BLAS routines, compile with the `-lblas` option. Unlike most compile-line options, the `-l` option must appear at the end of the command line, following any source files that call BLAS routines; see “General Compiler Syntax” on page 572. Here is an example command line for compiling `do_math.f90` to access BLAS routines:

```
$ f90 do_math.f90 -lblas
```

Year-2000 compatibility

Two new libU77 routines (`DATEY2K` and `IDATEY2K`, both described in this chapter) are provided in the Fortran 90 compiler to handle Year-2000 (Y2K) date-related issues on HP-UX 10.x and HP-UX 11.x. The `+U77` flag must be used with both of these routines.

Although both are provided for Y2K compliance, it is recommended that the standard `DATE_AND_TIME` intrinsic be used instead of these functions, when possible.

The guidelines for changing code which uses the date or idate libU77 routines are as follows:

- In code where date is referenced, replace `DATE` with `DATEY2K`. Also, make sure that `DATEY2K`'s argument is at least 11 characters in length.
- In code where the idate intrinsic (not the `libU77` idate routine) is used, replace `IDATE` with `IDATEY2K`.

Declaring library functions

Unlike intrinsics, library routines do not have an explicit interface within your program. This means (among other things) that, if the routine is a function, the compiler applies the implicit typing rules to the return value. When these rules are in effect, the return value is likely to be meaningless if the type implied by the function name does not agree with the type of the returned value or if the return type is not explicitly declared within the program unit that calls the routine.

Consider the following program, `call_ttynam.f90`. The program consists of two subroutines, both of which call the `libU77` function `TTYNAM`. This function returns a character value—the path name of a terminal device associated. But the return type is declared in only one of the subroutines; in the other subroutine, the type is undeclared, and the compiler therefore assumes—applying the rules of implicit typing—that the return value is of type real. The consequences of this assumption are illustrated in the output, below.

call_ttynam.f90

```
PROGRAM main
! illustrates the consequences of failure to declare
! the return type of a library function. Both
! subroutines do the same thing--invoke the libU77
! function TTYNAM. But only the second subroutine
! declares the return type of the function.
! This program must be compiled with the +U77 option.

  CALL without_decl
  CALL with_decl
END PROGRAM main

SUBROUTINE without_decl
  PRINT *, TTYNAM(6)  ! implicit typing is in effect
END SUBROUTINE without_decl

SUBROUTINE with_decl
  ! declare the return type of TTYNAM
  CHARACTER(LEN=80), EXTERNAL :: TTYNAM

  PRINT *, TTYNAM(6)
END SUBROUTINE with_decl
```

Here are the command lines to compile and execute the program, along with the output from a sample run:

```
$ f90 +U77 call_ttynam.f90
$ a.out
 0.0
 /dev/pts/0
```

For information about explicit interface, see “Procedure interface” on page 151. See “Implicit typing” on page 28 for the rules of implicit typing.

Declaring library routines as EXTERNAL

There are two cases when you should declare a library routine with the EXTERNAL attribute:

- The routine name is passed to a procedure as an actual argument
- The routine name is the same as an intrinsic name

The first case applies to both libU77 and BLAS routines. The second applies only to libU77 routines; as shown in Table 62, several of the names of libU77 routines are also those of intrinsics. Unless you declare these routines with the EXTERNAL attribute, the compiler will map the call to the intrinsic library.

Table 62 **libU77 naming conflicts**

FLUSH	FREE	GETARG
GETENV	IARGC	IDATE
LOC	MALLOC	SYSTEM
TIME		

For example, if a program unit makes a call to FLUSH, the compiler will make a call to the intrinsic, unless the program unit includes the following statement:

```
EXTERNAL FLUSH
```

See “EXTERNAL (statement and attribute)” on page 328 for a description of the EXTERNAL statement and attribute. As noted in the description, the attribute form of EXTERNAL cannot be used with subroutines, which must therefore be specified in the statement form.

Man pages

You can get detailed, online information for any libU77 or BLAS routine by using the man command to display an online reference page for that routine. The command-line syntax for the man command is:

man *section_number routine_name*

where *section_number* is either 3f (for libU77) man pages or 3x (for BLAS); and *routine_name* is the name of the libU77 or BLAS routine. For example, to display the man page for the libU77 routine FLUSH, give the command:

$ **man 3f flush**

To display the man page for the BLAS routine SAXPY, give the command:

$ **man 3x saxpy**

Two of the BLAS man pages provide general information about the BLAS routines: *blas1*(3x) describes basic vector operations, and *blas2*(3x) describes basic matrix operations.

libU77 routines

Table 63 lists the `libU77` routines by category, and Table 64 briefly describes each routine, including signature and argument information. The sizes of the data types listed in Table 64 are the default sizes, unless indicated otherwise. See Table 5 for the sizes of the default data types.

Table 63 Categories of `libU77` routines

Category	`libU77` routines
Date and time	`CTIME`, `DATEY2K`, `DTIME`, `ETIME`, `FDATE`, `GMTIME`, `IDATE`, `IDATEY2K`, `ITIME`, `LTIME`, `TIME`
Error handling	`GERROR`, `IERRNO`, `PERROR`
File system functions	`ACCESS`, `CHDIR`, `CHMOD`, `FSTAT`, `ISATTY`, `LINK`, `LSTAT`, `RENAME`, `STAT`, `SYMLNK`, `TTYNAM`, `UNLINK`
Information retrieval	`GETARG`, `GETCWD`, `GETENV`, `GETGID`, `GETLOG`, `GETPID`, `GETUID`, `HOSTNM`, `IARGC`
Input/Output	`FGETC`, `FPUTC`, `FSEEK`, `FTELL`, `GETC`, `PUTC`
Memory allocation	`FALLOC`, `FREE`, `MALLOC`
Miscellaneous	`LOC`, `QSORT`, `SYSTEM`
Process control	`ALARM`, `FORK`, `KILL`, `SIGNAL`, `SLEEP`, `WAIT`
Tape input/output	`TCLOSE` , `TOPEN`, `TREAD`, `TREWIN`, `TSKIPF`, `TSTATE`, `TWRITE`

Table 64 **libU77 routines**

Name	Description and signature
ACCESS	Determines the accessibility of a file. `INTEGER FUNCTION ACCESS(`*name*`,`*mode*`)` `CHARACTER(LEN=*) ::` *name*`,` *mode*
ALARM	Executes a subroutine after a specified time. `INTEGER FUNCTION ALARM (`*time*`,` *proc*`)` `INTEGER ::` *time* `EXTERNAL` *proc*
CHDIR	Changes the default directory. `INTEGER FUNCTION CHDIR(`*dir_name*`)` `CHARACTER(LEN=*) ::` *dir_name*
CHMOD	Changes the mode of a file. `INTEGER FUNCTION CHMOD (`*name*`,`*mode*`)` `CHARACTER(LEN=*) ::` *name*`,` *mode*
CTIME	Converts a system time to a 24-character ASCII string. `CHARACTER(LEN=*) FUNCTION CTIME (`*stime*`)` `INTEGER ::` *stime*

DATEY2K	Designed to replace the f90 DATE instrinsic. Its function and arguments are the same as the date intrinsic's ecept that the returned string contains a four-digit year in mm-dd-yyyy format instead of a two-digit year in mm-dd--yy format.
	`SUBROUTINE DATEY2K(DATE)`
	`CHARACTER*11 DATE`
	The +U77 flag (described in "+U77 option" on page 588) must be used with DATEY2K.

DTIME	Returns elapsed execution time since the last call to dtime or since the start of execution on the first call.
	`REAL FUNCTION DTIME(`*tarray*`)`
	`REAL ::` *tarray*`(2)`

ETIME	Returns the elapsed execution time, in seconds, for the calling process.
	`REAL FUNCTION ETIME (`*tarray*`)`
	`REAL ::` *tarray* `(2)`

Name	Description and signature
FALLOC	Allocates array space in memory. `SUBROUTINE FALLOC (`*nelem*`,` *elsize*`,` *clean*`,` *basevec*`,` *addr*`,` *offset*`)` `INTEGER ::` *nelem*`,` *elsize*`,` *clean*`,` *addr*`,` *offset* *basevec* must be declared as an array whose elements are *elsize* bytes in size. `FALLOC` allocates space for *basevec* to contain *nelem* elements.

FDATE	Returns the date and time as an ASCII string; available as a subroutine:
	`SUBROUTINE FDATE (`*string*`)`
	`CHARACTER(LEN=*) ::` *string*
	
	And as a function:
	`CHARACTER(LEN=*) :: FUNCTION FDATE()`

FGETC	Retrieves a character from a file specified by an HP Fortran 90 logical unit. `INTEGER FUNCTION FGETC (`*lunit*`,` *char*`)` `INTEGER ::` *lunit* `CHARACTER` *char*
FLUSH	Flushes file for specified unit number. `SUBROUTINE FLUSH (`*unit*`)` `INTEGER ::` *unit*
FORK	Creates a copy of the calling process. `INTEGER FUNCTION FORK()`
FPUTC	Writes a character to the file specified by an HP Fortran 90 logical unit, bypassing normal HP Fortran 90 I/O. `INTEGER FUNCTION FPUTC (`*lunit*`,` *char*`)` `INTEGER ::` *lunit* `CHARACTER ::` *char*
FREE	Releases memory previously allocated with `MALLOC` or `FALLOC`. `SUBROUTINE FREE (`*addr*`)` `INTEGER ::` *addr*
FSEEK	Repositions a file specified by an HP Fortran 90 logical unit. `INTEGER FUNCTION FSEEK (`*lunit*`,` *offset*`,` *from*`)` `INTEGER ::` *lunit*`,` *offset*`,` *from*

FSTAT	Returns detailed information about a file by logical unit number.
	`INTEGER FUNCTION FSTAT (`*lunit*`,` *statb*`)`
	`INTEGER ::` *lunit*`,` *statb*`(12)`

Name	Description and signature
FTELL	Returns the current position of the file associated with the specified logical unit. `INTEGER FUNCTION FTELL (`*lunit*`)` `INTEGER ::` *lunit*

GERROR	Returns the system error message to *string*; available as a subroutine:
	`CHARACTER(LEN=*) ::` *string*
	`SUBROUTINE GERROR (`*string*`)`
	
	And as a function:
	`CHARACTER(LEN=*) FUNCTION GERROR()`

GETARG	Returns command-line arguments. `SUBROUTINE GETARG (`*k*`,` *arg*`)` `INTEGER ::` *k* `CHARACTER(LEN=*) ::` *arg*
GETC	Retrieves a character from HP Fortran 90 logical unit 5. `INTEGER FUNCTION GETC (`*char*`)` `CHARACTER` *char*
GETCWD	Retrieves the pathname of the current working directory. `INTEGER FUNCTION GETCWD (`*dir_name*`)` `CHARACTER(LEN=*) ::` *dir_name*
GETENV	Retrieves the value of an environment variable. `SUBROUTINE GETENV (`*ename*`,` *evalue*`)` `CHARACTER(LEN=*) ::` *ename*`,` *evalue*
GETGID	Retrieves the group ID of the user of the process. `INTEGER FUNCTION GETGID()`

GETLOG	Retrieves the user’s login name; available as a subroutine:
	`SUBROUTINE GETLOG (`*name*`)`
	`CHARACTER(LEN=*) ::` *name*
	
	And as a function:
	`CHARACTER(LEN=*) FUNCTION GETLOG()`

GETPID	Returns the process ID of the current process. `INTEGER FUNCTION GETPID()`
GETUID	Returns the user ID of the user of the process. `INTEGER FUNCTION GETUID()`

Name	Description and signature

`GMTIME`	Returns the Greenwich mean time in HP-UX format within an array of time elements.
	`SUBROUTINE GMTIME (`*stime*`, `*tarray*`)`
	`INTEGER :: `*stime*`, `*tarray*`(9)`

`HOSTNM`	Retrieves the name of the current host. `INTEGER FUNCTION HOSTNM (`*name*`)` `CHARACTER(LEN=*) :: `*name*
`IARGC`	Returns the index of the last command-line argument. `INTEGER FUNCTION IARGC()`

`IDATE`	Returns the date in numerical form.
	`SUBROUTINE IDATE (`*iarray*`)`
	`INTEGER :: `*iarray*`(3)`

`IDATEY2K`	Designed to replace the HP f90 `IDATE` intrinsic. This returns the true year in its third argument, as opposed to the idate intrinsic, which returns the number of years since 1900 in its third argument.
	`SUBROUTEIN IDATEY2K(MONTH,DATE,YEAR)`
	`INTEGER MONTH,DAY,YEAR`
	The `+U77` flag (described in "+U77 option" on page 588) must be used with `IDATEY2K`.

`IERRNO`	Returns the error number of the last detected system error. `INTEGER FUNCTION IERRNO()`
`ISATTY`	Checks whether a logical unit is associated with a terminal device. `LOGICAL FUNCTION ISATTY (`*lunit*`)` `INTEGER :: `*lunit*

`ITIME`	Returns the time in numerical form.
	`SUBROUTINE ITIME (`*iarray*`)`
	`INTEGER :: `*iarray*`(3)`

`KILL`	Sends a signal number to a user's process. `INTEGER FUNCTION KILL (`*pid*`, `*signum*`)` `INTEGER :: `*pid*`, `*signum*
`LINK`	Creates a link to an existing file. `INTEGER FUNCTION LINK (`*name1*`, `*name2*`)` `CHARACTER(LEN=*) :: `*name1*`, `*name2*
`LOC`	Returns the address of an object. `INTEGER FUNCTION LOC (`*arg*`)`

Name	Description and signature

LSTAT	Returns detailed information about the symbolic link to a specified file. (Use STAT to obtain information about the file to which the link points.)
	`INTEGER FUNCTION LSTAT (`*name*`, `*statb*`)`
	`CHARACTER(LEN=*) :: `*name*
	`INTEGER :: `*statb*`(12)`

LTIME	Returns the local time in HP-UX format within an array of time elements.
	`SUBROUTINE LTIME (`*stime*`, `*tarray*`)`
	`INTEGER :: `*stime*`, `*tarray*`(9)`

MALLOC	Allocates memory. `SUBROUTINE MALLOC (`*size*`, `*addr*`)` `INTEGER :: `*size*`, `*addr*
NUM_PROCS	Returns the total number of processors on which the process has initiated threads. `INTEGER FUNCTION NUM_PROCS()`
NUM_ THREADS	Returns the total number of threads that the process creates at initiation, regardless of how many are idle or active. `INTEGER FUNCTION NUM_THREADS()`
PERROR	Retrieves system error messages. PERROR writes a message to HP Fortran 90 logical unit 7 for the last detected system error. `SUBROUTINE PERROR (`*string*`)` `CHARACTER(LEN=*) :: `*string*
PUTC	Writes a character to the file specified by HP Fortran 90 logical unit number 6, bypassing normal HP Fortran 90 I/O. `INTEGER FUNCTION PUTC (`*char*`)` `CHARACTER `*char*
QSORT	Uses the quick-sort algorithm to sort the elements in a one-dimensional array. `SUBROUTINE QSORT (`*array*`, `*len*`, `*isize*`, `*compar*`)` `INTEGER :: `*len*`, `*isize* `EXTERNAL `*compar* `INTEGER(2) `*compar*
RENAME	Renames a file to the specified new name. `INTEGER FUNCTION RENAME (`*from*`, `*to*`)` `CHARACTER(LEN=*) :: `*from*`, `*to*

Name	Description and signature
`SIGNAL`	Allows you to change the action for a signal. `INTEGER FUNCTION SIGNAL (`*signum*`, `*proc*`, `*flag*`)` `INTEGER :: `*signum*`, `*flag* `EXTERNAL `*proc*
`SLEEP`	Suspends the execution of a process for a specified interval. `SUBROUTINE SLEEP (`*itime*`)` `INTEGER :: `*itime*

`STAT`	Returns detailed information about a file by name. When the named file is a symbolic link, `STAT` returns information about the file to which the link points.
	`INTEGER FUNCTION STAT (`*name*`, `*statb*`)`
	`CHARACTER(LEN=*) :: `*name*
	`INTEGER :: `*statb*` (12)`

`SYMLNK`	Creates a symbolic link to an existing file. `INTEGER FUNCTION SYMLNK (`*name1*`, `*name2*`)` `CHARACTER(LEN=*) :: `*name1*`, `*name2*
`SYSTEM`	Executes an HP-UX command. `INTEGER FUNCTION SYSTEM (`*string*`)` `CHARACTER(LEN=*) :: `*string*
`TCLOSE`	Closes the tape device channel and removes its association with *tlu*. `INTEGER FUNCTION TCLOSE (`*tlu*`)` `INTEGER :: `*tlu*
`TIME`	Returns the system time (in seconds) since 00:00:00 Greenwich mean time, January 1, 1970. `INTEGER FUNCTION TIME()`
`TOPEN`	Associates a device name with a tape logical unit. `INTEGER FUNCTION TOPEN (`*tlu*`, `*devnam*`, `*label*`)` `INTEGER :: `*tlu* `CHARACTER(LEN=*) :: `*devnam*
`TREAD`	Reads the next physical record from tape to a buffer. `INTEGER FUNCTION TREAD (`*tlu*`, `*buffer*`)` `INTEGER :: `*tlu* `CHARACTER(LEN=*) :: `*buffer*
`TREWIN`	Rewinds the specified tape to the beginning of the first data file. `INTEGER FUNCTION TREWIN (`*tlu*`)` `INTEGER :: `*tlu*

Name	Description and signature
TSKIPF	Allows the user to skip over files and records. `INTEGER FUNCTION TSKIPF (`*tlu*`,` *nfiles*`,` *nrecs*`)` `INTEGER ::` *tlu*`,` *nfiles*`,` *nrecs*
TSTATE	Allows the user to determine the logical state of the tape I/O channel and to see the tape drive control status register. `INTEGER FUNCTION TSTATE (`*tlu*`,` *fileno*`,` *recno*`,` *errf*`,` *eoff*`,` *eotf*`,` *tcsr*`)` `INTEGER ::` *tlu*`,` *fileno*`,` *recno*`,` *tcsr* `LOGICAL ::` *errf*`,` *eoff*`,` *eotf*
TTYNAM	Returns a blank padded path name of the terminal device associated with a specified logical unit number. `CHARACTER(LEN=*) FUNCTION TTYNAM (`*lunit*`)` `INTEGER ::` *lunit*
TWRITE	Writes a physical record to tape from the specified buffer. `INTEGER FUNCTION TWRITE (`*tlu*`,` *buffer*`)` `INTEGER ::` *tlu* `CHARACTER(LEN=*) ::` *buffer*
UNLINK	Removes a specified directory entry. `INTEGER FUNCTION UNLINK (`*name*`)` `CHARACTER(LEN=*) ::` *name*
WAIT	Waits for a process to terminate. `INTEGER FUNCTION WAIT (`*status*`)` `INTEGER ::` *status*

BLAS routines

Table 65 lists the routines in the BLAS library and briefly summarizes the calculations they perform.

Table 65 BLAS routines

Routine name	Calculation performed
ISAMAX, IDAMAX, ICAMAX, IZAMAX	Return index of largest element in vector.
SASUM, DASUM, SCASUM, DZASUM	Sum absolute values.
SAXPY, DAXPY, CAXPY,ZAXPY	Add scalar multiple of vector to vector.
SCOPY, DCOPY, CCOPY, ZCOPY	Copy a vector.
SDOT, DDOT, CDOTC,CDOTU, ZDOTC, ZDOTU	Compute dot product of two vectors.
SGBMV, DGBMV, CGBMV, ZGBMV	Multiply band matrix times vector.
SGEMM, DGEMM, CGEMM, ZGEMM	Multiply two general matrices.
SGEMV, DGEMV, CGEMV, ZGEMV	Multiply general matrix times vector.
SGER, DGER, CGERC, CGERU, ZGERC, ZGERU	Compute dyadic product of two vectors.
SNRM2, DNRM2, SCNRM2, DZNRM2	Compute Euclidean norm of vector.
SROT, DROT, CROT, ZROT	Apply Givens plane rotation.
SROTM, DROTM	Apply a modified Givens rotation.
SROTG, DROTG, CROTG, ZROTG	Construct Givens plane rotation.

Routine name	Calculation performed
SROTMG, DROTMG	Construct modified Givens plane rotation.
SSBMV, DSBMV, CHBMV, ZHBMV	Multiply symmetric/Hermitian band matrix times vector.
SSCAL, DSCAL, CSCAL, CSSCAL, ZSCAL, ZDSCAL	Scale vector.
SSPMV, DSPMV, CHPMV, ZHPMV	Multiply symmetric/Hermitian packed matrix times vector.
SSPR, DSPR, CHPR, ZHPR	Compute symmetric/Hermitian dyadic product of vector with itself, leaving result in packed form.
SSPR2, DSPR2, CHPR2, ZHPR2	Compute symmetric/Hermitian dyadic product of two vectors, leaving result in packed form.
SSWAP, DSWAP, CSWAP, ZSWAP	Swap two vectors.
SSYMM, DSYMM, CHEMM, CSYMM, ZHEMM, ZSYMM	Multiply two symmetric matrices.
SSYMV, DSYMV, CHEMV, ZHEMV	Multiply symmetric/Hermitian matrix times vector.
SSYR, DSYR, CHER, ZHER	Compute symmetric/Hermitian dyadic product of vector with itself.
SSYR2, DSYR2, CHER2, ZHER2	Compute symmetric/Hermitian dyadic product of two vectors.
SSYR2K, DSYR2K, CHER2K, CSYR2K, ZHER2K, ZSYR2K	Compute symmetric product of matrix and transpose or adjoint of second matrix.
SSYRK, DSYRK, CHERK, CSYRK, ZHERK, ZSYRK	Compute product of matrix and its transpose or adjoint.
STBMV, DTBMV, CTBMV, ZTBMV	Multiply triangular band matrix times vector.
STBSV, DTBSV, CTBSV, ZTBSV	Multiply inverse of triangular band matrix times vector.

Routine name	Calculation performed
STPMV, DTPMV, CTPMV, ZTPMV	Multiply triangular packed matrix times vector.
STPSV, DTPSV, CTPSV, ZTPSV	Multiply inverse of packed triangular matrix times vector.
STRMM, DTRMM, CTRMM, ZTRMM	Multiply triangular matrix by general matrix.
STRMV, DTRMV, CTRMV, ZTRMV	Multiply triangular matrix times vector.
STRSM, DTRSM, CTRSM, ZTRSM	Multiply inverse of triangular matrix by general matrix.
STRSV, DSTRSV,CTRSV, ZTRSV	Multiply inverse of triangular matrix times vector.
XERBLA	Handle errors for BLAS matrix operations (Level 2 and Level 3 routines).

A I/O runtime error messages

This appendix lists and describes the I/O runtime error messages that can be returned by the IOSTAT=*integer-variable* specifier. (For information about this specifier, refer to Chapter 10 for the descriptions of READ, WRITE, OPEN and other I/O statements.) If an I/O error occurs during the execution of an I/O statement, and the statement includes the IOSTAT= *integer-variable* specifier, the status code for the error will be returned in *integer-variable*. Consider the following example:

```
INTEGER ios
.
.
.
OPEN (10, FILE='data_file', ERR=99, IOSTAT=ios)
```

If data_file is successfully opened, ios will return 0; if for any reason the file cannot be opened, a nonzero status code will be returned in ios. By referring to this appendix, you can get information about the error and how to correct it.

Runtime I/O errors

The error information listed in this section includes the codes returned by `IOSTAT=`, plus the following:

- The message that the runtime system would send to standard error if you did not include the `IOSTAT=` specifier.
- A diagnosis of the conditions that might have resulted in the error.
- Actions that the programmer can take to correct the error.

Table 66 Runtime I/O errors

Error no.	Error message	Description	Action
900	ERROR IN FORMAT	FORMAT statement syntax contains an error.	Refer to the syntax for “FORMAT” on page 330. Also see Chapter 8, “I/O and file handling,” on page 171 for the syntax of the format specification and edit descriptors.
901	NEGATIVE UNIT NUMBER SPECIFIED	Unit number was not greater than or equal to zero.	Use a nonnegative unit number.
902	FORMATTED I/O ATTEMPTED ON UNFORMATTED FILE	Formatted I/O was attempted on a file opened for unformatted I/O.	Open the file for formatted I/O or perform unformatted I/O on this file.
903	UNFORMATTED I/O ATTEMPTED ON FORMATTED FILE	Unformatted I/O was attempted on a file opened for formatted I/O.	Open the file for unformatted I/O or perform formatted I/O on this file.

Error no.	Error message	Description	Action
904	DIRECT I/O ATTEMPTED ON SEQUENTIAL FILE	Direct operation attempted on sequential file, direct operation attempted on opened file connected to a terminal.	Use sequential operations on this file, open file for direct access, or do not do direct I/O on a file connected to a terminal.
905	ERROR IN LIST-DIRECTED READ OF LOGICAL DATA	Found repeat value, but no asterisk. First character after optional decimal point was not T or F.	Change input data to correspond to syntax expected by list-directed input of logicals, or use input statement that corresponds to syntax of input data.
907	ERROR IN LIST-DIRECTED READ OF CHARACTER DATA	Found repeat value, but no asterisk. Characters not delimited by quotation marks.	Change input data to correspond to syntax expected by list-directed input of characters, or use input statement that corresponds to syntax of input data.
908	COULD NOT OPEN FILE SPECIFIED	Tried to open a file that the system would not allow for one of the following reasons: access to the file was denied by the file system due to access restriction; the named file does not exist; or the type of access request is impossible.	Correct the pathname to open the intended file.

Error no.	Error message	Description	Action
909	SEQUENTIAL I/O ATTEMPTED ON DIRECT ACCESS FILE	Attempted a BACKSPACE, REWIND, or ENDFILE on a terminal or other device for which these operations are not defined.	Do not use the BACKSPACE, REWIND, and ENDFILE statements.
910	ACCESS PAST END OF RECORD ATTEMPTED	Tried to do I/O on record of a file past beginning or end of record.	Perform I/O operation within bounds of the record, or increase record length.
912	ERROR IN LIST I/O READ OF COMPLEX DATA	While reading complex data, one of the following problems has occurred: no left parenthesis and no repeat value; repeat value was found but no asterisk; or no closing right parenthesis.	Change input data to correspond to syntax expected by list-directed input of complex numbers, or use input statement corresponding to syntax of input data.
913	OUT OF FREE SPACE	Library cannot allocate an I/O block (from an OPEN statement), parse array (for formats assembled at run-time), file name string (from OPEN) characters from list-directed read, or file buffer. The program may be trying to overwrite a shared memory segment defined by another process.	Allocate more free space in the heap area, open fewer files, use FORMAT statements in place of assembling formats at run time in character arrays, or reduce the maximum size of file records.

Error no.	Error message	Description	Action
914	ACCESS OF UNCONNECTED UNIT ATTEMPTED	Unit specified in I/O statement has not previously been connected to anything.	Connect unit using the OPEN statement before attempting I/O on it, or perform I/O on another, already connected, unit.
915	READ UNEXPECTED CHARACTER	Read a character that is not admissible for the type of conversion being performed. Input value was too large for the type of the variable.	Remove from input data any characters that are illegal in integers or real numbers.
916	ERROR IN READ OF LOGICAL DATA	An illegal character was read when logical data was expected.	Change input data to correspond to syntax expected when reading logical data or use input statement corresponding to syntax of input data.
917	OPEN WITH NAMED SCRATCH FILE ATTEMPTED	Executed OPEN statement with STATUS='SCRATCH', but also named the file. Scratch files must not be named.	Either remove the FILE= specifier, or open the file with a status other than STATUS='SCRATCH'.
918	OPEN OF EXISTING FILE WITH STATUS='NEW' ATTEMPTED	Executed OPEN statement with STATUS='NEW', but file already exists.	Either remove the STATUS= specifier from the OPEN statement, or use the STATUS='OLD'; STATUS='UNKNOWN'; or STATUS='REPLACE' specifier.
920	OPEN OF FILE CONNECTED TO DIFFERENT UNIT ATTEMPTED	You attempted to open a file that is already open with a different unit number.	Close the file with the current unit number before attempting to open it with a different unit number.

Error no.	Error message	Description	Action
921	UNFORMATTED OPEN WITH BLANK SPECIFIER ATTEMPTED	OPEN statement specified FORM='UNFORMATTED' and BLANK=xx.	Either use FORM='FORMATTED' or remove BLANK=xx.
922	READ ON ILLEGAL RECORD ATTEMPTED	Attempted to read a record of a formatted or unformatted direct file that is beyond the current end-of-file.	Read records that are within the bounds of the file.
923	OPEN WITH ILLEGAL FORM SPECIFIER ATTEMPTED	FORM= specified string other than 'FORMATTED' or 'UNFORMATTED'.	Use either 'FORMATTED' or 'UNFORMATTED' for the FORM= specifier in an OPEN statement.
924	CLOSE OF SCRATCH FILE WITH STATUS='KEEP' ATTEMPTED	The file specified in the CLOSE statement was previously opened with 'SCRATCH' specified in the STATUS= specifier.	Open the file with a STATUS= , specifying a string other than 'SCRATCH' or do not specify STATUS='KEEP' in the CLOSE statement for this scratch file.
925	OPEN WITH ILLEGAL STATUS SPECIFIER ATTEMPTED	STATUS= specified string other than 'OLD' 'NEW' 'UNKNOWN' 'REPLACE' or 'SCRATCH'.	Use 'OLD', 'NEW', 'UNKNOWN', 'REPLACE' or 'SCRATCH' for the STATUS= specifier in OPEN statement.
926	CLOSE WITH ILLEGAL STATUS SPECIFIER ATTEMPTED	STATUS= specified string other than 'KEEP' or 'DELETE'.	Use 'KEEP' or 'DELETE' for the STATUS= specifier in a CLOSE statement.

Error no.	Error message	Description	Action
927	OPEN WITH ILLEGAL ACCESS SPECIFIER ATTEMPTED	ACCESS= specified string other than 'SEQUENTIAL' or 'DIRECT'.	Use 'SEQUENTIAL' or 'DIRECT' for the ACCESS= specifier in an OPEN statement.
929	OPEN OF DIRECT FILE WITH NO RECL SPECIFIER ATTEMPTED	OPEN statement has ACCESS='DIRECT', but no RECL= specifier.	Add RECL= specifier to OPEN statement, or specify ACCESS= 'SEQUENTIAL'.
930	OPEN WITH RECL LESS THAN 1 ATTEMPTED	RECL= specifier in OPEN statement was less than or equal to zero.	Specify a positive number for RECL= specifier in OPEN statement.
931	OPEN WITH ILLEGAL BLANK SPECIFIER ATTEMPTED	BLANK= specified string other than 'NULL' or 'ZERO'	Use 'NULL' or 'ZERO' for BLANK= specifier in OPEN statement.
933	END (OR BEGIN) OF FILE WITH NO END=x SPECIFIER	End-of-file mark read by a READ statement with no END= specifier to indicate label to which to jump.	Use the END= specifier to handle EOF, or check logic.
937	ILLEGAL RECORD NUMBER SPECIFIED	A record number less than one was specified for direct I/O.	Use record numbers greater than zero.
942	ERROR IN LIST-DIRECTED READ - CHARACTER DATA READ FOR ASSIGNMENT TO NONCHARACTER VARIABLE	A character string was read for a numerical or logical variable.	Check input data and input variable type.
944	RECORD TOO LONG IN DIRECT UNFORMATTED I/O	Output requested is too long for specified (or pre-existing) record length.	Make the number of bytes output by WRITE less than or equal to the file record size.

Error no.	Error message	Description	Action
945	ERROR IN FORMATTED I/O	More bytes of I/O were requested than exist in the current record.	Match the format to the data record.
953	NO REPEATABLE EDIT DESCRIPTOR IN FORMAT STRING	No format descriptor was found to match I/O list items.	Add at least one repeatable edit descriptor to the format statement.
956	FILE SYSTEM ERROR	The file system returned an error status during an I/O operation.	See the associated file system error message.
957	FORMAT DESCRIPTOR INCOMPATIBLE WITH NUMERIC ITEM IN I/O LIST	A numeric item in the I/O list was matched with a nonnumeric edit descriptor.	Match format descriptors to I/O list.
958	FORMAT DESCRIPTOR INCOMPATIBLE WITH CHARACTER ITEM IN I/O LIST	A character item in the I/O list was matched with an edit descriptor other than A or R.	Match format descriptors to I/O list.
959	FORMAT DESCRIPTOR INCOMPATIBLE WITH LOGICAL ITEM IN I/O LIST	A logical item in the I/O list was matched with a edit descriptor other than L.	Match format descriptors to I/O list.
973	RECORD LENGTH DIFFERENT IN SUBSEQUENT OPEN	Record length specified in second OPEN conflicted with the value as opened.	Only BLANK=, DELIM=, and PAD= specifiers may be changed by a redundant OPEN.
974	RECORD ACCESSED PAST END OF INTERNAL FILE RECORD (VARIABLE)	An attempt was made to transfer more characters than internal file length.	Match READ or WRITE statement with internal file size.

Error no.	Error message	Description	Action
975	ILLEGAL NEW FILE NUMBER REQUESTED IN FSET FUNCTION	The file number requested to be set was not a legal file system file number.	Check that the OPEN succeeded and the file number is correct.
976	UNEXPECTED CHARACTER IN ”NAMELIST” READ	An illegal character was found in namelist-directed input.	Be sure input data conforms to the syntax rules for namelist-directed input, or remove illegal character from data.
977	ILLEGAL SUBSCRIPT OR SUBSTRING IN ”NAMELIST” READ	An invalid subscript or substring specifier was found in namelist-directed input. Possible causes include bad syntax, subscript/ substring component out-of-bounds, wrong number of subscripts and substring on non-character variable.	Check input data for syntax errors. Be sure subscript/substring specifiers are correct for data type. Specify only array elements within the bounds of the array being read.
978	TOO MANY VALUES IN “NAMELIST" READ	Too many input values were found during a namelist-directed READ. This message will be generated by attempts to fill variables beyond their memory limits.	Supply only as many values as the length of the array.
979	VARIABLE NOT IN NAMELIST GROUP	A variable name was encountered in the input stream that was not declared as part of the current namelist group.	Read only the variables in this namelist.

Error no.	Error message	Description	Action
980	NAMELIST I/O ATTEMPTED ON UNFORMATTED FILE	An illegal namelist-directed I/O operation was attempted on an unformatted (binary) file.	Specify FORM='FORMATTED' in OPEN statement, or use namelist-directed I/O only on formatted files.
1010	OPEN WITH ILLEGAL PAD SPECIFIER ATTEMPTED	An attempt was made to open a file with an illegal value specified for the PAD= specifier.	Specify either PAD='YES' or PAD='NO'.
1011	OPEN WITH ILLEGAL POSITION SPECIFIER ATTEMPTED	An attempt was made to open a file with an illegal value specified for the POSITION= specifier.	Specify POSITION='ASIS', POSITION='REWIND' or POSITION='APPEND'.
1012	OPEN WITH ILLEGAL DELIM SPECIFIER ATTEMPTED	An attempt was made to open a file with an illegal value specified for the DELIM= specifier.	Specify DELIM= 'APOSTROPHE', DELIM='QUOTE' or DELIM='NONE'.
1013	OPEN WITH ILLEGAL ACTION SPECIFIER ATTEMPTED	An attempt was made to open a file with an illegal value specified for the ACTION= specifier.	Specify ACTION='READ', ACTION='WRITE' or ACTION='READWRITE'.

Glossary

A

actual argument A value, **variable**, or **procedure** that is passed by a call to a procedure (**function** or **subroutine**). The actual argument appears in the source of the calling procedure. See also **dummy argument**.

adjustable array A **dummy argument** that is an **array** having at least one nonconstant **dimension**.

allocatable array A named **array** with the ALLOCATABLE **attribute** whose **rank** is specified at compile time, but whose **bounds** are determined at run time. Storage for the array must be explicitly allocated before the array may be referenced.

archive library A **library** of routines that can be **linked** to an executable **program** at link-time. See also **shared library**.

argument (1) A **variable**, declared in the argument list of a **procedure** or ENTRY statement, that receives a value when the procedure is called (a **dummy argument**). (2) The variable, expression, or procedure that is passed by a call to a procedure (an **actual argument**).

argument association The correspondence between an **actual argument** and a **dummy argument** during execution of a **procedure** reference.

argument keyword A **dummy argument** name. Argument keywords can be used to pass **actual arguments** to a **procedure** in any order if the procedure has an explicit interface.

array A rectangular pattern of **elements** of the same **data type**. The properties of an array include its **rank**, **shape**, **extent**, and **data type**. See also **bounds** and **dimension**.

array constructor A **rank**-one **array** represented as a sequence of **scalar** or array values that may be **constant** or **variable**.

array element An individual, **scalar** component of an **array** that is specified by the array name and, in parenthesis, one or more **subscripts** that identify the element’s position in the array.

array element order The order in **arrays** are laid out in memory. The array element order for HP Fortran 90 arrays is **column-major order**. Array element order can also be used to determine **sequence association**.

array pointer An **array** that has the POINTER **attribute** and may therefore be used to point to a **target** object.

array section A subset of an **array** specified by a **subscript triplet** or **vector subscript** in one or more **dimension**s. For an array a(4,4), a(2:4:2,2:4:2) is an array section containing only the evenly indexed **element**s a(2,2), a(4,2), a(2,4), and a(4,4).

array-valued Having the property of being an **array**. For example, an array-valued **function** has a **return value** that is an array.

association The mechanism by which two or more **names** may refer to the same entity. See also **argument association**, **host association**, **pointer association**, **sequence association**, **storage association**, and **use association**.

assumed-shape array An **array** that is a **dummy argument** to a **procedure** and whose **shape** is assumed (taken) from that of the associated **actual argument**. An assumed-shape array's upper **bound** in each **dimension** is represented by a colon (**:**). See also **assumed-size array**.

assumed-size array An older FORTRAN 77 feature. An **array** that is a **dummy argument** to a **procedure** and whose **size** (but not necessarily its **shape**) is assumed (taken) from that of the associated **actual argument**. The upper **bound** of an assumed-size array's last **dimension** is specified by an asterisk (*). See also **assumed-shape array**.

attribute A property of a **constant** or **variable** that may be specified in a **type declaration statement**. Most attributes may alternately be specified in a separate **statement**. For instance, the ALLOCATABLE statement has the same meaning as the ALLOCATABLE attribute, which appears in a type declaration statement.

automatic array An **explicit-shape array** that is local to a **procedure** and is not a **dummy argument**. One or more of an automatic array's **bounds** is determined upon entry to the procedure, allowing automatic arrays to have a different **size** and **shape** each time the procedure is invoked.

automatic data object A data object declared in a **subprogram** whose storage space is dynamically allocated when the subprogram is invoked; its storage is released on return from the subprogram. Fortran 90 supports **automatic arrays** and automatic character string variables.

B

bit A binary digit, either 1 or 0. See also **byte**.

blank common A **common block** that is not associated with a **name**.

block A series of consecutive **statement**s that are treated as a complete unit and are within a `SELECT CASE`, `DO`, `IF`, or `WHERE` **construct**.

block data program unit A **procedure** that establishes initial values for **variable**s in named **common block**s and contains no executable **statement**s. A block data program unit begins with a `BLOCK DATA` statement.

bounds The minimum and maximum values permitted as a **subscript** of an **array** for each **dimension**. For each **dimension**, there are two bounds—the upper and lower bounds—that define the range of values for **subscript**s.

BOZ constants A literal constant that can be formatted as binary, octal, or hexadecimal. See also **typeless constant**.

built-in functions `%REF` and `%VAL`—HP extensions that can be used to change **argument-**passing rules in **procedure** references.

byte A group of contiguous **bit**s starting on an addressable boundary. In HP machines, a byte is 8 **bits** in length.

C

character A digit, letter, or other symbol in the character set. See Appendix B, “Character set”.

character string A sequence of zero or more consecutive **character**s.

column-major order The default storage method for **array**s in HP Fortran 90. Memory representation of an array is such that the columns are stored contiguously. For example, given the array `a(2,3)`, **element** `a(1,1)` would be stored in the first location, element `a(2,1)`in the second location, element `a(1,2)`in the third location, and so on. See also **row-major order**.

common block A block of memory for storing **variable**s. A common block is a **global entity** that may be referenced by one or more **program unit**s.

compile-line option A flag that can be specified with the `f90` command line to override the default actions of the HP Fortran 90 compiler.

compiler directive A specially-formatted comment within a source **program** that affects how the program is compiled. Compiler directives are not part of the Fortran 90 Standard. In HP Fortran 90, compiler directives provide control over source listing, optimization, and other features.

component A constituent that is part of a **derived type**. A derived type may consist of one or more components. For example, `time%hour` refers to the `hour` component of `time` (and `time` is a

variable whose **data type** is a derived type defined in the **program**).

conformable Two **array**s are **conformable** if both arrays have the same **rank** (number of **dimension**s) and the same **extent** (number of **element**s for each dimension). A **scalar** is conformable with any array.

connected (1) A **unit** is **connected** if it refers to an **external file**. (2) An external file is connected if a unit refers to it. In both cases, connection is established either by the `OPEN` statement or by preconnection. See also **preconnected**.

constant A data object that retains the same value during a **program**'s execution. A constant's value is established when a program is compiled. A constant is either a **literal constant** or a **named constant**.

constant expression An **expression** whose value does not vary during the **program**'s execution. A constant expression's **operand**s are all **constant**s.

construct A series of **statement**s that begins with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with a corresponding `END SELECT`, `END DO`, `END IF`, or `ENDWHERE` statement.

D

data type A named category of data that has a set of values, a way to denote its values, and a set of **operation**s for interpreting and manipulating the values. Fortran 90 **intrinsic** data types include character, complex, double precision, integer, logical, and real. HP Fortran 90 also provides the byte and double complex data types as extensions. See also **derived type**.

deferred-shape array An **allocatable array** or a **pointer** array (an **array** with the `ALLOCATABLE` or `POINTER` **attribute**).

defined assignment A non-intrinsic assignment statement that is defined by an `ASSIGNMENT(=)` interface block and a subroutine.

defined operator An **operator** that is present in an `INTERFACE` statement and has its **operation** implemented by one or more user-defined **function**s.

demand-loadable A process is demand-loadable if its pages are brought into physical memory only when they are accessed.

derived type A user-defined (non-intrinsic) **data type** that consists of one or more **component**s. Each component of a derived type is either an **intrinsic** data type or another derived type.

dimension Each **subscript** of an **array** corresponds to a **dimension** of the array; arrays may have from one to seven dimensions. The number of dimensions is an array's **rank**. See also **extent**.

directive See **compiler directive**.

disassociated A **pointer** that is disassociated points to no target. A pointer becomes disassociated following a `DEALLOCATE` or `NULLIFY` **statement** involving the pointer or by the pointer being associated with (pointing to) a disassociated pointer.

dummy argument An entity whose **name** appears in the **argument** list of a **procedure** or `ENTRY` **statement**. It is associated with an **actual argument** when the procedure is called. The dummy argument appears in the source of the called procedure.

dummy array A **dummy argument** that is an **explicit-shape array**.

dusty deck program An older, pre-FORTRAN 77 program. Presumably called a "dusty deck" program because it was stored on punched cards and has not been changed since. Such programs generally rely on nonstructured programming techniques such as the `GOTO` statement.

E

element See **array element**.

elemental To be **elemental**, an intrinsic **operation**, **procedure**, or assignment must apply independently to every **element** of an **array** or apply independently to the corresponding elements of a set of **conformable** arrays and **scalar**s

equivalencing The process of sharing storage units among two or more data objects by means of the `EQUIVALENCE` **statement**.

executable statement An instruction that causes the **program** to perform one or more computational or branching actions.

explicit interface A **procedure** interface whose properties (including the name and attributes of the procedure and the order and attributes of its arguments) are known by the calling **program unit**. A procedure may have an explicit interface in a **scoping unit** if it is any of the following:

- Described by an **interface block**
- An **internal procedure**
- A **module procedure**
- A **statement function**

explicit-shape array An **array** with explicitly-declared bounds for each **dimension**.

expression A series of **operand**s and (optionally) **operator**s and parentheses that forms either a data reference or a computation. See also **constant**

expression, **initialization expression**, and **specification expression**.

extended operator See **defined operator**.

extent The number of **element**s in one **dimension** of an **array**.

external file A **file** that is stored on a medium external to the executing **program**.

external name The **name** of an object referenced by a **program unit**, as it appears to the **linker**. Case is not significant in the names that appear in Fortran source files; but it is significant in external names.

external procedure A procedure that is not contained in a **main program**, **module**, or another **subprogram**.

F - H

file A sequence of **records** (**character**s or values processed as a unit).

See also **external file** and **internal file**.

function A **procedure** that returns a value (the **function result**) and that can be referenced in an **expression**.

function result The data object returned from a call to a **function**.

generic procedure A **procedure** in which at least one **actual argument** may have more than one **data type**. Generic procedures may be **intrinsic** or user-defined.

global entity A **program unit**, **common block**, or **external procedure** whose **scope** is the entire executable program.

High Performance Fortran (HPF) An extension to the Fortran 90 Standard that provides user-directed data distribution and alignment. HPF is not a standard, but rather a set of features desirable for parallel programming.

host A **program unit** or **subprogram** that contains an **internal procedure** or **module**.

host association The process by which an **internal procedure**, **module procedure**, or **derived type** definition accesses the entities of its **host**.

I - K

initialization expression A more restricted form of **constant expression** that is used to initialize data.

inquiry function An **intrinsic** function whose **return value** provides information based on the principal **argument**s' properties and not the arguments' values.

intent An **attribute** of a **dummy argument** that indicates whether the argument is used for transferring data into the **procedure**, out of the procedure, or both.

internal file A variable that is used as a **file** storage medium for formatted I/O. Internal files are stored in memory and typically are used to convert data from a machine representation to a **character** representation by use of edit descriptors.

internal procedure A **procedure** contained in a **main program** or another **subprogram**.

intrinsic Assignment statements, **data type**s, **operation**s, and **procedure**s are **intrinsic** if they are defined in the Fortran 90 Standard and may be used, without being defined, in any scoping unit.

keyword option A Fortran 90 feature that allows an **actual argument** to appear anywhere in the argument list to a **procedure** reference.

kind type parameter An integer parameter whose value determines the range for an **intrinsic** data type; for example `INTEGER(KIND=2)`. The kind type parameter also determines the precision for complex and real **data type**s.

L - M

label An integer, one to five digits long, that precedes a **statement** and identifies it with a unique number. A statement's label provides a way to transfer control to the statement or to reference it as a `FORMAT` statement.

library A **file** that contains object code for **subroutine**s and data that can be used by programs written in Fortran 90, among other languages. See also **linker**.

linker The `ld` utility. The linker resolves references in a program's source to routines that are not in the source **file** being compiled. The linker matches each reference, if possible, to the corresponding **library** routine.

literal constant A **constant** that does not have a **name**. A literal constant's value is written directly into a program. See also **named constant**.

lower bounds See **bounds**.

main program The first **program unit** that starts executing when a **program** is run. The first **statement** of a main program usually is the `PROGRAM` statement.

module A **program unit** that contains definitions of **derived type**s, **procedure**s, namelists, and **variable**s that are made accessible to other program units. A module begins with the `MODULE` **statement** and its public definitions are made available to other program units by means of the `USE` statement.

module procedure A **procedure** that is contained in a **module** and is not an **internal procedure**.

N - O

name A letter followed by up to 254 alphanumeric characters (letters, digits, underscores, and $) that identifies an entity in an HP Fortran 90 **program unit**, such as a **common block**, **dummy argument**, **procedure**, program unit, or **variable**.

named constant A constant that has a **name**. See also **literal constant**.

numeric type A complex, double precision, integer, or real **data type**.

obsolescent feature A feature defined in the FORTRAN 77 Standard that still is in common use but is considered to be redundant, such as the arithmetic `IF` statement.

The use of obsolescent features is discouraged. The Fortran 90 Standard summarizes the obsolescent features.

operand An **expression** that precedes or follows an **operator**. For example, in `a + b`, both `a` and `b` are operands.

operator A sequence of one or more characters in an **expression** that specifies an **operation**. For example, in `a + b`, `+` is an operator.

option See **compile-line option**.

optional argument A **dummy argument** that does not require a corresponding **actual argument** to be supplied when its **procedure** is invoked.

P - R

pointer A **variable** that has the `POINTER` **attribute**, which enables it to reference (point to) variables of a specified **data type** (rather than storing the data itself).

pointer association The process by which a **pointer** becomes associated with the storage space of its target. Pointer association occurs during pointer assignment or a valid `ALLOCATE` statement.

preconnected Three input/output **unit**s are **preconnected** to files by the operating system and need not be **connected** by the `OPEN` statement. The preconnected units are:

- Unit 5 (standard input)
- Unit 6 (standard output)
- Unit 7 (standard error)

procedure A unit of program code that may be invoked. A procedure can be either a **function** or a **subroutine**.

program A sequence of instructions for execution by a computer to perform a specific task. A program is executable after successful compilation and **linking**.

program unit A **main program**, a **module**, an external **procedure**, or a **block data subprogram**.

rank The number of **dimension**s of an **array**. **Scalar**s have a rank of zero.

record A sequence of values treated as a whole within a **file**.

renaming feature A feature of the USE **statement** that allows **module** entities to be renamed within the **program unit** having access to the entities by **use association**.

return value See **function result**.

row-major order The default storage method for **array**s in C. Memory representation is such that the rows of an array are stored contiguously. For example, given the array `a[3][2]`, the **element** `a[0][0]` would be stored in the first location, element `a[0][1]` in the second location, element `a[1][0]` in the third location, and so on. See also **column-major order**.

S

scalar A data item that has a **rank** of zero and therefore is not an **array**.

scope The part of a **program** in which a **name** or declaration has a single interpretation.

scoping unit A **derived-type** definition, an interface body (excluding derived-type definitions or interface bodies it contains), or a **program unit** or **subprogram** (excluding any derived-type definitions, interface bodies, or subprograms it contains).

sequence association The **association** between **dummy argument** and **actual argument** that occurs when the two differ in **rank** or character length. Dummy and actual arguments are matched **element** by element or character by character, starting with the first and proceeding in order. See also **array element order** and **column-major order**.

sequence derived type A **derived type** whose definition includes the SEQUENCE **statement**. The **components** of a sequence derived type are in the storage sequence as specified in the definition of the derived type.

shape An array's **extent** (number of **element**s) in each dimension and **rank** (number of **dimension**s).

shared library A **library** of routines that can be **linked** to an executable **program** at runtime, allowing the shared library to be used by several programs simultaneously. See also **archive library**.

size The total number of **element**s in an **array**; the product of all its **extent**s.

specific procedure A procedure for which each actual argument must be of a specific data type. See also **generic procedure**.

specification expression A limited form of an **expression** that can appear in a specification **statement**—for example, a **type declaration statement**—and can be evaluated on entry to a **procedure**.

statement A sequence of characters that represents an instruction or step in a **program**. A single statement usually, but not always, occupies one line of a program.

statement function A **function** that returns a **scalar** value and is defined by a single scalar expression.

statement label See **label**.

storage association The **association** of different Fortran objects with the same storage. Storage association is achieved by means of **common blocks** and **equivalencing**.

storage sequence The order in which Fortran objects are laid out in memory. Programmers can control storage sequence by means of **common blocks** and **equivalencing**, and by defining **sequence derived types**. The storage sequence of **arrays** is determined by **array element order**.

stride The increment that may optionally be specified in a **subscript triplet**. If it is not specified, the stride has a value of one.

structure A data object that is **scalar** and is of **derived type**.

structure component See **component**.

subprogram See **procedure**.

subroutine A **procedure** that is referenced by a CALL statement; values returned by a subroutine are usually provided through the subroutine's **arguments**.

subscript A **scalar** value within the bounds of one **dimension** of an **array**. To specify a single array **element**, a subscript must be specified for each of the array's dimensions.

subscript triplet An **array section** specification that consists of a starting **element**, an ending element, and (optionally) a **stride** separated by colons (:).

substring A contiguous segment of a **scalar** character string. Note that a substring is not an **array section**.

T - Z

target A named data object that may be associated with a **pointer**. A target is specified in a TARGET **statement** or in a **type declaration statement** that has the TARGET **attribute**.

type See **data type**.

type declaration statement A **statement** that specifies the **data type** and, optionally, **attributes** for one or more **constants**, **functions**, or **variables**.

typeless constants A **literal constant** that is formatted to represent a **bit** pattern and therefore does not imply the **type** of the constant. **BOZ constants** and Hollerith constants are both typeless.

unit number A logical number that can be **connected** to a **file** to provide a means for referring to the file in input/output **statement**s.

upper bounds See **bounds**.

use association The association of **name**s among different **scoping unit**s as specified by a USE **statement**. See also **module**.

user-defined operator See **defined operator**.

user-defined assignment See **defined assignment**.

variable A data object whose value may be defined and redefined during a **program**'s execution. For example, **array element**s or **array section**s, named data objects, **structure component**s, and **substring**s all can be variables.

vector subscript A method of referencing multiple, possibly discontinuous **element**s of an **array** by using a **rank**-one array of integer values as a **subscript**.

whole array An **array** reference—for example, in a **type declaration statement**—that consists of the array name alone, without the **subscript** notation. Whole array operations affect every **element** in the array, not just a single, subscripted element.

zero-sized array An **array** with at least one **dimension** that has at least one **extent** of zero. A zero-sized array has a **size** of zero and contains no **element**s.

Index

A

Index

Index

B

Index

Index

Index

Index

Index

Index

Index

E

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index