Concurrent Fortran 77 Reference Manual

0890240-100
August 2004

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2881 Gateway Drive, Pompano Beach, FL 33069-4324. Mark the envelope **"Attention: Publications Department**." This publication may not be reproduced for any other reason in any form without written permission of the publisher.

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- January 1989	000	CX/UX 4.0
Current Releease -- July 2004	100	Concurrent Fortran 77 6.1

Preface

Scope of Manual

This manual provides an overview of the Concurrent Fortran compiler, general enhancements to the compiler, violations of the standard, and general source components of the Fortran language.

Information in this manual applies to PowerPC™ platforms as well as RedHawk™ Intel®.

Structure of Manual

A brief description of the chapters and appendixes in this manual follows:

- Chapter 1 provides a general introduction and overview of the Fortran 77 compiler. Language extensions, enhancements, syntax notation, and violations of the 1977 standard Fortran are discussed.
- Chapter 2 describes the source program components; included in this chapter are character sets, statements, syntactical elements, program unit structure and terminology. Also discussed are data types, data constants, storage alignment and arrays.
- Chapter 3 discusses Fortran expressions and assignments. The Fortran language permits arithmetic, character, relational, and logical expressions. The use of Fortran expressions in assignment statements is discussed in detail in this chapter.
- Chapter 4 deals with the specification and declaration statements. Each statement is described in detail containing a definition, syntax line and explanation of the syntax.
- Chapter 5 provides a general description of control statements, their order of execution, and transfer of control. Examples and definitions are provided throughout this chapter.
- Chapter 6 describes the general input and output statements of the Fortran language. The input statements transfer data stored on an external storage medium into memory while the Fortran output statements transfer data from memory to an external storage medium.
- Chapter 7 provides information about format specification, defining size of input and output fields, the type of data being read or written and how the data are to be edited.
- Chapter 8 deals with the subprograms which are units independent of the main program and are written by the user or supplied by the compiler. Inter-language interfacing between C and Fortran is covered in depth.

- Chapter 9 contains the Fortran library description. The library contains the Fortran intrinsic functions as well as functions that are in addition to the Fortran 77 standard.
- Chapter 10 provides the information necessary for compilation and execution. Discussed in this chapter are compiling, preprocessing, linking, and assembling. Also included in this chapter are the **f77** compiler options and necessary support files.
- Appendix A illustrates array storage.
- Appendix B lists non-standard extensions to Fortran 77.
- Appendix C enumerates incompatibilities with Fortran 66.

The index contains an alphabetical list of topics, names, etc. found in the manual.

Man page descriptions of programs and library routines can be found in system manual pages.

Syntax Notation

The following notation is used throughout this guide:

italic	Books, reference cards, and items that the user must specify appear in *italic* type. Special terms may also appear in *italic*.
list bold	User input appears in **list bold** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **list bold** type.
`list`	Operating system and program output such as prompts and messages and listings of files and programs appears in `list` type. Keywords also appear in `list` type.
emphasis	Words or phrases that require extra emphasis use emphasis type.
`[]`	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments
{}	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces with the choice.
`...`	An ellipsis follows an item that can be repeated.
(H)	Sections, chapters, and appendixes that document Fortran extensions and implementation details particular to Concurrent bear this suffix.
~~b~~	Blank spaces in input and output records and formats are designated as ~~b~~s.
Concurrent	Concurrent-specific text and commands appear in Concurrent type. On color monitors, this is blue.

ConcurrentParameter	Concurrent-specific parameters appear in Concurrent Parameter type. On color monitors, this is blue.

Referenced Publications

The following publications are referenced in this document:

0890288	HAPSE Reference Manual
0890423	PowerUX Programming Guide
0890459	Compilation Systems Volume 1 (Tools)
0890460	Compilation Systems Volume 2 (Concepts)
0891019	C Reference Manual

Contents

Chapter 1 Introduction

Chapter 2 Source Program Components

Chapter 3 Expressions and Assignment Statements

Chapter 4 Specification Statements

Chapter 5 Control Statements

Chapter 6 Fortran Input/Output

Chapter 7 Formatted Input and Output

Chapter 8 Subprograms and Statement Functions

Chapter 9 Fortran Library

Chapter 10 Compilation and Execution (H)

Appendix A Array Storage

Appendix B Non-Standard Extensions to Fortran 77 (H)

Appendix C Incompatibilities with Fortran 66

Glossary

Index

Tables

1
Introduction

1
Introduction

Overview

The Fortran compiler is based on the full language definition of American National Standard Fortran X3.9–1978. This chapter details enhancements made to the compiler and general syntax notation used throughout this manual.

Language Extensions

The Fortran 77 standard includes almost all of Fortran 66 as a subset. The most important additions are:

- A character string data type
- File–oriented input/output statements
- Direct access I/O.

In addition to implementing the language specified in the new standard, this compiler implements a few extensions. Most are useful additions to the language. The other extensions make it easier to communicate with C procedures.

Enhancements (H)[1]

The following enhancements have been made to standard Fortran 77:

The Fortran character set permits lowercase characters, dollar signs, and underscores in symbolic names; any uppercase characters in a symbolic name are automatically translated to lowercase.

A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line. The remaining characters form the body of the line. If a tab appears elsewhere on a line, the compiler treats the tab as a blank space.

Lines may extend to 132 columns with the **`-col132`** command-line option.

1. Sections, chapters, and appendixes that document Fortran extensions and implementation details particular to Concurrent Computer Corporation bear this suffix.

Octal, hexadecimal, binary, and Hollerith data are supported.

Null character strings are supported.

Hollerith data can be stored as the value of numeric or logical variables and array elements.

New operators, `.XOR.`, `.ROTAT.`, and `.SHIFT.` have been added, and the logical operators now may be used with integer operands to perform bitwise logical computations.

The Fortran compiler supports the `DATAPOOL` global data mechanism.

The `POINTER` statement declares pointer blocks. Pointer blocks are essentially based common blocks, a base address for storage of the member variables is supplied via **`malloc(3F)`**, for example.

The Fortran compiler recognizes an automatic storage class.

The Fortran compiler supports the `CEXTERNAL` keyword and other useful mechanisms to interface directly with C language routines.

The Fortran compiler supports a `DOUBLE COMPLEX` data type. Each double complex number is represented by a pair of double–precision real variables. A double complex version of every `COMPLEX` built–in function is provided. The specific function names begin with "`z`" rather than "`c`".

The compiler also accepts declarations of `LOGICAL *1` and `BYTE`, which are one byte long. Many specific intrinsic function names have been added to facilitate these new data types.

`INTEGER *1` variables may be unsigned on option.

The `IMPLICIT` statement may take the keyword `UNDEFINED` in place of a type, to turn off implicit typing for variables starting which specified letters.

The compiler permits single subscripts in `EQUIVALENCE` statements and assumes that all missing subscripts are equal to one. The compiler prints a warning message for each incomplete subscript.

New control constructs have been added, including `LOOP`, `WHILE`, `DO–UNTIL`, `FOR`, and `SELECT CASE`.

Namelist–directed I/O has been implemented in Concurrent Fortran.

The Fortran 77 standard introduces internal files (memory arrays), but restricts their use to formatted, sequential I/O statements. The Fortran I/O system also permits use of internal files in direct reads and writes.

The compiler accepts a variable length input format. An "`&`" symbol in the first position of a line indicates that the line continues a statement that is on the previous line.

Entry points of type `CHARACTER` may have different lengths.

`INTERNAL` subroutines and functions are supported.

Violations of the Standard

The Concurrent implementation of Fortran 77 violates the 1977 standard as described below.

T and TL Formats (H)

The implementation of the `T` (absolute tab) and `TL` (left tab) format descriptors is slightly imperfect. These codes allow you to reread or rewrite part of a record that has already been processed. The implementation uses seeks, so if the unit (for example, a terminal) does not allow seeks, a run–time error message,

```
can't seek
```

is generated and the program aborts.

The chosen implementation places no upper limit on the length of a record. You need not "predeclare" a record length except where specifically required by the compiler or the operating system.

2
Source Program Components

2
Source Program Components

General Component Information

A Fortran *source program* consists of one or more program units containing Fortran statements and optional comments. A *program unit* is classified either as a main program or a subprogram. Only one main program is permitted in a source program. *Subprograms*, if any, are identified with a `FUNCTION`, `SUBROUTINE`, or `BLOCK DATA` statement as the first statement in the program unit.

Fortran Character Set

The Fortran character set is composed of the uppercase and lowercase letters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

the decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

and the following special characters:

Character	Name	Character	Name
b	Blank	$	Dollar Sign
=	Equals	'	Single Quote or Apostrophe
+	Plus	"	Double Quote
-	Minus	:	Colon
*	Asterisk	!	Exclamation Point
/	Slash	_	Underscore
(	Left Parenthesis	<t>	Tab
)	Right Parenthesis	&	Ampersand
,	Comma	%	Percent Sign
.	Period or Decimal Point	#	Sharp Sign

Treatment of Uppercase and Special Characters

The compiler translates uppercase letters to their lowercase counterparts for keywords and symbolic names. Uppercase letters appearing in character or Hollerith strings are not translated to lowercase, and therefore, are not equivalent to their lowercase counterparts in character comparisons during program execution.

If a character other than those listed appears in any Fortran statement within the source program (e.g., a backspace character, `#`, `%`, `>`, `<`, `[`, `]`, `^`, `?`, etc.), the compiler produces a syntax error. ASCII characters that are not a part of the language can be used in comments, in character constants, and in Hollerith constants.

Collating Sequence

The *collating sequence* is the arrangement of characters in an order such that when two characters are compared numerically, one character is either less than, equal to, or greater than the other. The order of the characters in the ASCII character set determines the collating sequence. Uppercase and lowercase letters are arranged alphabetically in the collating sequence and, when compared, have a different internal representation. The set of digits 0 through 9 precede the uppercase letters A through Z in the collating sequence, and the uppercase letters precede the lowercase letters a through z. Refer to the **`ascii(5)`** man page for the full ASCII character set.

Syntactical Elements of the Language

The syntactical elements of the Fortran language are symbolic names, keywords, statement labels, operators, and constants.

symbolic name	Also called *identifier*, consists of 1 to 1023 letters, digits, dollar signs, or underscores, the first of which must be a letter.
keyword	Identifies a Fortran statement (e.g., `DO`, `FORMAT`, `IF`, `STOP`, etc.) or is a separator in a Fortran statement (e.g., `THEN`, etc.). A keyword identifying a Fortran statement is usually the first word of the statement. Whether a particular sequence of characters represents a keyword or a symbolic name is implied by context. Fortran keywords have no abbreviated forms.
statement label	Also called a *statement number*, is one to five decimal digits appearing anywhere in columns 1 through 5 of the initial line of a fixed format statement. Zero is not a valid statement label. Statement labels provide a point of reference so that another statement within the program unit can refer to the labeled statement.
operator	Mnemonic or special character used in Fortran expressions to perform arithmetic computations, to concatenate character strings, or to perform relational and logical comparisons.

data constant — A fixed value that is not subject to change. It can be a signed or unsigned number, a logical value, a character string (literal), or a Hollerith string. Numeric constants are interpreted as base-10 (decimal) numbers unless they are explicitly in another base.

Fortran Statements

Fortran statements are classified as executable or non-executable. *Executable statements* specify actions to be performed and are identified by Fortran keywords. *Non-executable statements* are directives to the compiler which describe the characteristics and arrangement of input data, set the initial values for variables and array elements, indicate input and output editing information, define and classify program units, or specify entry points in subprograms.

Table 2-1. Executable Statements

ACCEPT	DO	IF	RETURN
Assignment statements	ELSE	INQUIRE	REWIND
ASSIGN	ELSE IF	LOOP	SELECT CASE
BACKSPACE	END	OPEN	STOP
CALL	END INTERNAL	PAUSE	TYPE
CASE	ENDFILE	PRINT	UNTIL
CASE DEFAULT	EXIT	PUNCH	WHILE
CLOSE	FOR	READ	WRITE
CONTINUE	GOTO	REPEAT	

Table 2-2. Non-Executable Statements

Statement function definition	DOUBLE COMPLEX	INTRINSIC
AUTOMATIC	DOUBLE PRECISION	LOGICAL
BLOCK DATA	ENTRY	NAME
BYTE	EQUIVALENCE	NAMELIST
CEXTERNAL	EXTERNAL	PARAMETER
CHARACTER	FORMAT	PROGRAM
COMMON	FUNCTION	REAL
COMPLEX	IMPLICIT	SAVE
DATA	INTEGER	STATIC
DATAPOOL	INTERNAL FUNCTION	SUBROUTINE
DIMENSION	INTERNAL SUBROUTINE	

Lines

A Fortran statement is written in one or more lines containing four fields. Each new statement must begin on a new line. The first line of the statement is the initial line, and the statement can be continued on succeeding lines called *continuation lines*.

Each line in a program unit is one of the following:

- Initial line of a Fortran statement.
- Continuation line for a Fortran statement.
- Comment line.
- Include line.

Each Fortran statement line in a source program consists of the following fields:

Column	Fields
1-5	Statement Label Field
6	Continuation Field
7-72	Statement Field
73-end	Identification Field

With the **`-col132`** option, each Fortran statement line in a source program consists of the following fields. Note there is no identification field. The statement field extends to column 132.

Column	Fields
1-5	Statement Label Field
6	Continuation Field
7-132	Statement Field

Statement Label Field

A *statement label* appears on the initial line of a statement anywhere in columns 1 through 5.

The following rules apply for labels:

- A label must be a one- to five-digit, non-zero, unsigned integer.
- Blanks or leading zeroes that are part of the label are insignificant.
- Labels may not contain letters or special characters.
- No characters other than those comprising the label may appear in the statement label field.
- No two statements within the same program unit may have the same label.
- Continuation lines and comment lines may not have statement labels.
- A tab character in the statement label field indicates the end of the statement number. Characters after the tab are regarded as virtually starting in column 7.

These symbols and characters have the following meaning when in column 1:

`C`	Comment
`*`	Comment
`#`	Comment
`&`	Line continues a statement that is on a previous line
`D`	Debugging line

An executable statement may refer only to other executable statements or to `FORMAT` statements by statement label.

A label on a non-executable statement is interpreted as referring to the next sequential executable statement when the label is used as the object of a `GO TO` statement.

Continuation Field

A statement is continued on another line by placing a continuation character in the continuation field (column 6) of the next line or by placing an ampersand (&) in column 1 of the next line. The continuation field must be blank or contain a zero for the initial line of a statement, but must contain a continuation character if the line is to be interpreted as a continuation line and not as the initial line of a new statement. Any character from the Fortran character set except blank or zero can be used as a continuation character. Continuation lines cannot have statement labels in the statement label field.

There is no fixed limit on the number of continuation lines allowed in a statement. The number of continuation lines is limited by the memory available during compilation, which invariably exceeds the ANSI limit of 19 continuation lines.

Statement Field

The statement field contains the text of the Fortran statement. Each statement consists of combinations of the syntactical items of the language. Statements are written between columns 7 and 72 of the initial line and of any continuation lines. The initial line of the statement must have a blank or `0` in column 6. Blanks before, after, or between syntactical items in Fortran statements are insignificant unless they are part of a character or Hollerith constant. For example, both of the following statements are correct:

```
IF(NUMBER.GE.0)GOTO10
IF (NUMBER .GE.  0) GO TO 10
```

Care should be taken to ensure that no part of the text of a statement appears after column 72, since the compiler does not process any line after column 72. Any symbolic name or other syntactical item can be split across two lines; the compiler resumes its scan of the next line in column 7 of the continued line. Note, however, that this practice reduces the readability of the source program.

With the **`-col132`** option, statement text may extend to column 132. Text that appears beyond column 132 will be ignored with a caution.

Identification Field

Columns 73 through the end of the line comprise the identification field. These columns are ignored by the compiler and can be used to sequentially number the lines of the entire source program or of particular program units. Letters, numbers, and special characters can be used in these columns.

If a tab character appears in the statement label field, some of the characters in the identification field are regarded as statement field characters.

With the **`-col132`** option the identification field does not exist.

Comments

Comment lines are ignored by the compiler. Comments are used to describe the actions of particular sections of program units, the meaning of symbolic names, the input and output data, etc., and are a useful documentation aid for the programmer.

A `C`, `*`, or `#` in column 1 of a line indicates a comment line. The one exception is the `#pragma` directive; see "`#pragma Lines (H)` on page 7. The comment is entered anywhere from column 2 through the end of the line. Comment lines cannot be

continued on succeeding lines using a continuation character in column 6. The text of the comment is continued by repeating the "C" in column 1 of additional comment lines. Comment lines can be interspersed with continuation lines. Comment lines that follow an END statement are considered part of the next program unit.

The compiler also recognizes the exclamation point as a comment indicator. If "!" appears on the same line as a Fortran statement, the remainder of the line is ignored unless the exclamation point appears in a character string or Hollerith constant. As with other characters, a "!" in column 6 indicates a continuation line.

Blank Lines

The compiler ignores blank lines in a source program. Blank lines can be used freely in the source program to set off sections in a program unit to improve readability. The compiler treats blank lines as comment lines and prints them in the source listing.

Debugging Lines (H)

Debugging lines are placed in a program unit and treated either as valid statements or as comments depending on whether the debug option (**-D**) is specified at compile time. Lines containing debugging statements are indicated by a "D" in column 1. Debugging lines can be interspersed with continuation and comment lines, and they can be initial lines with optional statement labels or continuation lines.

#pragma Lines (H)

The compiler recognizes a single #pragma directive. The '#' character must appear in column 1.

```
#pragma options various compiler-related -Q options
```

The #pragma options line may be used to issue command-line options within a source file. Only compiler-appropriate **-Q** options may be used, i.e., **-Qobjects=1000** is appropriate whereas **-Qfile_buffer_limit=15000000** does not apply to the compiler and will result in an error. No other options are recognized. The #pragma options directive applies to the entire source file following its appearance. If #pragma is used more than once to control options for different subprograms, the #pragma line intended for a specific subprogram must appear within its body. The scope of the directive still extends beyond the subprogram. An example of a valid directive:

```
#pragma options -Qobjects=950 -Qgrowth_limit=400
```

Include Lines (H)

The compiler replaces the statement

```
INCLUDE "namespec"
```

with the contents of the file *namespec*. At present, you may nest up to ten INCLUDE statements. If a relative path name is specified in *namespec*, it is relative to the directory the source file is in, not the directory from which the compiler is invoked.

The compiler also allows the user to specify a list of directories in which to search for Fortran INCLUDE files. This can be accomplished by using the **-I** option which allows for single directory searches (but can be specified multiple times) or listing directories in the F77INCLPATH environment variable. Each of these methods is discussed in Chapter 10.

The search order, which applies only to files whose names do not begin with a "/", is:

- The directory containing the include file;
- Directories specified with **-I** options; and
- Directories listed in F77INCLPATH, in left to right order.

Any relative path information supplied in the INCLUDE name is preserved during the search.

Program Unit Structure and Terminology

Every program unit must have a name. The name of a main program is supplied implicitly by the compiler or by you in a PROGRAM or NAME statement. You must supply the name for any FUNCTION or SUBROUTINE subprogram; the name is the first identifier in a FUNCTION or SUBROUTINE statement.

Program units can appear in the source program in any order; however, within a program unit, there are restrictions on the order in which statements may appear.

The restrictions within a program unit are as follows:

- A statement that names a program unit must always be the first statement of the program unit.
- Specification statements must precede any statement function definitions and executable statements. IMPLICIT statements may appear anywhere among the specification statements. DATA statements may appear anywhere.
- FORMAT and PARAMETER statements may appear anywhere in a program unit.
- ENTRY statements may appear anywhere in a subprogram except in an IF, FOR, WHILE, DO-UNTIL, or LOOP block, between a DO loop statement

and the statement ending the DO loop, or within an INTERNAL subprogram.

- Statements defining statement functions (see Chapter 4) must precede any executable statements.
- The END statement must be the last statement in a program unit.
- Comment lines may appear anywhere in a source program. However, any comment lines after an END statement are considered part of the next program unit.
- #pragma lines may appear anywhere in a source program. The directive applies to the current program unit and all following program units.
- INTERNAL subprograms may appear anywhere an executable statement may appear.

Table 2-3 illustrates the required order of statements and lines in a program unit. Vertical lines separate statements that can be interspersed. Horizontal lines show statements that must be separated.

Table 2-3. Order of Statements and Lines

<table>
<tr><td rowspan="4">Comment lines,
Blank lines,
Debugging lines,
#pragma lines,
Include lines</td><td colspan="3">PROGRAM, NAME, FUNCTION, SUBROUTINE
or BLOCK DATA statements</td></tr>
<tr><td rowspan="3">FORMAT,
PARAMETER,
and
ENTRY
statements</td><td rowspan="3">DATA
statements</td><td>Specification
statements</td></tr>
<tr><td>Statement
function
statements</td></tr>
<tr><td>Executable
statements and
INTERNAL
subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

Execution Sequence

Executable statements comprise the execution sequence of the program, and each executable statement is processed sequentially unless a transfer of control statement indicates otherwise. Execution begins with the first executable statement in the source program. Non-executable statements and comments do not affect the execution of the program.

A transfer of control causes execution to proceed with a statement other than the next sequential statement in the program unit.

PROGRAM and NAME Statements

A PROGRAM statement gives a name to the main program and, if used, must be the first statement of the main program unit. The keyword NAME can be used instead of the keyword PROGRAM to name a main program.

SYNTAX

```
PROGRAM name

NAME name
```

DESCRIPTION

name Specifies the name of the main program. The selected name must conform to the rules for symbolic names and must not be used to name any subprogram, common block, or entry point within the same source program.

A main program contains any valid Fortran statement except a FUNCTION, SUBROUTINE, BLOCK DATA, ENTRY, or RETURN statement. A SAVE statement in a main program has no effect.

In the following example, EDITOR is the name of a Fortran main program:

```
PROGRAM EDITOR
. . .
END
```

END Statement

The final statement in any program unit must be an END statement; END indicates the end of a main program, subprogram, or block data subprogram.

SYNTAX

```
END
```

The END statement must begin on an initial line and be the only syntactical item on the line.

Symbolic Names

A symbolic name consists of 1 to 1023 letters, digits, dollar signs, or underscores, the first of which must be a letter. Blanks are ignored inside symbolic names.

Correct	Incorrect	Explanation
`PROGA`	`12ABC`	First letter not alphabetic
`SORT$PGM`	`*V:DATA`	Invalid characters
`TRIM_VAL`	`_beg`	First letter not alphabetic
`CUMULATIVE SUM`		
`K34`		
`Lcase`		

The name `CUMULATIVE SUM` is interpreted as one name, cumulative sum, and the variable name Lcase is converted to lower case (lcase).

Symbolic names are used to define the following program entities:

Constants	Statement functions
Variables	Function subprograms
Arrays	Subroutine subprograms
Actual arguments	Block data subprograms
Dummy arguments	Function entry points
Main programs	Subroutine entry points
Intrinsic functions	Common blocks
Internal subprograms	Datapool names

Associated with a symbolic name is the concept of *scope*. The scope of a symbolic name is “local” to the program unit containing the name or “global”, i.e., known in all program units in the source program. A name with local scope represents only one entity in a single program unit, but the same name can represent another entity in another program unit. A name with global scope must represent only one global entity throughout the entire source program.

Table 2-4 indicates the scope of symbolic names for particular program entities

Table 2-4. Scope of Symbolic Names for Program Entities

Program Entity	Scope
Constants	Local
Symbolic names of constants	Local
Variable names	Local
Array names	Local
Dummy argument names	Local
Main program name	Global
Intrinsic function names (generic and specific)	Local
Statement function names	Local
External function names	Global
Subroutine names	Global
Block data subprogram names	Global
Function entry point names	Global
Subroutine entry point names	Global
Internal subprogram names	Local
Common block names	Global
Datapool names	Local

The scope of a dummy argument name in a statement function definition, and the scope of an implied-`DO` variable in an input/output statement or in a `DATA` statement, is that statement.

Data Types

The *data type* of a symbolic name implies the kind of data the symbolic name represents, as well as how large, how small, and, for numeric quantities, how accurate the value of the symbolic name can be. The maximum and minimum size for data of a particular type and the accuracy limits on the data depend on the storage unit sizes defined for the data type.

Among the program entities defined above, the concept of data type applies to

Constants	Intrinsic functions
Variables	Statement functions
Arrays	Function subprograms
Dummy arguments	Function entry points

A symbolic name identifying a main program, subroutine, common block, or block data subprogram has no data type.

The Fortran compiler permits the following data types:

`INTEGER` An optionally signed whole number containing no fractional portion and no decimal point. Integers are stored in one-byte, two-byte (word), or four-byte (long word) form.

`REAL` An optionally signed real number containing an integer part or a fractional part, or both. Numbers of type real are stored in four-byte form (with a 23-bit mantissa).

`DOUBLE PRECISION` An optionally signed real number that is stored with a greater degree of accuracy than a single precision real number. Double precision numbers are stored in eight-byte form (with a 55-bit mantissa).

`COMPLEX` A pair of optionally signed real numbers is stored. The first is the real portion of a complex number, and the second is the imaginary portion of a complex number.

`DOUBLE COMPLEX` A pair of double precision numbers is stored. The first is the real portion of a double precision complex number, and the second is the imaginary portion of a double precision complex number.

`LOGICAL` A value representing the logical concept true or false. Logicals are stored in one-byte, two-byte (word), or four-byte (long word) form.

`CHARACTER` A non-numeric, non-logical value consisting of a string of ASCII characters.

Table 2-5. Fortran Data Types

Data Type	Lengths	Comments
`INTEGER`	`INTEGER*1` or `BYTE`	Uses 1 byte of storage. Values range from -128 to 127. With the **`-uns_int1`** option, values range from 0 to 255.
	`INTEGER*2`	Uses 2 bytes of storage. Values range from -32768 to 32767.
	`INTEGER*4`	Uses 4 bytes of storage. Values range from -2147483648 to +2147483647.

Table 2-5. Fortran Data Types (Cont.)

Data Type	Lengths	Comments
REAL	REAL*4	Uses 4 bytes of storage. Values range from 1.17549435082228740e-38 to 3.40282346638528854e+38 in IEEE floating-point format. Values are accurate to 6 decimal digits in each floating-point format
DOUBLE PRECISION	REAL*8	Uses 8 bytes of storage. Values range from 2.225073850720140e-308 to 1.79769313486231470e+308 in IEEE floating-point format. Values are accurate to 15 decimal digits in IEEE floating-point format.
COMPLEX		Uses 4 bytes of storage in each part. Values in each part range from 1.17549435082228740e-38 to 3.40282346638528854e+38 in IEEE floating-point format. Values are accurate to 6 decimal digits in IEEE floating-point format.
DOUBLE COMPLEX		Uses 8 bytes of storage in each part. Values in each part range from 2.22507385850720140e-308 to 1.79769313486231470e+308 in IEEE floating-point format. Values are accurate to 15 decimal digits in IEEE floating-point format. with 16 digits of accuracy in each part.
LOGICAL	LOGICAL*1	Uses 1 byte of storage. Values are .TRUE. or .FALSE. See "Implementation of the LOGICAL Data Type (H)" on page 3-22.
	LOGICAL*2	Uses 2 bytes of storage. Values are .TRUE. or .FALSE. See "Implementation of the LOGICAL Data Type (H)" on page 3-22.
	LOGICAL*4	Uses 4 bytes of storage. Values are .TRUE. or .FALSE. See "Implementation of the LOGICAL Data Type (H)" on page 3-22.
CHARACTER	CHARACTER*n	Uses n bytes of storage, storing one character per byte. Values of n are positive.

Default Lengths for Data Types

The default length for each data type is dependent on the data type. INTEGER and LOGICAL default to 4 bytes but can be changed at compile time from 4 bytes to 2 bytes. This is done by using the **-i2** option.

Notes:

- Assigning a single precision value to an entity of extended precision does not result in increased accuracy, unless the single precision value is a constant.
- Use of the **-i2** option does not override any explicit length declarations for integer or logical entities.
- A program entity is defined to be of a certain data type implicitly by the first letter of the name or explicitly by an explicit type statement (Refer to Chapter 4).

Storage Alignment

There is a minimal alignment requirement imposed by the machine for each data type.

`INTEGER *1`, `LOGICAL *1`, `BYTE`, and `CHARACTER` data must be aligned on byte boundaries. `INTEGER *2` and `LOGICAL *2` data must be aligned to two-byte boundaries. All other data types must be aligned to four-byte boundaries. Use of `REAL *8` and `COMPLEX *16` variables aligned on eight-byte boundaries achieves a slightly higher level of data access performance. Such eight-byte padding occurs automatically for local variables, and for common blocks, pointer blocks and datapools unless the user forces four-byte alignment by using the **`-Qalign_double=4`** or **`-stdf77`** option.

The compiler and subsequent system components attempt to align data items to the required boundaries when possible without direction from the user. Specifically, the compiler reserves filler areas in common blocks and pointer blocks to bring data to the mandatory alignment. When this is done, a warning is issued by the compiler.

When an `EQUIVALENCE` statement disallows storage allocation which meets the mandatory requirements, a fatal error occurs and a diagnostic message indicates the variables concerned.

Data Constants

Data constants are numeric, character, logical, or Hollerith values that do not vary. A constant can be given a symbolic name with the `PARAMETER` statement.

A negative numeric constant contains a minus sign. If no sign is present in a numeric constant or if a plus sign is present, the number is considered positive. A signed complex number of the following form:

```
-(3, 4)
```

is equivalent to:

```
(-3.0, -4.0)
```

Embedded blanks in a numeric constant appearing in an assignment statement or a Fortran expression do not affect the interpretation of the constant; however, blanks in numeric constants read from data records are significant in list-directed I/O and, for formatted I/O, are interpreted as zeroes or are interpreted as being insignificant under the control of the `BN` and `BZ` format descriptors (see Chapter 7). Blanks in character or Hollerith constants are significant.

The compiler permits the following constants:

Character	Hollerith
Complex	Integer
Double Complex	Logical
Double Precision	Real

The following constants are permitted wherever numeric constants appear:

Hexadecimal
Octal
Binary

Hexadecimal Data (H)

Hexadecimal constants contain digits drawn from the set:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F
```

SYNTAX

z'*n*'
z"*n*"
x'*n*'
x"*n*"
'*n*'x
"*n*"x
*w*x*n* (where *n* has *w* digits, similar to Hollerith
*w*z*n* (where *n* has *w* digits, similar to Hollerith
0x*n*
0z*n*

DESCRIPTION

n	Specifies a base-16 whole number
x and z	Can be specified in either upper- or lowercase
w	Is an integer indicating the length in characters of the base-16 number

If the last form is used, where the width of the hexadecimal constant is specified as zero, all legal digits that follow are used to form the constant value, with truncation or zero padding, as needed. Thus, hexadecimal constants of the forms `0x4` or `0zD4D413` are legal.

A hexadecimal constant is signed or unsigned and contains up to 16 digits. Each hexadecimal constant assumes a data type based on its context in a data statement or expression. If an obvious context does not exist, it is treated as a four-byte integer, truncated or zero-extended to the left as necessary. For example, if a hex constant is added to an `INTEGER*2` variable, it is treated as an `INTEGER*2` constant. If it is added to a real variable, then the constant is interpreted as a real constant.

Examples:

Correct	Incorrect	Explanation
`z'0'`	`'G23'X`	Invalid digit
`'ffffffff'x`	`z'123`	No closing apostrophe
`z'0123456789abc'`	`z'456″`	Unmatched string delimiter
`4xda12`	`'45b'z`	z follows constant value
`0zABCD`	`3Zab`	Constant value shorter than width

Octal Data (H)

Octal constants contain digits drawn from the set:

```
0 1 2 3 4 5 6 7
```

SYNTAX

○'*n*'
○"*n*"
'*n*'○
"*n*"○
w○*n* (where *n* has *w* digits, similar to Hollerith)
0○*n*

DESCRIPTION

n	Specifies a base-8 whole number
○	Can be specified in either upper- or lowercase
w	Specifies an integer indicating the length in characters of the base-8 number

If the last form is used, where the width of the octal constant is specified as zero, all legal digits that follow are used to form the constant value, with truncation or zero padding, as needed. Thus, octal constants of the forms `0○4` or `0○000577` are legal.

An octal constant is signed or unsigned and contains up to 22 digits. Each octal constant assumes a data type based on its context in a data statement or expression. If an obvious context does not exist, it is treated as a four-byte integer, truncated or zero-extended to the left as necessary. For example, if an octal constant is added to an `INTEGER*2` variable, it is treated as an `INTEGER*2` constant. If it is added to a real variable, then the constant is interpreted as a real constant.

Examples:

Correct	Incorrect	Explanation
○'0'	'823'○	Invalid digit in constant value
"07"○		
4O0422	3O06	Constant value shorter than width
0○00777		

Binary Data (H)

Binary constants contain digits drawn from the set:

```
0 1
```

SYNTAX

```
b'n'
b"n"
'n'b
"n"b
wbn (where n has w digits, similar to Hollerith)
0bn
```

DESCRIPTION

n	Specifies a base-2 whole number
b	Can be specified in either upper- or lowercase
w	Specifies an integer indicating the length in characters of the base-2 number

If the last form is used, where the width of the binary constant is specified as zero, all legal digits which follow are used to form the constant value, with truncation or zero padding, as needed. Thus, binary constants of the forms `0b1` or `0B110101` are legal.

A binary constant is signed or unsigned and contains up to 64 digits. Each binary constant assumes a data type based on its context in a data statement or expression. If an obvious context does not exist, it is treated as a four-byte integer, truncated or zero-extended to the left as necessary. For example, if a binary constant is added to an `INTEGER *2` variable, it as treated as an `INTEGER *2` constant. If it is added to a `REAL` variable, then the constant is interpreted as a real constant.

Examples:

Correct	Incorrect	Explanation
`b'0'`	`'021'b`	Invalid digit
`"01"B`		
`4B0101`	`3B11`	Constant value shorter than width
`0b10111`		

Integer Data

An *integer constant* is a whole number that does not contain a decimal point.

SYNTAX

```
[+]n
-n
```

DESCRIPTION

n Specifies a signed or unsigned whole number.

An integer constant contains only the decimal digits 0 through 9 and a leading plus or minus sign.

Examples:

Correct	Incorrect	Explanation
`0`	`-1.`	Decimal point not allowed
`+230`	`45,678,732`	Commas not allowed
`1987530`		
`-03967`		

Real Data

A *single precision real constant* is a whole number, a fractional number, or a number that contains an integer and fractional portion. In their simplest form, real constants contain a decimal point, but they can be represented in a form resembling scientific notation with a signed or unsigned exponent.

When a real data constant is used in a double precision context, e.g., when assigned to a double precision variable or within a double precision expression, the full source precision (to machine double precision accuracy) is used, regardless of the presence of a "D" exponent. See note 10. on page 3-8 for details.

SYNTAX

[+]*n*.*d*[E*s*]
-*n*.*d*[E*s*]

DESCRIPTION

where *n.d* specifies a signed or unsigned real number.

n Specifies an optional integer portion.

d Specifies an optional fractional portion.

. A decimal point must be present if the scientific notation form is not used. The decimal point is optional if E*s* is used, in which case the implied decimal point is to the right of *n*.

where E*s* when used, is the number represented in a form resembling scientific notation.

E Specifies a decimal point shift, *s* follows.

s Specifies a signed or unsigned one- or two-digit integer number denoting a power of 10. The exponent indicates the decimal point is to be shifted *s* places (positive *s* a right shift, negative *s* a left shift). *s* cannot be omitted but can be zero.

Only the digits 0 through 9, a decimal point, an E, and a plus or minus sign are valid characters for real constants.

Example:

```
30.702E5                  is   3070200.000000
-.5171848261204058273E-3  is   -0.000517184
```

As many digits as desired can be written in the fractional portion of a real constant, but only the most significant digits are used. The fractional portion of a real constant is accurate to 6+ digits; that is, the sixth most significant digit is accurate, whereas the seventh is sometimes accurate depending on the value assigned to the data value.

The IEEE floating-point format specifies a single-precision signaling NaN (Not-a-Number) bit pattern. When such a bit pattern is encountered as an operand, an exception is generated. Thus, NaN is useful for detecting uses of uninitialized variables. The `NAN$` constant intrinsic may be used to initialize or assign single-precision real variables the single-precision NaN bit pattern (`0x7FBFFFFF`). See the **`nan$(3F)`** man page for further information.

An executing program receives a floating-point exception, `SIGFPE`, resulting from use of a NaN, with code `(siginfo_t->si_code & FPE_FLTINV)`. Assignment of NaN may or may not generate this exception, but arithmetic operations always do.

The Fortran compiler may receive a NaN floating-point exception and abort with an error message during compilation for two reasons. The first reason is source code that specifies a NaN bit pattern via a hexadecimal or other constant to be used as a floating-point constant. Use the appropriate `NAN$` intrinsic instead of a hard-coded bit pattern. The second reason is through compiler transformation of code containing a `NAN$` intrinsic. A NaN exception thus generated produces an error message containing the string "during constant folding," indicating that Concurrent Fortran's CCG optimizer has determined that the code contains an arithmetic operation that would use a NaN value. The compiler reports the source file line number containing the operation that caused the exception. Examine the source for an operation involving a variable that may contain NaN. For information on possible program optimizations, please see the "Program Optimization" Chapter of the *Compilation Systems Volume 2 (Concepts)* manual.

Examples:

Correct	Incorrect	Explanation
`-5.42`	`9770`	Decimal point missing
`4.5E-3`	`4,100.`	Comma not allowed
`.103E+5`	`$99.95`	Special character $ not allowed
`-42E10`	`4.3E.5`	Exponent must be integer
`56700.00E0`		
`0.007826`		
`+8410.`		

Double Precision Data

Double precision numbers permit a greater degree of accuracy than single precision real constants because they are stored with a 52-bit mantissa in IEEE.

SYNTAX

[+] *n.d*D*s*
−*n.d*D*s*

DESCRIPTION

where *n.d* specifies a signed or unsigned real number.

n Specifies an optional integer portion.

d Specifies an optional fractional portion. A double precision constant is represented in scientific notation form.

. The decimal point is optional; the implied decimal point is to the right of *n*.

where D*s* designates scientific notation form.

D Specifies the constant is double precision

s Specifies a decimal point shift. It is signed or unsigned, one to seven-digit integer number denoting a power of 10. The exponent indicates the decimal point is to be shifted *s* places (positive *s* a right shift, negative *s* left shift). The *s* cannot be omitted, but can be zero.

Only the digits 0 through 9, a decimal point, a D, and a plus or minus sign are valid characters for double precision constants.

Example:

30.702D5	is	3070200.000000
-.5171848261204058273D-3	is	-0.000517184

As many digits as desired can be written in the fractional portion of a double precision constant, but only the most significant digits are used. The fractional portion of a double precision constant is accurate to 15+ digits in IEEE; that is, the sixteenth most significant digit is accurate, whereas the seventeenth is sometimes accurate depending on the value assigned to the data value.

The IEEE floating-point format provides a double-precision signaling NaN bit pattern. The constant intrinsic DNAN$ generates this bit pattern (0x7FF7FFFFFFFFFFFF) at compile time. DNAN$ may be used to initialize a double-precision real variable through a DATA or assignment statement. See the **dnan$(3F)** man page and the preceding section describing the REAL data format for further information on using this intrinsic.

Examples:

```
 59032D+5
-.78156390273D0
2D-3
+25.56000714283D0
```

Complex Data

A *complex constant* is a pair of integers or real numbers or any mixture of the two. The pair of numbers is enclosed in parentheses and separated by a comma. Each number must be a positive or negative number. The first number is the real part of the complex number and the second number is the imaginary part.

Each number in the pair of numbers is converted to a single precision number with a 23-bit mantissa before being stored internally.

The IEEE floating-point format provides a single-precision signaling NaN bit pattern. The constant intrinsic `CNAN$` generates this bit pattern in complex format (`0x7FBFFFFF`, `0x7FBFFFFF`) at compile time. `CNAN$` may be used to initialize a single-precision complex variable through a `DATA` or assignment statement. See the **`cnan$(3F)`** man page and the preceding section describing the `REAL` data format for further information on using this intrinsic.

SYNTAX

(*cr,ci*)

DESCRIPTION

cr,ci Specify signed or unsigned numbers. Blanks can precede and follow each number.

Examples:

```
(5, 7.6)
(1.346,+52.01)
(3, 200)
(289346252E,9.2)
(0.,.1)
(-7.0, 6.19)
(7.1E2, 8.2)
```

A symbolic name cannot appear as one of the numbers of the pair. The constant is interpreted to mean:

*cr + ci * i*

where *i* equals the square root of -1. Thus, the following complex constants have values as indicated:

```
(1.34,52.01)            is 1.34 + 52.01i
(98344.,0.34452E+02)    is 98344.0 + 34.452i
(-1.,-1000.)            is -1.0 -1000.0i
```

Double Complex Data (H)

A *double complex constant* is a pair of numbers, at least one of which is double precision. The pair of numbers is enclosed in parentheses and separated by a comma. Each number must be a positive or negative number. The first number is the real part of the double complex number and the second number is the imaginary part.

Each number in the pair of numbers is converted to a double precision number with a 52-bit mantissa in IEEE before being stored internally.

The IEEE floating-point format provides a double-precision signaling NaN bit pattern. The constant intrinsic `ZNAN$` generates this bit pattern in double-precision complex format (`0x7FF7FFFFFFFFFFFF`, `0x7FF7FFFFFFFFFFFF`) at compile time. `ZNAN$` may be used to initialize a double-precision complex variable through a `DATA` or assignment statement. See the **`znan$(3F)`** man page and the preceding section describing the `REAL` data format for further information on using this intrinsic.

SYNTAX

(*cr*,*ci*)

DESCRIPTION

cr,ci Specify real or double precision constants. Blanks can precede and follow each number.

Examples:

```
(1.346,+52.01D3)
(289346252D,9.2D-2)
```

A symbolic name cannot appear as one of the numbers of the pair.

Logical Data

A *logical constant* is one of two fixed values:

```
.TRUE. or .FALSE.
```

representing the Boolean values “true” and “false”, respectively. The periods in each constant must be present to distinguish the constants as fixed values rather than symbolic names.

Variables, arrays, and functions can be defined as logical in specification statements. A value of one represents “true” and a zero value represents “false”. Only the least-significant bit of logical variables is used to determine the truth value when performing logical operations with the **-V** or **-VAX** compatibility options. When using the **-Qlogical_true_is_nonzero** option, any non-zero value represents “true”. For more information on the LOGICAL data type, see “Implementation of the LOGICAL Data Type (H)” on page 3-22.

Character Data

A *character constant* is a string of zero or more characters delimited by single or double quote marks. The character string contains any ASCII character.

SYNTAX

'*c1c2c3c4...cn*'

"*c1c2c3c4...cn*"

DESCRIPTION

c1c2c3c4...cn — Specifies a character string containing ASCII characters. The length of a character constant can be zero or null, e.g.,

“”

The blank character is a valid character in the string.

A non-printable character (e.g., a tab character or backspace character) is stored as itself internally.

Single or double quote marks that delimit a string are not a part of the string and are not counted as part of the string’s length. If single quotes delimit the string, use two single quotes side by side within the string to have one single quote appear as part of the string. If double quotes delimit the string, use two double quotes side by side within the string to cause one double quote to be stored as part of the string. Each pair of consecutive single quotes or double quotes counts as one character in the string and occupies only one position in the string.

The length of the character string is the count of the number of characters in the string. Conceptually, each character in the string also has a position that is numbered consecutively 1, 2, 3, etc.; the number indicates the sequential position of a character in the string beginning at the left and proceeding to the right.

A substring is a contiguous portion of a character string.

Table 2-6. Examples of Character Constants

Correct	Length of Constant
'␣'	1
'THE␣ANSWER␣CAN"T␣BE␣DETERMINED'	30
'A␣+␣16'	6
“RESULT:␣␣”	9
'1234567'	7
'AB63X9YZ[$<*'	12

Incorrect	Explanation
`'RESULT UNDEFINED!`	No closing single quote
`'YESTERDAY'S TALLY WAS:'`	Matching single quote missing

Hollerith Data

A *Hollerith constant* is a non-empty string of characters. Hollerith constants may appear only in a DATA statement, a FORMAT statement, and during assignment to numeric variables.

SYNTAX

*w*H*h1h2h3...hw*

DESCRIPTION

w	Specifies an unsigned, non-zero integer indicating the number of characters in the Hollerith string.
h1h2h3...hw	Specifies the characters in the constant. A blank is a significant character in the Hollerith string. A character in a Hollerith string can be any ASCII character.

If the Hollerith data does not fill the sequence of storage locations reserved for the data, the compiler fills the trailing storage locations with blanks. If the data overflows the assigned storage locations, any additional characters to the right are lost.

When a numeric entity is defined with a Hollerith value, the entity and its associated entities should not be used in an arithmetic context.

Example:

```
1H␣
17HTODAY'S␣TOTAL␣IS:
```

If Hollerith constants are used in output FORMAT statements, *w* must indicate the exact number of characters in the Hollerith string, or the output format will be interpreted incorrectly. For example,

```
100 FORMAT (18HTHE RESULT IS:,F6.2)
```

results in the following Hollerith string and a format error will result:

```
THE RESULT IS:,F6.
```

Variables

A *variable* is a symbolic name with a specific data type. The name of a variable refers to a location in memory where a data value is stored or is to be stored. Referencing a variable means that the variable is used in a context where its value is needed.

The *data type* of a variable defines the type of data the variable will contain, limits the storage available for the variable, and, for numeric variables, defines the precision of the value.

Explicit type statements define the data type and length of a variable. A variable can be explicitly defined only once. An `IMPLICIT` statement permits symbolic names beginning with a specific letter or permits a range of such letters to be designated as belonging to a particular data type. Explicit type statements take precedence over `IMPLICIT` definitions.

If the variable is not defined explicitly and if an `IMPLICIT` statement does not apply, the following rules are applicable:

- All variable names beginning with the letter `I`, `J`, `K`, `L`, `M`, or `N` are defined to be of type `INTEGER`.
- All variable names beginning with a letter `A-H` or `O-Z` are defined to be of type `REAL`.

Examples:

Implicit Integer Variables	Implicit Real Variables
`I`	`X`
`J1`	`DAYS TOTAL`
`LINECOUNT`	`COUNT`

Arrays

An *array* is a sequence of variables that have the same symbolic name and data type.

A subscript appended to the array name distinguishes each element or member of the array, and the array element name refers to a specific storage location. When the value of an array element is referenced, the array name and its subscript are written together; the subscript is enclosed in parentheses and can be an arithmetic expression. Array element names (i.e., those written with subscripts) must have a subscript for each dimension of the array.

Declaring an Array

Arrays used in a program unit must be declared at the beginning of the program unit using a DIMENSION, AUTOMATIC, STATIC, COMMON, DATAPOOL, or an explicit type statement. The array declaration defines the upper and lower bounds of the array, the data type of the array, and the number of dimensions in the array.

SYNTAX

keyword name (*d1* [, *d2*, ..., *d7*]) [,...]

DESCRIPTION

keyword	Specifies DIMENSION, AUTOMATIC, STATIC, COMMON, DATAPOOL, or an explicit type statement keyword.
name	Specifies the symbolic name of the array.
d1,d2...,d7	Specifies the dimensions of the array. Each *d* has one of two forms: *eu* or *el*:*eu*
el	Specifies unsigned integer constants or integer constant expressions and designates the lower bound of a dimension. If the lower bound is omitted, it is assumed to be 1.
eu	Specifies unsigned integer constants or integer constant expressions and designates the upper bound of a dimension. The lower and upper bound specifications are positive, negative or zero, but the upper bound must not be less than the lower bound. If the upper bound of the rightmost dimension is an asterisk (*), the array is an assumed-size array (see Chapter 8).

An array has from one to seven dimensions. The size of an array is equal to the product of the dimensions of the array.

Dimension expressions in an array declarator must not contain a variable name, an array element name, or a function reference. Only arithmetic expressions containing numeric constants or symbolic names of integer constants are permitted in array declarators. The result of the dimension expression must be an integer.

Arrays are stored linearly in main memory. Table 2-7 illustrates array storage. Note that the leftmost subscript varies most rapidly. Elements of array A(3,3,2) are stored as follows:

Table 2-7. Array Storage

Memory Location	Array Element
1st	`A (1,1,1)`
2nd	`A (2,1,1)`
3rd	`A (3,1,1)`
4th	`A (1,2,1)`
5th	`A (2,2,1)`
6th	`A (3,2,1)`
7th	`A (1,3,1)`
8th	`A (2,3,1)`
9th	`A (3,3,1)`
10th	`A (1,1,2)`
11th	`A (2,1,2)`
12th	`A (3,1,2)`
13th	`A (1,2,2)`
14th	`A (2,2,2)`
15th	`A (3,2,2)`
16th	`A (1,3,2)`
17th	`A (2,3,2)`
18th	`A (3,3,2)`

The data type of an array is implied by the first letter of the name if the array is not declared in an explicit type statement. If numeric data of a different type is assigned as the value of a numeric array element name, the data is converted to the data type of the array. An array with a given name may be declared only once in a program unit.

The number and size of dimensions in one array declarator can be different from the number and size of dimensions in another array declarator that is associated by common, equivalence, or argument association.

Examples:

```
DIMENSION I(0:9,0:9), J(0:4,0:4), K(-5:5)
INTEGER VECTOR (10*10), TABLE (25, 25)
COMMON A(3,3,3)
```

Referencing an Array

The value of an array element is referenced by using the array name with the subscript that pinpoints the appropriate element in the array; thus, the reference, known as an *array element name*, must include the array name and a subscript for each dimension of the array.

SYNTAX

array-name (*s1* [, *s2*, ..., *s7*])

DESCRIPTION

array-name	Specifies a symbolic name.
s1, s2, ...,s7	Subscripts specified as integer constants or as subscript expressions that are explicitly converted to integer values.

An unsubscripted array name is used as follows:

- In a COMMON, EQUIVALENCE, DATA, SAVE, AUTOMATIC, STATIC, DATAPOOL, or explicit type statement.
- As a dummy or actual argument in an argument list.
- In an array declarator in a COMMON, DIMENSION, or explicit type statement.
- In an I/O statement in an input or output list, as a unit specifier for an internal file, or as a format specifier (providing the array is not an assumed-size dummy array).

An unsubscripted array name designates the whole array. The appearance of the unsubscripted array name implies that the number of values to be processed is equal to the number of elements in the array, and that the elements of the array are to be taken in sequential order, beginning with the first element of the array.

In EQUIVALENCE statements and in assignment statements, using an unsubscripted array name references only the first element of the array and no more.

A subscript expression contains constants, symbolic names of constants, function references, variables, or array element names. Any symbolic names used in the subscript expression must be defined. Evaluation of a subscript does not alter the value of any other subscript.

Examples:

```
A(1)
I(-3,2,4)
J(0,4)
POS(I,I+1)
X(1,2,1,1,2)
VECTOR(2 * INT(Q))
```

Character Substrings

A *character substring* is a contiguous portion of a character string value. A *substring* is a single character in a character string, a subset of contiguous characters in a character string, or the entire character string in duplicate. Within a character context, a substring reference refers to a substring of characters within a character string.

Substring Referencing for Variables

SYNTAX

name (*e1*:*e2*)

DESCRIPTION

name Specifies a character variable.

e1 Specifies a substring expression that produces a positive integer number. It specifies the leftmost character position of a substring.

e2 Specifies a substring expression that produces a positive integer number. It specifies the rightmost character position of a substring.

Example:

If variable `CHARS` is declared to be of data type character and has the value:

```
ABCDEFGHI
```

then the following will reference the substring `DEFG`:

```
CHARS(4:7)
```

If *e1* or *e2* is an expression, then numeric variables, array element names, or function references can appear in the expression.

If *e1* is omitted, 1 is used. If *e2* is omitted, the compiler uses the number denoting the length of the parent character string as the default value for the substring reference. Note that a substring reference of the form:

```
CHARS(:)
```

is equivalent to using the variable `CHARS` with no substring reference.

The values of *e1* and *e2* must be such that:

```
1 <= e1 <= e2 <= length
```

where *length* is the total number of characters in the parent character string.

Substring Referencing for Array Elements

SYNTAX

name (*d1*[,...])(*e1*:*e2*)

DESCRIPTION

name Specifies the array element name.

d1,... Specifies the subscript portion denoting which element of the array to reference.

e1, e2 Specify substring expressions as defined in the previous section.

For example, if CHARS(1) contains: "ALL COWS EAT GRASS" then the following substring reference produces the sequence COWS EAT GRASS:

```
INTEGER LM, RM
LM = 5
RM = 18
WRITE(6,*) CHARS(1)(LM:RM)
```

Initialization of Variables and Arrays at Compile Time

When a source program has been compiled and is ready for execution, all variables and array elements are undefined (i.e., have an unknown value) unless the entity is:

- Given a value in a DATA statement or in a declaration statement
- On the left-hand side of an assignment statement
- Passed as a subroutine or function argument
- In the I/O list of a read
- Referenced by an I/O keyword
- In a COMMON block
- Equivalenced to an initialized entity
- In a statement function

A character entity is defined when all characters in that entity are defined. If the length of a character value is longer than the length of the character entity to which it is assigned, additional characters at the right are ignored when the value is assigned (i.e., the character value is truncated). If the length of a character value is shorter than the length of the character entity to which it is assigned, blanks are added to the right of the character value to fill the entire character entity (i.e., the entity is padded to the right with blanks).

A variable, array element, or substring is initially defined only once in a program unit; if its value changes later, the entity is redefined or its value becomes undefined. Any

variable, array element, or substring can be initially defined (i.e., defined in a DATA or declaration statement) except for:

- A dummy argument.
- A variable in a function subprogram whose name is also the name of a function subprogram, or the name of an entry point in the function subprogram.
- A variable declared as AUTOMATIC.

If two entities are associated (i.e., share the same memory space), only one is initially defined in a DATA statement in the same source program. Refer to Chapter 4 for more information on the DATA statement.

Arguments

Arguments permit one program unit to communicate values to another. Arguments appear to the right of a function, subroutine name or entry point and are enclosed in parentheses and are separated by commas.

SYNTAX

name (*a1, ..., an*)

DESCRIPTION

name Specifies the function, subroutine name or entry point.

a1,...,an Specifies a list of actual arguments.

Actual arguments are constants, expressions or symbolic names appearing in the argument list of a function reference or subroutine call. The values of the actual arguments are passed to the function or subroutine, and the values are associated with *dummy arguments* defined for the function or subroutine subprogram. Actual and dummy arguments are explained further in Chapter 8.

Definition Status

When a program unit begins execution, the value of a variable is either defined or undefined. A variable is initially *defined* if a value is given to it in a DATA statement. A variable is defined after program execution begins by using assignment or input statements.

An *undefined* variable has no predictable value; all variables, unless initially defined in a DATA statement, are undefined when a program begins execution with the first executable statement of the program. Once a variable is defined (i.e., contains a value), the value does

not change unless the value is destroyed (becomes undefined) or is redefined to contain a different value.

When variables have been declared AUTOMATIC in a subroutine or function, they become undefined when a RETURN or an END statement is encountered.

Association of Symbolic Names

Symbolic names that are associated refer to the same program entity or memory locations. Association permits an entity defined in one program unit to be referenced later by another program entity of different type. Entities are associated by COMMON statements, EQUIVALENCE statements, ENTRY statements, or through the argument list of a function reference, subroutine call, or ENTRY statement.

In the following example, IVAR and RVAR are associated:

```
SUBROUTINE SUB1
COMMON /COM/ IVAR
INTEGER IVAR
...
END

FUNCTION FUNC1 ()
COMMON /COM/ RVAR
REAL RVAR
...
END
```

3
Expressions and Assignment Statements

3
Expressions and Assignment Statements

Expressions and Statements Overview

The Fortran language permits arithmetic, character, relational, and logical expressions. An *expression* is composed of operands, operators, and, if necessary, parentheses.

Fortran expressions are commonly used in assignment statements because they produce a single numeric, character, or logical value that can be assigned as the value of a variable or array element name. Expressions are also used as dimension bounds in array declarators; as subscripts; as substring bounds in substring references; as arguments in function references and subroutine calls; as control parameters in `DO`, `FOR`, `LOOP`, `DO WHILE`, `DO-UNTIL`, and `IF` statements; and in control information lists and I/O lists in input and output statements.

Assignment statements are executable statements that assign a value to a variable or an array element. The four kinds of assignment statements are arithmetic, character, logical, and statement label assignments.

Arithmetic Expressions and Assignments

Arithmetic expressions produce numeric values that are used in an arithmetic context, such as in an arithmetic assignment statement. Evaluation of an arithmetic expression results in a single numeric value.

Simple and Compound Arithmetic Expressions

A *simple arithmetic expression* consists of one operand in an arithmetic context. The operand can be a numeric constant or the symbolic name of a constant, a numeric variable name, a numeric array element name, or a numeric function reference.

A *compound arithmetic expression* consists of two or more numeric operands, connected by arithmetic operators, appearing in an arithmetic context. Any of the following entities are valid numeric operands for compound arithmetic expressions:

- Numeric constants or symbolic names of constants
- Numeric variables
- Numeric array elements

- Numeric function references
- Numeric subexpressions nested within the compound expression

Any combination of the preceding operands can appear in an arithmetic expression.

Variables and array element names must be defined (i.e., have a value) before they can be used in arithmetic expressions.

The following kinds of data can appear in arithmetic expressions:

Binary data
Octal data
Hexadecimal data
Hollerith data
Integer data
Real data
Double precision data
Complex data
Double complex data

Constant Arithmetic Expressions

A *constant expression* is a simple or compound arithmetic expression that contains only constants. An expression used in a specification statement (e.g., as an array declarator in a `COMMON`, `DATAPOOL`, `AUTOMATIC`, `STATIC`, `DIMENSION`, or explicit type statement) must be an integer constant expression; i.e., variable names, array element names, or function references are not permitted, and the resulting data type of the expression must be an integer.

Character Constant Expression

A *character constant expression* is one in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed within parentheses. The character operator "`//`" is allowed for string concatenation. The intrinsic function, `CHAR`, when given an integer constant parameter, generates a character constant of length one at compile time; it may be used to form a character constant expression. Character constant expressions are the only character expressions that may be used to set character `PARAMETER` values.

Arithmetic Operators

Arithmetic expressions are formed using unary and binary operators. *Unary* (meaning one) operators have only one operand, which appears to the right of the operator. *Binary* (meaning two) operators have two operands; the operator is written between the two operands. Table 3-1 lists the unary and binary operators

.

Table 3-1. Unary and Binary Arithmetic Operators

Unary Operators	Representing
+	Identity
–	Negation
Binary Operators	**Representing**
* *	Exponentiation
*	Multiplication
/	Division
+	Addition
–	Subtraction

Precedence of Arithmetic Operators

Expressions containing two or more operators are interpreted based on a precedence hierarchy associated with arithmetic operators. Table 3-2 gives the precedence hierarchy for arithmetic operators. Operators of equal precedence are grouped together.

Table 3-2. Precedence Hierarchy of Arithmetic Operators

Operation	Operator	Precedence
Exponentiation	* *	First (Highest)
Multiplication	*	Second
Division	/	Second
Addition	+	Third
Subtraction	–	Third
Unary plus	+	Third
Unary minus	–	Third

Whenever two or more operators of equal precedence appear in sequence, they are evaluated from left to right. However, if two or more subexpressions with exponents occur in sequence, they are evaluated from right to left. Parentheses in the following expressions indicate how the expression is evaluated for operators of equal precedence:

```
X-Y+Z     is interpreted as (X-Y)+Z
X/Y*Z     is interpreted as (X/Y)*Z
X**Y**Z is interpreted as X**(Y**Z)
```

Parentheses are used in an expression to override the precedence rules:

```
X - (Y + Z)
X / (Y * Z)
```

A negative exponent of type integer, real, double precision, complex, or double complex is permitted and is written in the form:

*x1 ** x2*

where *x1* and *x2* can be integer, real, double precision, complex, or double complex numbers, and *x2* is less than 0. The preceding exponentiation is interpreted as follows:

`1 / (`*x1* `** ABS(`*x2*`))`

The rules of division apply in the above interpretation.

Examples of Arithmetic Expressions

Parentheses are used to explicitly indicate the manner in which the expression is to be evaluated.

Expression

```
4 + 6 * 10 / 2 ** 2 - 1 = 18
```

Evaluation

```
(4 + ((6 * 10) / (2 ** 2))) - 1 = 18
```

Parentheses are also used to preclude the use of some algebraic transformations by the compiler.

Expression

```
A + B + C
```

Evaluation

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

which is mathematically equivalent (but is not, in general, computationally equivalent). Using:

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

constrains the pairing of operands to achieve the desired result.

An entire expression or any subexpression is signed or unsigned if enclosed in parentheses:

```
+(4 + 6 * 10 / 2 ** 2 - 1) = 18
-(4 + 6 * 10 / 2 ** 2 - 1) = -18
-(4 + 6 * 10 / 2 ** 2) - 1 = -20
```

Expression

```
-1 ** 3
```

Evaluation

```
-(1 ** 3)
```

Any operation that is not mathematically defined (such as division by zero) is not permitted during program execution.

Exponentiation Rules

The following rules apply in expressions containing exponents:

An exponent and its base is an entity that is an expression in parentheses.

- Negative numbers cannot be raised to non-integer exponents.
- Raising a zero base to a zero power or to a negative power is prohibited.

NOTE: These computations are not mathematically defined.

Data Type Conversions (Mixed Modes)

If the operands in an arithmetic expression are all of one arithmetic data type, the value of the expression has that data type and is defined as an integer, real, double precision, complex, or double complex expression.

A *mixed-mode arithmetic expression* contains a mixture of operands with different numeric data types. In a mixed-mode expression, as each subexpression within the expression is evaluated, the resulting mode is that of the operand with the highest ranking data type. The precedence of the data type is as follows:

Data Type	Precedence
`DOUBLE COMPLEX`	First (Highest)
`COMPLEX`	Second
`DOUBLE PRECISION`	Third
`REAL`	Fourth
`INTEGER`	Fifth
`INTEGER*2`	Sixth
`INTEGER*1`	Seventh
Hollerith	Eighth

The rules for determining the data type for the value of an expression are given below. A section of notes follows the explanations.

Resulting Data Type	Explanation
`INTEGER`	The value of an expression or subexpression is of type `INTEGER` (1) if all operand(s) are of type `INTEGER` or (2) if the operand(s) are a mixture of `INTEGER` entities and entities of a lower rank. Hollerith data is always treated as integer data in an assignment. See Notes 1 and 2 below.
`REAL`	The value of an expression or subexpression is of type `REAL` (1) if all operand(s) are of type `REAL` or (2) if the operand(s) are a mixture of `REAL` entities and entities of lower rank. See Notes 1, 3, 4, and 5 below.
`DOUBLE PRECISION`	The value of an expression or subexpression is of type `DOUBLE PRECISION` (1) if all operand(s) are of type `DOUBLE PRECISION` or (2) if the operand(s) are a mixture of `DOUBLE PRECISION` entities and entities of lower rank. See Notes 1, 3, 4, and 5 below.
`COMPLEX`	The value of an expression or subexpression is of type `COMPLEX` (1) if all operand(s) are of type `COMPLEX` or (2) if the operand(s) are a mixture of `COMPLEX` entities and entities of lower rank. In expressions containing `COMPLEX` numbers, integer and double precision numbers are converted to real values which are then converted to complex numbers. The real value is the real portion of the complex number, and an imaginary part of `0.0` is supplied.
`DOUBLE COMPLEX`	The value of an expression or subexpression is of type `DOUBLE COMPLEX` (1) if all operand(s) are of type `DOUBLE COMPLEX` or (2) if the operand(s) are a mixture of `DOUBLE COMPLEX` entities and entities of lower rank. In expressions containing `DOUBLE COMPLEX` numbers, integer, and real numbers are converted to double precision values which are then converted to double complex numbers. The double precision value is the real portion of the double complex number, and an imaginary part of `0.0D0` is supplied.

Notes:

1. During expression evaluation, the data type of intermediate results and of the final result is the data type of the operand with the highest precedence.

   Example:

   ```
   ((6 + 5) + 3.2) + 4.5D3
   ```

   is evaluated as follows:

Subexpression	Intermediate Result	Data Type of Intermediate Result
`6 + 5`	`11`	Integer
`6 + 5 + 3.2`	`14.200000`	Real
`6 + 5 + 3.2 + 4.5D3`	`4514.20000000000`	Double precision

2. If an integer division occurs, any fractional part is truncated. Note that:

   ```
   1 / 2 * 3.0
   ```

   results in a value of `0.0` because the integer division is performed first according to the rules of precedence for operators. The conversion to a real number occurs when the real multiplication of `0 * 3.0` is performed.

3. If a real, double precision, complex, or double complex number is raised to an integer power, the integer is not converted. For example, use:

   `A ** 2` and not `A ** 2.0`

   where `A` is a real number.

4. Any integer numbers that are converted to real, double precision, complex or double complex form are given a fractional part of zero.

5. Division and multiplication using real or double precision numbers are subject to round-off errors.

   Example:

   A. 2./3. + 1./3. can be less than 1.0.

   B. 10 * 0.9 may not be exactly equal to 9.

6. Expressions appearing as subscripts in an array reference are independent of the expression in which the array appears. The subscript expressions are evaluated in their own mode and neither affect the data type of the outer expression nor are affected by it.

7. Expressions appearing as arguments in function references are evaluated in their own mode independently of the expression in which the function reference appears. For example, if `X` is of type real and `J` is of type integer and the `INT` (conversion-to-integer) function is used, the expression `INT(X+J)` is an integer expression, and `X+J` is a real expression.

Expressions as arguments can affect the mode of the function result and, thus, of the expression in which the result is used. For example, for many intrinsic functions, if the generic name of the function is used, the data type of the function's result is the data type of the argument(s) processed for the function. See Chapter 9 for more information on Fortran intrinsic functions.

8. Values of expressions and subexpressions cannot exceed the limits associated with the data type of the expression. For example, a real intermediate result cannot be outside the domain of values for real numbers.

9. Hexadecimal, octal, and binary constants assume data types and sizes based on the context of the expression. When paired with or assigned to real operands, these constants are treated as real; when paired with or assigned to integer operands, these constants are treated as integer. The size of the constant is also determined by the context in the same way. When there is no context to determine the data type and size for the constant, as with an argument in a CALL statement, the type defaults to INTEGER *4.

10. Real constants specified to double precision significance have their accuracy preserved if the constant is used in a double precision context.

    Example:

    ```
    DOUBLE PRECISION DP
    DATA DP /1.234567890123456/
    WRITE (6,*) DP
    DP = 2.345678901234567
    WRITE (6,*) DP
    ```

    would produce:

    ```
    1.234567890123456
    2.345678901234567
    ```

    as output rather than truncating the real constants to 6 digits of accuracy before use. Imprecisely representable real constants such as 0.1 also have double precision accuracy if used in a double precision context.

11. All compile-time real constant arithmetic is performed with double precision accuracy.

Table 3-3 gives examples illustrating mixed-mode expressions for addition, subtraction, multiplication, and division; notation in the first column represents any of the four operations. Table 3-4 gives examples illustrating mixed-mode expressions using exponentiation. Integer, Real, Double Precision, Complex, and Double Complex operands are represented by `I`, `R`, `D`, `C`, and `Z`, respectively.

Table 3-3. Examples Illustrating Mixed-Mode Expressions for +, -, * and /

Expression x1 op x2	Interpretation	Resulting Data Type
`I1 + R2`	`REAL(I1) + R2`	Real
`R1 + I2`	`R1 + REAL(I2)`	Real
`D1 + I2`	`D1 + DBLE(I2)`	Double Precision
`D1 + R2`	`D1 + DBLE(R2)`	Double Precision
`I1 + D2`	`DBLE(I1) + D2`	Double Precision
`R1 + D2`	`DBLE(R1) + D2`	Double Precision
`C1 + I2`	`C1 + CMPLX(REAL(I2),0.)`	Complex
`C1 + R2`	`C1 + CMPLX(R2,0.)`	Complex
`C1 + D2`	`C1 + CMPLX(REAL(D2,0.))`	Complex
`I1 + C2`	`CMPLX(REAL(I1),0.) + C2`	Complex
`R1 + C2`	`CMPLX(R1,0.) + C2`	Complex
`D1 + C2`	`CMPLX(REAL(D1),0.) + C2`	Complex
`Z1 + I2`	`Z1 + DCMPLX(DBLE(I2),0.0D0)`	Double Complex
`Z1 + R2`	`Z1 + DCMPLX(DBLE(R2),0.0D0)`	Double Complex
`Z1 + D2`	`Z1 + DCMPLX(D2,0.0D0)`	Double Complex
`Z1 + C2`	`Z1 + DCMPLX(C2)`	Double Complex
`I1 + Z2`	`DCMPLX(DBLE(I1),0.0D0) + Z2`	Double Complex
`R1 + Z2`	`DCMPLX(DBLE(R1),0.0D0) + Z2`	Double Complex
`D1 + Z2`	`DCMPLX(D1,0.0D0) + Z2`	Double Complex
`C1 + Z2`	`DCMPLX(C1) + Z2`	Double Complex

Table 3-4. Examples Illustrating Mixed-Mode Expressions for **

Expression x1 op x2	Interpretation	Resulting Data Type
`I1 ** R2`	`REAL(I1) ** R2`	Real
`R1 ** I2`	`R1 ** REAL(I2)`	Real
`D1 ** I2`	`D1 ** DBLE(I2)`	Double Precision
`D1 ** R2`	`D1 ** DBLE(R2)`	Double Precision
`I1 ** D2`	`DBLE(I1) ** D2`	Double Precision
`R1 ** D2`	`DBLE(R1) ** D2`	Double Precision

Table 3-4. Examples Illustrating Mixed-Mode Expressions for ** (Cont.)

Expression x1 op x2	Interpretation	Resulting Data Type
`C1 ** I2`	`C1 ** CMPLX(REAL(I2),0.)`	Complex
`C1 ** R2`	`C1 ** CMPLX(R2,0.)`	Complex
`C1 ** D2`	`C1 ** CMPLX(REAL(D2,0.))`	Complex
`I1 ** C2`	`CMPLX(REAL(I1),0.) ** C2`	Complex
`R1 ** C2`	`CMPLX(R1,0.) ** C2`	Complex
`D1 ** C2`	`CMPLX(REAL(D1),0.) ** C2`	Complex
`Z1 ** I2`	`Z1 ** DCMPLX(DBLE(I2),0.0D0)`	Double Complex
`Z1 ** R2`	`Z1 ** DCMPLX(DBLE(R2),0.0D0)`	Double Complex
`Z1 ** D2`	`Z1 ** DCMPLX(D2,0.0D0)`	Double Complex
`Z1 ** C2`	`Z1 ** DCMPLX(C2)`	Double Complex
`I1 ** Z2`	`DCMPLX(DBLE(I1),0.0D0) ** Z2`	Double Complex
`R1 ** Z2`	`DCMPLX(DBLE(R1),0.0D0) ** Z2`	Double Complex
`D1 ** Z2`	`DCMPLX(D1,0.0D0) ** Z2`	Double Complex
`C1 ** Z2`	`DCMPLX(C1) ** Z2`	Double Complex

Arithmetic Assignments

An *arithmetic assignment* statement assigns a numeric, logical, character, or Hollerith value to a numeric variable or array element name. Arithmetic assignment statements are executable statements.

SYNTAX

name = e

DESCRIPTION

name Specifies a variable name or subscripted array element name.

e Specifies an arithmetic, logical or character expression, a variable, a subscripted array element name, a constant, the symbolic name of a constant, or a Hollerith value.

Any symbolic name appearing to the right of the equals sign must be defined (i.e., have a value). A symbolic name appearing to the left of the equals sign may or may not be defined. An expression or function reference cannot appear to the left of the equals sign in an arithmetic assignment statement except as an array subscript.

The equals sign does not imply equality but indicates that the value to the right of the equals sign is to be assigned as the value of the element to the left of the equals sign. Note that the following is valid:

```
J = J + 1
```

The preceding assignment statement causes the present value of variable J to be incriminated by 1 and the resulting value assigned as the new value for J. See also "Multiple Assignment Statements (H)" on page 3-31 and "Array Assignment Statements (H)" on page 3-33.

Examples:

```
B = 32.7
A = B
J = +I
Q(1) = Z ** 2 + N(L-J)
T32(1) = L / (34.,-0.003244) + (.0045,0)
PI = 4 * (ATAN(0.5) + ATAN(0.2) + ATAN(0.125))
```

If the data type of *name* and the value of *e* are different, the following conversion rules apply:

- If *name* is of data type `INTEGER`, a real or double precision value for *e* is truncated to integer form (any fractional part is lost). For a complex or double complex number, the imaginary part of the complex or double complex number is ignored and the real part of the complex or double complex number is truncated to integer form. If e is a character entity or a Hollerith value, the left most n characters of the entity, where n is the size in bytes of name, are places in the corresponding bytes of name. The value of each byte is the integer value defined by the `ICHAR` intrinsic function. If e is a logical entity, a verbatim copy of the bit pattern of e is placed right-justified into name.
- If *name* is of data type `REAL`, a double precision value for *e* is truncated to a real number, an integer value is converted to a real number with a fractional part of zero, and the real portion of a complex or double complex number is used (truncated to a real if double complex) while the imaginary portion is ignored.
- If *name* is of data type `DOUBLE PRECISION`, an integer value for *e* is converted to a double precision number and given a fractional part of zero, and a real value is extended to double precision form. For complex numbers the real portion of the complex number is used and the imaginary part is ignored.
- If *name* is of data type `COMPLEX`, an imaginary part of `0.0` is assigned for an integer, real, or double precision value of *e*. The real part of the complex number is derived as follows: a real number is used as is, a double precision number is truncated to a real number, and an integer is converted to a real number.
- If *name* is of data type `DOUBLE COMPLEX`, an imaginary part of `0.0D0` is assigned for an integer, real, or double precision value of e. The real part of the complex number is derived as follows: a real number is extended to

double precision, a double precision is used as is, and an integer is converted to a double precision number.

.SHIFT. and .ROTAT. Integer Operators (H)

The .SHIFT. and .ROTAT. operators treat the bits in a numeric or logical value as a bit string of 0's and 1's. .SHIFT. and .ROTAT. shift digits in a bit string to the left or right or rotate bits to the left or right within a string, respectively. The bit string is derived from an integer or logical expression or constant that appears as the left operand of the shift or rotate operation. The number of binary digits to shift or rotate is indicated as an INTEGER expression to the right of the shift or rotate operator.

The .SHIFT. and .ROTAT. operators have equal precedence with each other and have a higher precedence than the other numeric, arithmetic, character, and relational operators.

The .SHIFT. operator has the following form:

e .SHIFT. *i*

DESCRIPTION

e Specifies an integer or logical constant or expression.

i Specifies an integer constant or expression yielding a value between -32 and +32, inclusive, for INTEGER *4 integers, and -16 and +16, inclusive, for INTEGER *2 integers. The value of *i* specifies both the magnitude and direction of the shift. If *i* is positive, the direction of the shift is to the left; if *i* is negative, the direction of the shift is to the right.

The .SHIFT. operator denotes a logical shift with zero fill. The data type of the result of a logical shift operation is the same as that of *e*.

The .ROTAT. operator has the following form:

e .ROTAT. *i*

DESCRIPTION

e Specifies an integer or logical constant or expression.

i Specifies an integer constant or expression yielding a value between -32 and +32, inclusive, for INTEGER *4 integers, and between -16 and +16, inclusive, for INTEGER *2 integers. The value of *i* specifies both the magnitude and direction of rotation. If *i* is positive, the direction of the rotation is to the left; if *i* is negative, the direction of the rotation is to the right.

The data type of the result of a logical rotate operation is the same as that of *e*.

Examples:

The following shift and rotate operations have the indicated effect on the bit string and result in the specified hexadecimal value:

Expression	Result
`'01234567'X .SHIFT. 8`	`'23456700'X`
`'01234567'X .SHIFT. -8`	`'00012345'X`
`'01234567'X .ROTAT. 8`	`'23456701'X`
`'01234567'X .ROTAT. -8`	`'67012345'X`

Character Expressions and Assignments

Character expressions produce values that are strings of zero or more characters.

Character Expressions

Data values that include letters, numbers, or special characters are called *character strings* or *character data*. Any character capable of representation in the computer appears in a character string.

A *character expression* results in a value that is a character string. Evaluation of a character expression results in a single character value, or a string of zero or more characters that can be used in a character context such as a character assignment statement. Numeric and logical data or symbolic names are not permitted in character expressions or assignment statements.

A *simple character expression* consists of a single operand: one character constant or the symbolic name of a character constant, one character variable, one character array element name, or one character function reference.

A *compound character expression* consists of two or more operands connected by the character concatenation operator (`//`), appearing in a character context.

Any of the following entities are valid character operands for compound character expressions:

- Character constants or symbolic names of character constants
- Character variables
- Character array elements
- Character function references
- Character subexpressions nested within the compound expression

Any combination of the preceding operands can appear in a character expression.

Variables and array element names must be defined (i.e, have a value) before they are used in character expressions.

Character String Operations

Character string concatenation is an operation in which two or more character string operands are combined into a single string by appending the right operand to the left operand. The concatenation operator is the only character string operator and is used as follows:

```
'ABC' // 'DE'
```

Two slashes (//) denote the concatenation operation. The result of the previous concatenation is the character value:

```
ABCDE
```

When two or more concatenation operators are present in a character expression, the concatenation operations are evaluated from left to right.

Example:

```
'AB' // 'CD' // 'E'
```

is equivalent to:

```
('AB' // 'CD') // 'E'
```

Character Assignments

A *character assignment statement* assigns a character string value to a variable name or array element name. Character assignment statements are executable statements.

SYNTAX

name = *e*

DESCRIPTION

name Specifies a variable name or subscripted array element name.

e Specifies a character expression, a variable, a subscripted array element name, a character constant, or the symbolic name of a character constant.

Any symbolic name in *e* to be concatenated or whose value is to be assigned must be defined to be of type character and have a character value. A symbolic name appearing to

the left of the equals sign may or may not have a value, but it must have been defined in an `IMPLICIT` or explicit declaration statement to be of type character.

An expression or function reference cannot appear to the left of the equals sign in a character assignment statement except as a subscript or in a substring reference.

If the value of *e* is less than the size of *name*, blanks are appended to the right of the value of *e* to fill the extra character positions. If *e*'s value is greater than the size of *name*, additional characters are truncated at the right of the string when the value is assigned.

Substring references may appear on the left or right side of a character assignment. If a substring reference appears on the left, the assignment is made to the positions defined in the substring; characters to the left or right of the substring in the parent string are unaffected. Note that the truncation or padding of *e*'s value affects only the area defined by the substring reference.

If *name* refers to a symbolic constant declared as type `CHARACTER*(*)`, the length of the variable is set to the length of the character expression.

A string is considered defined if one character in one position of the string is defined. Only as much of *e*'s value as is needed must be defined for the value to be used in a character assignment.

See also "Multiple Assignment Statements (H)" on page 3-31 and "Array Assignment Statements (H)" on page 3-33.

Examples:

```
CHARACTER A*2 B*4
A=B
```

The first two characters in string `B` must be defined, but `B(3:4)` need not be defined.

```
CHARACTER *30 LCASE, UCASE, DIGITS, SPECHARS
CHARACTER BLANK
CHARACTER *150 ALLCHARS
. . .

DIGITS = "0123456789"
BLANK = 'b'
UCASE = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LCASE = 'abcdefghijklmnopqrstuvwxyz'
SPECHARS = '!"$''()*:+;,.=-'
ALLCHARS = UCASE // LCASE // DIGITS // BLANK // SPECHARS
```

Relational and Logical Comparisons and Assignments

Comparisons are performed using relational and logical operators in comparison expressions.

Relational Expressions

A *relational expression* compares the resultant values of two arithmetic expressions or of two character expressions. A mixture of character and arithmetic operands is not permitted. The arithmetic or character expression is a simple or compound expression.

Operands being compared are connected by one of the following binary operators:

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

The periods are part of each operator and must enclose the mnemonic. Relational operators have a lower precedence than arithmetic or character operators, and all relational operators are of equal precedence with each other. As a result, evaluation of a series of relational expressions proceeds from left to right in a statement. Parentheses are used to alter the order of evaluation.

Two relational operators cannot appear side by side in the same expression.

Variables and array element names must be defined (i.e., have a value) before they can be used in relational expressions.

The following kinds of data appear in relational expressions:

Binary data
Octal data
Hexadecimal data
Hollerith data
Integer data
Real data
Double precision data
Complex data
Double complex data
Character data

A relational expression results in one of two logical values: `.TRUE.` or `.FALSE.` If the relationship being compared exists, the result is true; if the relationship does not exist, the value is false.

SYNTAX

el op e2

DESCRIPTION

e1, e2 Specify operands to be compared.

If *e1* and *e2* are arithmetic expressions and if one operand is complex or double complex, the other operand is converted to complex or double complex form. Complex or double complex entities are equal only if their corresponding real and imaginary parts are equal. When two arithmetic expressions of differing data types are compared, the value of the relational expression is:

`((`*e1*`) - (`*e2*`)) op 0`

0 is of the same type as the expression ((*e1*) - (*e2*)).

If two character expressions are compared, *e1* is considered to be less than *e2* if *e1* precedes the value of *e2* in the collating sequence; *e1*'s value is greater than *e2*'s if *e1*'s value follows *e2*'s in the collating sequence. If the character operands are of unequal length, the length of the shorter is expanded (padded with blanks on the right) to be the length of the longer before the comparison is made.

Note that multiple comparisons of the following form are not permitted:

```
A .GE. B .LE. C
```

Logical operators are used to establish multiple comparisons (see "Logical Expressions" on page 3-17).

Examples:

```
IF (COL .EQ. 72) STOP
IF ('A' .EQ. CHAR) GO TO 10
IF (LM .GT. RM) PRINT *, LM
LVALUE = X .EQ. Y
PRINT *, X .GT. Y
```

Logical Expressions

A *logical expression* expresses a logical computation that produces a single result of type logical with a value of true or false. A logical expression contains a single logical operand, or two or more logical operands connected by logical operators.

A *logical operand* is any of the following:

- Logical constant or the symbolic name of a logical constant

- Logical variable or array element name
- Relational expression
- Function reference yielding a logical value
- Subexpression yielding a logical value

Variables and array element names must be defined (i.e., have a value) before being used in logical assignment statements.

Note that a logical operand represents the value `.TRUE.` or `.FALSE.`

The logical operators that connect logical operands are as follows:

Operator	Logical Meaning	Precedence
`.NOT.`	Negation	First (Highest)
`.AND.`	Conjunction	Second
`.XOR.`	Exclusive “or”	Second
`.OR.`	Disjunction	Third
`.EQV.`	Equivalence	Fourth
`.NEQV.`	Non-equivalence	Fourth

Periods must enclose each operator. The `.NOT.` operator is unary; all others are binary operators requiring two operands.

Arithmetic, character, and relational operators have a higher precedence than logical operators.

When subexpressions containing operators of equal precedence appear in the same expression, the subexpressions are evaluated from left to right. Parentheses are used in subexpressions to alter the order of evaluation.

Two logical operators cannot be used side by side unless the `.NOT.` operator is used as the second operator. The following is valid:

```
X .GE. Y .AND. .NOT. Z
```

NOTE: Z is a logical variable.

The meaning of each operator is explained below:

Logical Expression	Explanation
`.NOT. P`	The expression is true if the value of P is false and is false if the value of P is true.
`P .AND. Q`	The expression is true only if both P and Q are true; otherwise, the expression is false.
`P .XOR. Q`	The expression is true only if P is true and Q is false or if Q is true and P is false. The expression is false if both P and Q are true or if they are both false.
`P .OR. Q`	The expression is true if both P and Q are true or if either P or Q is true. The expression is false only if both P and Q are false.
`P .EQV. Q`	The expression is true only if P and Q have the same logical value, whether true or false.
`P .NEQV. Q`	The expression is true only if P and Q do not have the same logical value, whether true or false.

Table 3-5 gives the truth tables for the logical operators.

Table 3-5. Truth Tables for the Logical Operators

P	Q	.NOT. P	P.AND.Q	P.XOR.Q	P.OR.Q	P.EQV.Q	P.NEQV.Q
T	T	F	T	F	T	T	F
T	F	F	F	T	T	F	T
F	T	T	F	T	T	F	T
F	F	T	F	F	F	T	F

Multiple comparisons using relational and logical operands are permitted, but care should be taken to ensure that the comparison is written correctly:

```
X .GT. 3 .AND. X .LE. 50        Legal
X .GT. 3 .AND. .LE. 50          Illegal
```

A relational operator cannot compare values that are of type logical:

```
IF (FLAG) STOP                   Legal
IF (FLAG .EQ. .TRUE.) STOP       Illegal
```

Part of a logical expression is not evaluated if the result of the expression is already determined. For example,

```
L = E1 .OR. E2
```

If `E1` is true, `E2` is not necessarily evaluated.

The precedence of the logical operators determines the result of a logical expression. The .XOR. and .NEQV. operators have the same effect but different precedence.

Example:

```
X .GT. 3 .XOR. Y .LT. 6 .NEQV. Z .EQ. 1
```

is equivalent to

```
(X .GT. 3 .XOR. Y .LT. 6) .NEQV. Z .EQ. 1
```

but

```
X .GT. 3 .NEQV. Y .LT. 6 .XOR. Z .EQ. 1
```

is equivalent to

```
X .GT. 3 .NEQV. (Y .LT. 6 .XOR. Z .EQ. 1)
```

Logical Operations Using Integer Operands (H)

The Concurrent Fortran compiler permits integer variables, integer array elements, and integer constants to be used as operands in logical expressions. All logical operators (.AND., .OR., .XOR., .EQV., .NEQV., and .NOT.) are valid for logical/integer operations. Any data (decimal, integer, binary, octal, hexadecimal, or Hollerith) stored in an integer storage location can be used in logical/integer operations.

When the logical operators .AND., .OR., .XOR., .EQV., or .NEQV. are used on INTEGER operands, the effect of the operator is as follows:

I .AND. J	The bit-by-bit logical intersection of the data in I and J.
I .OR. J	The bit-by-bit logical union of the data in I and J.
I .XOR. J	The bit-by-bit logical exclusive “or” of the data in I and J.
I .EQV. J	The bit-by-bit logical equivalence of the data in I and J.
I .NEQV. J	The bit-by-bit logical non-equivalence of the data in I and J.
.NOT. J	Results in the logical not of each bit in J.

Examples of logical/integer operations for bit and byte manipulation testing are as follows (assume that variables J and K are INTEGER *4 variables):

```
K = (K .AND.'FFFFFF00'X) .OR.
    (J .SHIFT. -16 .AND. '00FF0000'X)
```

Move the second from the left byte of J into the right most byte of K without disturbing the other bytes of K.

The parentheses in the preceding example are used for clarity and are not necessary since the hierarchical order of the operators guarantee performance of the operations in the proper order.

```
IF (J .AND. '00000001'X) 500, 500, 700
```

Jump to statement number 750 if the `J` is odd (i.e., bit 0 contains a 1); otherwise transfer control to statement number 500.

```
IF ((J .AND. '4'O) .EQ. 0) K = K .OR. '20'O
```

If bit 2 of `J` is off, turn on Bit 4 of `K`.

Logical Assignments

A *logical assignment statement* assigns a logical value of `.TRUE.` or `.FALSE.` to a variable name or array element name. Logical assignment statements are executable statements.

SYNTAX

name = e

DESCRIPTION

name Specifies a variable name or subscripted array element name.

e Specifies a logical or integer expression, a variable, a subscripted array element name, a logical or integer constant, the symbolic name of a logical or integer constant.

A symbolic name appearing to the right of the equals sign need not be of type logical, but the result must be a logical or integer value. A symbolic name appearing to the left of the equals sign may or may not have a value, but it must have been defined in an `IMPLICIT` or explicit type statement to be of type `LOGICAL`.

If *e* is an integer expression, variable, constant or the symbolic name of an integer constant, the value of *name* is the verbatim bit pattern of *e*, right-justified within *name*.

An expression or function reference cannot appear to the left of the equals sign except as part of a subscript.

See also “Multiple Assignment Statements (H)” on page 3-31 and “Array Assignment Statements (H)” on page 3-33.

Examples:

```
L = .TRUE. .AND. SWITCH
CHECK = (.NOT. P) .OR. Q
L = F .OR. (.NOT. C) .AND. R .GE. 23.935E-1
```

Implementation of the LOGICAL Data Type (H)

The Fortran 77 standard does not specify the implementation of the LOGICAL data type except that it is the same size as INTEGER and REAL (ANSI X3.9-1978, Section 2.13). To maximize the portability of non-standard programs, Concurrent provides four mutually exclusive implementations of the LOGICAL data type. Compiler options let you select an implementation; if these options are used together, the compiler generates an error.

Table 3-6 summarizes the features of the four implementations of the LOGICAL data type; a full description of each implementation follows the table. A little vocabulary is necessary before reading the table.

A LOGICAL *expressio*n is an expression involving a LOGICAL operator in which each operand is one of the following:

- A LOGICAL constant (.TRUE. or .FALSE.)
- A LOGICAL variable (scalar or array)
- A LOGICAL function call
- A comparison expression (with .EQ. or .NE.)
- Another LOGICAL expression

A *simple* LOGICAL *expression* is an expression in which each operand is one of the following:

- A LOGICAL constant (.TRUE. or .FALSE.)
- A LOGICAL scalar variable
- Another simple LOGICAL expression

A *truth value* is the value of a LOGICAL expression or simple LOGICAL expression. Depending on the implementation, one or more values may represent a true truth value or a false truth value.

A *value-producing context* is the right-hand side of an assignment statement or an operand of an arithmetic operator.

A *flow-of-control context* is a decision or control situation--for example, IF and WHILE statements.

A *bitwise operation* is an operation that usually generates faster code than a short-circuit operation. This operation does not require testing-and-branching which are time-consuming on modern RISC architectures.

A *short-circuit operation* is an .AND. or .OR. operation that avoids evaluating both operands when evaluation of the first operand determines the final result. Short-circuiting requires testing-and-branching; the overhead of short-circuiting may be prohibitive on modern RISC architectures if the cost of evaluating each operand is small.

Table 3-6. Implementation of the LOGICAL Data Type

Implementation	Constant Values	Truth Values	Operation in Value-Producing Context	Operation in Flow-of-Control Context
Default	`.FALSE.` `0x0`	false: the value 0x0	Short circuit*	Short circuit*
	`.TRUE.` `0x1`	true : the value 0x1		
VAX (**`-V`** or **`-VAX`**)	`.FALSE.` `0x0`	false: any `LOGICAL` expression with a low-order bit of 0	Bitwise	Short circuit*
	`.TRUE.` `0xffffffff`	true : any `LOGICAL` expression with a low-order bit of 1		
Logical-True-Nonzero (**`-Qlogical_true_is_nonzero`**)	`.FALSE.` 0x0	false: the value 0x0	Short circuit	Short circuit
	`.TRUE.` 0x1	true : any nonzero value		
No-Short-Circuit (**`-Qno_short_circuit`**)	`.FALSE.` 0x0	false: the value 0x0	Bitwise	Bitwise
	`.TRUE.` 0x1	true : the value 0x1		

* In these instances, Concurrent Fortran performs a short-circuit operation unless a simple `LOGICAL` expression is involved; in that case, it performs a faster bitwise operation.

Default Implementation (H)

The *default implementation* executes as rapidly as possible. The constant `.FALSE.` is represented by `0x0` and `.TRUE.` is represented by `0x1`. `LOGICAL` expressions with other values are considered ambiguous and should be avoided.

For speed, all simple `LOGICAL` expressions, whether they are in value-producing or flow-of-control contexts, are handled with bitwise operations, not short-circuit operations. However, `LOGICAL` expressions that are not simple are short-circuited; consider this fact when using operations with side effects, such as function calls. For example,

```
F .AND. PRINT_HELLO()
```

If `LOGICAL` variable F is false, the function `PRINT_HELLO()` is not called.

VAX Implementation (H)

The Concurrent Fortran *VAX implementation* emulates the DEC VAX LOGICAL implementation. Select this implementation with either the **-V** or the **-VAX** compiler option. The constant .FALSE. is represented by 0x0 and .TRUE. is represented by 0xffffffff (all bits set). The low-order bit of a LOGICAL expression determines its truth value. A LOGICAL expression with a low-order bit of 0 has a false truth value; a LOGICAL expression with a low-order bit of 1 has a true truth value.

For speed, all simple LOGICAL expressions are handled with bitwise operations, not short-circuit operations. All LOGICAL expressions in value-producing contexts use bitwise operations. In a flow-of-control context, .AND. and .OR. are short circuited unless a simple LOGICAL expression is involved; in that case, Concurrent Fortran performs a faster bitwise operation.

When the Concurrent Fortran driver is invoked with the **-V** or **-VAX** option, it automatically links in an object file called /**lib/vax.o**. The presence of this file directs the library to use only the low-order bit of a LOGICAL expression to determine truth value and to return 0xffffffff for .TRUE.

logical_true_is_nonzero Implementation (H)

The *logical_true_is_nonzero implementation* uses an interpretation of truth value used by some other Fortran compilers, and by the C programming language. Select this implementation with the **-Qlogical_true_is_nonzero** compiler option. As in default mode, the constant .FALSE. is represented by 0x0 and .TRUE. is represented by 0x1. A LOGICAL expression with a zero value has a false truth value; a LOGICAL expression with a nonzero value has a true truth value. In this mode, the composition of true truth values prohibits the determination of truth values with a bitwise operation. Therefore, this implementation always short circuits and never uses bitwise operations.

If many complicated LOGICAL expressions are involved, performance of executing code is slower than that of the default and VAX implementations.

no_short_circuit Implementation (H)

The *no_short_circuit implementation* always uses bitwise operations and never short circuits. Select this implementation with the **-Qno_short_circuit** compiler option. As in default mode, the constant .FALSE. is represented by 0x0 and .TRUE. is represented by 0x1. Avoid LOGICAL expressions with other values because these values are ambiguous.

Use of Arithmetic, Character, and Logical Expressions

Expressions can contain combinations of other expressions.

Example:

```
A+B .GT. D+C .OR. X+Y .GT. Y+Z
```

If a character substring is referenced in a character expression, no part of the substring is undefined.

Any function reference, variable, or array element name used in an expression must have a value at the time the expression is executed. If a function reference defines the value of a symbolic name, the same symbolic name does not appear elsewhere within the same statement.

Examples:

```
A(I,J) = F(I)
Y = G(X) + X
```

These statements are not permitted if the reference to function F defines I or if the reference to function `G` defines `X`.

Evaluation of a subscript, a substring reference, or the arguments of a function reference in an expression do not affect the data type of the expression.

If a function reference appears in an expression where the entire expression need not be evaluated (e.g., in logical expressions such as `A .GT. B .OR. C(I)`, where `A` is greater than `B`), the value of function `C`'s argument(s), if defined by the function, is undefined when the statement completes execution.

Function references nested within other function references are evaluated from innermost to outermost. For example, function `R(I)` is evaluated first in the following expression:

```
X(R(I))
```

Note that `0.5*J` is not the same as `J/2` and that `X*(Y-Z)` is not the same as `X*Y-X*Z`, since the arithmetic result can be different.

In the evaluation of the expression:

```
D + R + I
```

where `D`, `R`, and `I` represent a double precision, real, and integer number, respectively, the data type of the operand that is added to I is unpredictable because the compiler can add `D` and `R`, `R` and `I`, or `D` and `I` first. To avoid this problem, use parentheses to define the entity to be added first.

Summary of Mixed Assignments and Operator Precedence

Table 3-7 lists all Fortran operators in their order of precedence. Table 3-8 indicates whether an expression of one data type can be assigned as the value of a variable or array element of a different data type.

Table 3-7. Fortran Operators - Order of Precedence

Operation	Denotation	Precedence
Shift	.SHIFT.	1 (Highest)
Rotate	.ROTAT.	1
Exponentiation	**	2
Multiplication	*	3
Division	/	3
Addition	+	4
Subtraction	-	4
Unary plus	+	4
Unary minus	-	4
Character concatenation	//	5
Less than	.LT.	6
Less than or equal to	.LE.	6
Greater than	.GT.	6
Greater than or equal to	.GE.	6
Not equal to	.NE.	6
Equal to	.EQ.	6
Logical negation	.NOT.	7
Conjunction	.AND.	8
Exclusive “or”	.XOR.	8
Disjunction	.OR.	9
Logical Equivalence	.EQV.	10
Logical Non-Equivalence	.NEQV.	10

Table 3-8. Validity of Mixed Variable Assignments

Variable or Array Element Being Assigned To	Data Type of Expression Being Assigned	Result
INTEGER *1	Integer [1]	INTEGER *1
	Real [2]	INTEGER *1
	Double Precision [2]	INTEGER *1
	Complex [2]	INTEGER *1
	Double complex [2]	INTEGER *1
	Hollerith	INTEGER *1
	Logical	INTEGER *1
	Character [5]	INTEGER *1
INTEGER *2	Integer [3]	INTEGER *2
	Real [4]	INTEGER *2
	Double Precision [4]	INTEGER *2
	Complex [4]	INTEGER *2
	Double complex [4]	INTEGER *2
	Hollerith	INTEGER *2
	Logical	INTEGER *2
	Character [5]	INTEGER *2
INTEGER	Integer	INTEGER
	Real	INTEGER
	Double Precision	INTEGER
	Complex	INTEGER
	Double complex	INTEGER
	Hollerith	INTEGER
	Logical	INTEGER
	Character [5]	INTEGER
REAL	Integer	REAL
	Real	REAL
	Double Precision	REAL
	Complex	REAL
	Double complex	REAL
	Hollerith	REAL
	Logical	Prohibited
	Character	REAL
DOUBLE PRECISION	Integer	DOUBLE PRECISION
	Real	DOUBLE PRECISION
	Double Precision	DOUBLE PRECISION
	Complex	DOUBLE PRECISION
	Double complex	DOUBLE PRECISION
	Hollerith	DOUBLE PRECISION
	Logical	Prohibited
	Character	DOUBLE PRECISION

Table 3-8. Validity of Mixed Variable Assignments (Cont.)

Variable or Array Element Being Assigned To	Data Type of Expression Being Assigned	Result
COMPLEX	Integer	COMPLEX
	Real	COMPLEX
	Double Precision	COMPLEX
	Complex	COMPLEX
	Double complex	COMPLEX
	Hollerith	Prohibited
	Logical	Prohibited
	Character	Prohibited
DOUBLE COMPLEX	Integer	DOUBLE COMPLEX
	Real	DOUBLE COMPLEX
	Double Precision	DOUBLE COMPLEX
	Complex	DOUBLE COMPLEX
	Double complex	DOUBLE COMPLEX
	Hollerith	Prohibited
	Logical	Prohibited
	Character	Prohibited
LOGICAL *1	Logical	LOGICAL *1
	Hollerith	LOGICAL *1
	Character	LOGICAL *1
	All other data types are prohibited	
LOGICAL *2	Logical	LOGICAL *2
	Hollerith	LOGICAL *2
	Character	LOGICAL *2
	All other data types are prohibited	
LOGICAL	Logical	LOGICAL
	Hollerith	LOGICAL
	Character	LOGICAL
	All other data types are prohibited	
CHARACTER	Hollerith	CHARACTER
	Character	CHARACTER
	All other data types are prohibited	

NOTES:

1. Integer numbers outside the range of INTEGER *1 numbers are reduced modulo 256 to produce an integer number in the range of INTEGER *1 numbers.
2. Real and double precision numbers are converted to integers, and then the least significant (low-order) eight bits of the result are assigned as the value of the INTEGER *1 entity.

3. Integer numbers outside the range of `INTEGER *2` numbers are reduced modulo 65536 to produce an integer number in the range of `INTEGER *2` numbers.
4. Real and double precision numbers are converted to integers, and then the least significant (low-order) sixteen bits of the result are assigned as the value of the `INTEGER *2` entity.
5. Character literals only are allowed, and a warning is issued.

ASSIGN Statement

The ASSIGN statement associates a statement label with a variable. The statement label must be present in the same program unit as the ASSIGN statement, and the variable must implicitly be of type integer or be defined in a declaration statement to be of type integer. The variable can be used later in the same program unit in an assigned GO TO statement or as a format specifier for a FORMAT statement.

SYNTAX

ASSIGN *label* TO *int-var*

DESCRIPTION

label Specifies the statement label of an executable statement or FORMAT statement.

int-var Specifies an integer variable name.

The ASSIGN statement is an executable statement that causes the specified statement label to be assigned as the value of the variable *int-var*.

Execution of an ASSIGN statement is the only way a symbolic name can be defined to have a statement label value; the ASSIGN statement must precede the assigned GO TO statement in the execution sequence but not necessarily in the source program. The variable is not to be used later in the program in an arithmetic context in which its value is altered. For example, a sequence like the following should not be used:

```
ASSIGN 5 TO LINE
LINE = LINE + 1
```

The variable can be redefined as an integer variable for use in an arithmetic context:

```
LINE = 100
```

But, in so doing, LINE is no longer associated with the statement labeled 5 and cannot be used in an assigned GO TO statement or as a format specifier.

Multiple Assignment Statements (H)

The *multiple assignment statement* assigns a value to more than one variable. Multiple assignment statements are executable statements.

SYNTAX

name1 = name2 = ... namei = e

DESCRIPTION

name Specifies a variable name or array element, including character variables, character subscripts and character substrings.

e Specifies an arithmetic or character expression, variable name, array element, character substring, constant, or the symbolic name of a constant. For character values, all elements of the name list must also be of type character.

The same restrictions apply here as for single assignment statements. Any use of a symbolic name in the far right expression must be defined. A name from the list cannot be an expression or function reference, except as an array subscript or in a substring reference.

Each equals sign (=) does not imply equality, but rather that the value to the right of the equals sign is to be assigned to the value of the element to the left of the equals sign. Assignment occurs from right to left. The rightmost assignment is performed first, assigning the rightmost value to the element second from the right, with the second assignment assigning the value of the element second from the right to the element third from the right, and so on. Type conversion and its effects are cascaded through each assignment. Observe the following example, where `A` and `C` are `REAL*4` and `B` is `INTEGER*4`.

```
A = B = C = 4.3
```

This is equivalent to

```
C = 4.3
B = C
A = B
```

and the values printed for `A`, `B`, and `C` are

```
A=4.0000   B=4   C=4.3000
```

For character assignments, values are affected by truncations and blank fills.

Example

```
CHARACTER U*16, T*6, S*10
S = T = U = "my cat has fleas"
```

The values printed for `S`, `T`, and `U` are

```
S = "my catbbbb"   T= "my cat"   U= "my cat has fleas"
```

Note the truncation required to fit the value of `U` into `T` affects the value assigned to `S`, as assignment occurs from right to left.

Array Assignment Statements (H)

The array assignment statement assigns a value to every visible element of an array.

SYNTAX

name = *e*

DESCRIPTION

name Specifies an actual or dummy array name.

e Specifies an arithmetic or character expression, variable name, array element, character substring, constant, or the symbolic name of a constant. The *e* may not be an array name.

The value of the expression *e* is assigned to every visible element of the array specified by name. The same type of restrictions that apply to single assignment statements also apply to array assignments.

Name may be a dummy array name, but must not be of an assumed size. Every dimension must be specified or an error results. Adjustably dimensioned dummy arrays are allowed, and only the portion visible to the subprogram is modified.

4
Specification Statements

4
Specification Statements

General Specification Statements

Specification statements define the data types of symbolic names, declare the storage requirements of variables and arrays, define initial values for variables and array elements, specify the dimensions of arrays, define program entities that are known globally throughout the source program among all program units, and give symbolic names to arithmetic, character, and logical constants. Specification statements are non-executable and have no effect during the execution of the source program.

This chapter presents the declaration statements first, in alphabetical order, followed by the rest of the specification statements also in alphabetical order. Each statement has a detailed description containing a definition, syntax line and explanation of the syntax.

Character Declarations

A variable name, array name, the symbolic name of a constant, an external function name, or a statement function name is declared to be of type character in an explicit type statement.

SYNTAX

```
CHARACTER [*n [,] ] v [*n] [/clist/] [, v [*n] [/clist/] ] ...
```

DESCRIPTION

v (REQUIRED) Specifies a variable name, an array name (with an optional array declarator), the symbolic name of a constant, a function name, or a dummy procedure name. If more than one name is specified, each name is separated by commas.

n* (OPTIONAL) Defines the total number of characters the symbolic name has. If specified with the keyword `CHARACTER`, the length specification applies to all symbolic names in the statement except those that contain their own length. For arrays, the specified length applies to each element of the array. If a length is not specified, `*1` is the default. The length specification is an unsigned, non-zero integer constant, a length expression enclosed in parentheses with a positive integer value, or an asterisk in parentheses. If the length specification is an integer constant or expression, the default maximum allowed value is `1023`. To increase this value, refer to the **`-Nt` option described on the Concurrent Fortran man page.

/*clist*/ (OPTIONAL) If *v* is a variable name, then *clist* specifies a constant or the symbolic name of a constant. If *v* is an array name, then *clist* is a comma-separated list of constants or symbolic names of constants that are assigned on a left-to-right, one-to-one basis as the value of the corresponding element in the array (the same number of constants must appear in *clist* as there are elements in the array). A *clist* may not be specified for the symbolic name of a constant, a function name, or a dummy procedure name. See "DATA Statement" on page 4-12 for more information on how to specify *clist*.

Examples:

```
PARAMETER (LEN=100)
CHARACTER *1 X
CHARACTER *(5+23) C
CHARACTER *(LEN) CHRS
CHARACTER *(*) B
CHARACTER *6 C/'INPUT '/, D/'OUTPUT'/,E
```

A *length expression* must be an integer constant expression.

If a character constant is given a symbolic name in a `PARAMETER` statement, and a length of `(*)` is specified in the `CHARACTER` statement declaring the symbolic name, the length

of the name is assumed to be the length of the corresponding constant expression in the PARAMETER statement.

The length for a character statement function or statement function dummy argument of type character must be an integer constant or a length expression.

Examples:

```
CHARACTER *120 LINE, PAGE(66) *85
CHARACTER X, Y, Z
CHARACTER TEN *10, TWENTY *20, THIRTY *30
```

The compiler sets a maximum length for character strings that defaults to 1023. The default can be set by giving a positive integer argument to the **-Nt** option (e.g., **-Nt3000**).

Logical Declarations

A variable name, array name, the symbolic name of a constant, an external function name, or a statement function name can be declared to be of type LOGICAL in an explicit type statement.

SYNTAX

```
LOGICAL[*n]  v [*n] [/clist/] [, v [*n] [/clist/] ] ...
```

DESCRIPTION

v (REQUIRED) Specifies a variable name, an array name (with an optional array declarator), the symbolic name of a constant, a function name, or a dummy procedure name. If more than one name is specified, each name is separated by commas.

**n* (OPTIONAL) Defines a length in bytes for the symbolic name being declared. If specified with the keyword LOGICAL, the length specification applies to all symbolic names in the statement except those that contain their own length specification. The length specification is an unsigned, non-zero integer constant that is a valid length for data type LOGICAL (i.e., 1, 2 and 4). If a length is not specified, the default is 4.

/*clist*/ (OPTIONAL) If *v* is a variable name, then *clist* specifies a constant or the symbolic name of a constant. If *v* is an array name, then *clist* is a comma-separated list of constants or symbolic names of constants that are assigned on a left-to-right, one-to-one basis as the value of the corresponding element in the array (the same number of constants must appear in *clist* as there are elements in the array). A *clist* may not be specified for the symbolic name of a constant, a function name, or a dummy procedure name. See "COMMON Statement" on page 4-9 for more information on how to specify *clist*.

Logical data types along with their possible length specifications are as follows:

```
LOGICAL*1
LOGICAL*2
LOGICAL*4
LOGICAL
```

Examples:

```
LOGICAL TVALUE, FVALUE
LOGICAL*2 ON, OFF*4, CONTROL
LOGICAL T/.TRUE./, F/.FALSE./
```

For information about controlling the size of converted LOGICAL variables and constants, see "%LOG1, %LOG2, and %LOG4 Logical Size Intrinsics (H)" on page 9-9.

Numeric Declarations

A variable name, array name, the symbolic name of a constant, an external function name, or a statement function name can be declared to be of type integer, real, double precision, complex, or double complex in an explicit type statement. Explicit type statements are in effect for a single program unit, and they override or confirm any implicit definition based on the first letter of the variable name. A symbolic name cannot be explicitly typed more than once in the same program unit.

SYNTAX

```
INTEGER[*n]       v [*n] [/clist/] [, v [*n] [/clist/] ] ...
REAL[*n]          v [*n] [/clist/] [, v [*n] [/clist/] ] ...
DOUBLE PRECISION v [, v] ...
COMPLEX[*n]       v [*n] [/clist/] [, v [*n] [/clist/] ] ...
DOUBLE COMPLEX    v [, v] ...
```

DESCRIPTION

v (REQUIRED) Specifies a variable name, an array name (with an optional array declarator), the symbolic name of a constant, a function name, or a dummy procedure name. If more than one name is specified, each name is separated by commas.

**n* (OPTIONAL) Defines a length in bytes for the symbolic name being declared. If specified with the type keyword, the length specification applies to all symbolic names in the statement except those that contain their own length specification. The length specification is an unsigned, non-zero integer constant that is a valid length for the numeric data type (i.e., 1, 2 and 4 for `INTEGER`, 4 and 8 for `REAL`, 8 and 16 for `COMPLEX`). If a length is not specified, the default length is 4 for `INTEGER` and `REAL` and 8 for `COMPLEX`. `REAL*8` is equivalent to `DOUBLE PRECISION` and `COMPLEX*16` is equivalent to `DOUBLE COMPLEX`.

/*clist*/ (OPTIONAL) If *v* is a variable name, then *clist* specifies a constant or the symbolic name of a constant. If *v* is an array name, then *clist* is a comma-separated list of constants or symbolic names of constants that are assigned on a left-to-right, one-to-one basis as the value of the corresponding element in the array (the same number of constants must appear in *clist* as there are elements in the array). A *clist* may not be specified for the symbolic name of a constant, a function name, or a dummy procedure name. See "DATA Statement" on page 4-12 for more information on how to specify *clist*.

Explicit type statements specify the data type of a symbolic name for all occurrences of the symbolic name within a program unit. The name of a main program, subprogram, or block data subprogram must not appear in an explicit type statement.

An explicit type statement that confirms the type of an intrinsic function is permitted but is not required. The data type of an intrinsic function cannot be changed in an explicit type statement; a warning message is issued if this is attempted.

Numeric data types along with their possible length specifications are as follows:

`INTEGER*1`	One-byte integer
`INTEGER*2`	Two-byte integer
`INTEGER*4`	Four-byte integer
`INTEGER`	Four-byte integer
`REAL`	Four-byte real
`REAL*4`	Four-byte real
`REAL*8`	Eight-byte real
`DOUBLE PRECISION`	Eight-byte real
`COMPLEX`	Eight-byte complex
`COMPLEX*16`	Sixteen-byte complex
`DOUBLE COMPLEX`	Sixteen-byte complex

Examples:

```
INTEGER*2  A, B, C, D(2,6), Z
INTEGER A(5) /1,2,3,4,5/
COMPLEX CN1, CN2
REAL*8 IA, IB*4, IC, ID(25)*4, IE(3,3,3), IO
DOUBLE PRECISION IAD, IBD, ICD, TOTAL
DOUBLE COMPLEX  QUAD
```

For information about controlling the size of converted `INTEGER` variables and constants, see "%INT1, %INT2, and %INT4 Integer Size Intrinsics (H)" on page 9-9.

AUTOMATIC Statement (H)

Variables and array names may be declared of automatic storage, which allows them to be allocated on the stack. This means that the values in the variables and array names are not preserved between invocations of the routine containing the declarations.

SYNTAX

```
AUTOMATIC v [, v ] ...
```

DESCRIPTION

v (REQUIRED) Specifies a variable name or an array name (with an optional array declarator). If more than one name is specified, each name is separated by commas.

Automatic variables cannot appear in `EQUIVALENCE`, `DATA`, `SAVE`, or `STATIC` statements.

CEXTERNAL Statement (H)

A CEXTERNAL statement allows the user to tell the compiler that an external function name is in the C language global name space, and object code references to this name should not be appended with an underscore.

SYNTAX

```
CEXTERNAL  name  [ , name ]  ...
```

DESCRIPTION

name (REQUIRED) Specifies the symbolic name of an external C language function. This name can be called as a subroutine or function, or passed in an argument list. Object code references to *name* are not appended with an underscore. The same symbolic name appears only once in any CEXTERNAL statement in a program unit.

The user should assure that a proper return type is given to the symbolic name if it is called as a function.

CEXTERNAL can be used to specify C library routines or system services to be called directly from Fortran. See "Calling C Functions Directly (H)" on page 8-31 for more information.

COMMON Statement

The values of variables and array names in one program unit are not known in other program units unless the values are passed as arguments in the function reference or subroutine call, or unless the variable(s) or array(s) are placed in a common region of memory. A COMMON statement is used to set up a common block of memory that any program unit can access. Use of common blocks communicates values among program units and conserves storage space. Common blocks provide a means of associating entities in different program units and allows different program units to define and reference the same data without using arguments. Common blocks also permit storage units to be shared.

SYNTAX

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

DESCRIPTION

/name/ (OPTIONAL) Specifies the slash-enclosed symbolic name of a common block. It consists of up to 1023 characters. It cannot be the name of a function, subroutine, or any entry point for a function or subroutine.

list (REQUIRED) Specifies a comma-separated list of variable names or unsubscripted array names for each named or unnamed common block. Zero or more common blocks can be specified in a single source program. Names of dummy arguments in a subprogram dummy argument list must not appear in the list.

Common blocks with the same name that are declared in different program units share the same storage area.

One common block for each named or unnamed common block declaration is created. The storage locations are given the symbolic names specified in the list in the order specified in the list. If two common block names are identical, each later defined *list* is appended to the list of names already in that named common block.

If the common block name is omitted, the common area is called *blank common*. If no common block name is specified after the keyword COMMON, the list is placed in blank common, and the slashes can be omitted entirely. Other common block names specified after the first list of common items must be enclosed with slashes or, if the common block name is omitted, two adjacent slashes must be specified. In the following example, VAR1, ARRAY(2,2), and VAR4 are placed in blank common and VAR2 and VAR3 are placed in the named common area CMBLOCK:

```
COMMON VAR1, ARRAY(2,2) /CMBLOCK/ VAR2, VAR3 // VAR4
```

If an array appears, all elements in the array are placed in the common block. Individual array elements cannot be assigned to common blocks. Arrays that were dimensioned previously in DIMENSION or explicit type statements cannot be redimensioned in COMMON statements, and arrays that are dimensioned in COMMON statements cannot be redimensioned later in the program unit.

Even though entities in a common block do not have to be of the same data type, they should be of the same data type because storage is shared on a one-to-one basis. When using INTEGER *1, INTEGER *2, LOGICAL *1, LOGICAL *2, and character entities, care should be taken to ensure that data of other types are aligned on a word boundary. If they are not, filler is inserted and a warning issued.

Each program unit that uses a common block must declare that block with a COMMON statement. Each common block is a collection of contiguous memory locations, known as a *storage sequence*. Common blocks declared in different subprograms but with the same common block name are *storage associated*, and share the same memory locations. Symbolic names that refer to the units can be different in different program units. The order in which the symbolic names are declared in the COMMON statement determines which memory locations they are associated with in the common block.

The name of the common block is the only name that is communicated between program units. Names in each common list are local to the program units and can differ from one to another, but, as a good programming practice, the same names should be used in each common list.

Within a program unit, the size of the common block is the sum of each of the elements, plus filler required for word alignment. Entities equivalenced to elements of the common block are considered to be members of the block, and may therefore increase the size of the common block.

Globally, the size of a common block is the maximum of the sizes defined by the various program units.

Examples:

```
COMMON ARRAY(-3:5,0:7), CHARS(50), X, Y, Z
COMMON /COM1/ A, B, C /COM2/ L, M, N
COMMON /AREA1/ TOTAL // TAXES, INTEREST
```

Shared Memory Interface (H)

Shared memory allows a program to access memory that is also accessible to 1) another process(es), or 2) a hardware device. This allows multiple processes to share data and communicate efficiently, and it allows a process or processes to communicate directly with external devices (assuming those devices access memory directly). A section of shared memory can be directed to a particular physical address, or not, depending on the user's requirements.

Shared memory may be accessed through ordinary COMMON blocks and DATAPOOLs, with one difference: the COMMON block and DATAPOOL must be declared VOLATILE. For example,

```
SUBROUTINE SUB
COMMON /ACOM/ A, B, C (1000)
VOLATILE ACOM
.
.
```

```
.
END
```

The VOLATILE declaration, by itself, does not direct ACOM to shared memory; it merely informs the compiler that items in ACOM may change value asynchronously to the execution of subroutine SUB. Except for this declaration, the program sees no difference between accessing ACOM and accessing any other COMMON block or DATAPOOL. For more information about the VOLATILE statement, see "VOLATILE Statement (H)" on page 4-40.

The **shmdefine(1)** program is used to specify that a COMMON block or DATAPOOL is to be placed in shared memory. This program provides the means to name as many shared memory regions as needed and specifies which COMMON block(s) or DATAPOOL(s) go in which shared memory region. Each shared memory region contains one or more COMMON blocks or DATAPOOLs. The **shmdefine(1)** program creates an object file that must be linked with the Fortran program. When the program starts executing, the shared memory region(s) are automatically attached to the program, so all references to any variable in a COMMON block or DATAPOOL that was placed in a shared memory region refer to the appropriate location in shared memory.

Use the **shmconfig(1M)** utility if a shared memory region must reside at a particular physical memory address. Otherwise, the shared memory region is loaded by the operating system wherever it can find space.

Note that the contents of a shared memory region are not retained across system reboots.

DATA Statement

A DATA statement gives initial values to variables, entire arrays, and single array elements. DATA statements are non-executable and appear anywhere after the specification statements, but before the END statement that closes the program unit. DATA statements initialize entities at compile time only.

If values are changed during the execution of the program, they remain changed no matter how many times the DATA statements are passed during execution.

SYNTAX

```
DATA nlist /clist/ [ [,] nlist /clist/ ] ...
```

DESCRIPTION

nlist (REQUIRED) Specifies one or more variable names, array names, array element names, or implied-DO lists. Entities in the list are separated by commas.

/*clist*/ (REQUIRED) Specifies a comma-separated list of constants or symbolic names of constants, that are assigned on a left-to-right, one-to-one basis as the value of the corresponding entity in *nlist* (the same number of constants must appear in *clist* as there are entities in the corresponding *nlist*). A constant is any numeric, character, Hollerith, or logical value.

If the *nlist* entity is of type LOGICAL, then the corresponding *clist* constant must be of logical type. If the *nlist* entity is of type CHARACTER *1, the *clist* constant must be of type character, or integer, or an octal, hexadecimal, or binary constant in the range or 0-255. If the *nlist* entity is of type CHARACTER with length greater than one, the *clist* constant must be of type CHARACTER. When the *nlist* entity is of type integer, real, double precision, complex, or double complex, the corresponding *clist* constant must also be of type integer, real, double precision, complex, or double complex.

An *nlist* cannot contain dummy arguments or function names. Names of entities in common blocks may appear only in the list within a block data subprogram.

If an unsubscripted array name is specified in *nlist*, there must be enough constants specified in *clist* to be assigned as values for all the elements of the array. Array elements are assigned values in the natural order in which their subscripts vary.

A repetition factor of the following form can be specified for each constant in *clist*:

*r*c*

where *c* is a numeric, character, Hollerith, or logical constant (or the symbolic name of a constant), and *r* is an unsigned integer constant or symbolic name of a constant indicating the number of times *c* is to be repeated. The repetition factor must be in the range of integer values. For example:

```
CHARACTER*1 A(3)
DATA A /3*'b'/
```

is the same as specifying:

```
CHARACTER*1 A(3)
DATA A /'b', 'b', 'b'/
```

If a subscripted array element is specified, the subscript specification must be an integer constant or integer constant expression.

The definition of a character entity causes all characters within that entity to become defined, and each character constant defines exactly one variable or array element.

Example:

```
LOGICAL L
CHARACTER *6 ALPHA
DATA X, J, L /3.5, 7, .TRUE./, IA /9/, ALPHA /'ABCDEF'/
```

Conversion of Hollerith Data

Hollerith data may be stored as the value of a numeric, character, or logical variable. Each numeric or logical variable consumes its size in Hollerith characters, if possible. If there are too few characters in the Hollerith constant to fill the variable, the characters are left justified in the variable and unused trailing bytes are filled with blanks. Any remaining characters are discarded.

Initialization by Numeric Constants (H)

The compiler permits initialization by a bit-pattern constant. You may initialize a `LOGICAL`, `REAL`, or `INTEGER` variable in a `DATA` statement using a bit-pattern constant. Enter this constant as a letter followed by a string enclosed within quotes. If the letter is a "`b`", the string is binary; only ones and zeroes are permitted. An octal string is indicated by an "`o`" and may contain digits from zero through seven. If the letter is "`z`", then the string is hexadecimal and contains digits from zero through nine and letters from a to f. Bit-pattern constants may also take the additional forms described in Chapter 2.

As an example, to initialize the three elements of an array to ten, you might code:

```
INTEGER A(3)
DATA A /B'1010', O'12',Z'A'/
```

Implied-DO in Data Statements

An implied-DO is an intra-statement DO loop that permits a portion of a statement within the range of the implied-DO to be executed repetitively. Implied-DOs appear in DATA statements and in input and output statements.

Implied-DO lists can be used in a DATA statement to initialize part of an array or to initialize elements of an array in a different order from the manner in which they are stored internally.

SYNTAX

```
(dlist, i = e1, e2 [, e3])
```

DESCRIPTION

dlist Specifies one or more array element names and optional implied-DO lists.

i Specifies the name of an integer variable (the implied-DO variable).

e1,e2,e3 Specify integer expressions that contain implied-DO variables or variables from outer implied-DO lists that have this implied-DO within their ranges.

The range of an implied-DO is the list *dlist*. An iteration count and the values of the implied-DO variable are established from *e1*, *e2*, and *e3* exactly as for a DO loop, except that the iteration count must be positive.

When an implied-DO list appears in a DATA statement, the list items in *dlist* are specified once for each iteration of the implied-DO list, with the appropriate substitution of values for any occurrence of the implied-DO variable *i*. The variable *i* appears only in *dlist* as a subscript or substring or as part of a subscript or substring expression.

The appearance of an implied-DO variable name in a DATA statement does not affect the definition status of a variable of the same name in the same program unit.

NOTE: When initializing an entire array to the same value, use of the unsubscripted array name takes much less compiler time than the use of an implied-DO loop.

Expressions are permitted for subscripted array names in *dlist*, and the expressions contain the implied-DO variables of any implied-DO list that has the subscript expression within its range.

Nested implied-DO lists must be enclosed in parentheses and be wholly contained in the surrounding implied-DO list.

Implied-DO lists function like a DO statement and therefore can be executed zero times.

Examples:

```
DIMENSION A(3,5)
CHARACTER *3 CHRS
DATA ((A(I,J),I=1,3),J=1,5)/15*0.1/
DATA A/15*0.1/
DATA I, J, K /0, 0, 0/, X, Y /0., 0./
DATA CHRS /'ABC'/
```

DATAPOOL Statement (H)

A DATAPOOL statement identifies a memory area called a datapool. It contains Fortran specification statements that can be compiled separately from the object program. The memory locations of each variable or array in the DATAPOOL statement are defined when the program is linked, not at compile time.

Using datapools is helpful in the development of large-scale software systems because the definition of global or shared data values can be centralized into a single memory area. The advantage of using a datapool is that it is generally not necessary to recompile all subprograms if the order or content of a datapool is changed, a procedure that is required if COMMON blocks are used.

The datapool is defined using the Concurrent Fortran compiler. Concurrent Fortran forms a datapool dictionary for each datapool definition that is linked with the program and contains space for the datapool declarations.

SYNTAX

DATAPOOL */name/ list* [[,] */name/ list*] ...

DESCRIPTION

/name/ (REQUIRED) Specifies the slash-enclosed symbolic name of a datapool. The */name/* cannot be the name of a function, subroutine, or entry point, or any other global name in the source program.

list (REQUIRED) Specifies a list of one or more variables or array names for each named datapool. If more than one name is specified in the list, each name is separated by a comma.

Other considerations when creating datapools are:

- A datapool dictionary that contains the specified datapool variables must be created with Concurrent Fortran previous to link time.
- Variables in datapool areas may be initially defined in a datapool definition file using **DATA** statements.
- Any data type, length or array dimension declarations defined for particular variables or arrays in the datapool definition must also be declared in the same manner in the Fortran source program. This is to assure proper type, size, dimension, and reference information for the datapool variables.

Defining a Datapool Area (H)

Datapool definitions indicate which variables are contained within a datapool. A datapool definition file contains the following statements

:

IMPLICIT	explicit type
DATAPOOL	END
DATA	INCLUDE
DIMENSION	comment
EQUIVALENCE	

All data types are allowed.

A "C" in column 1 of a line indicates a comment line. Comments can also be placed on the same line as a statement by specifying an exclamation point "!", followed by the comment.

If desired, IMPLICIT statements can be placed at the beginning of the definition file to control default type selection. The definition statements can be arranged in any order.

Only one END statement per definition file is allowed. If END is not present, one is assumed upon reaching the end of the file. Because it is a requirement that datapool definition information for each variable be duplicated in the Fortran source file, users are encouraged to omit the END statement and incorporate the actual datapool definition file into the source using an INCLUDE statement in order to assure accurate duplication.

Multiple datapools can be defined in the same definition file. There are no restrictions as to the order or placement of the definitions; however, a variable cannot be placed in more than one datapool.

NOTE: Equivalencing is allowed between variables in the same datapool. An attempt to equivalence variables in different datapools results in an error.

Datapool names and variable names and declarations have the same limitations as Fortran identifiers and declarations. The following is a sample datapool definition file:

```
      IMPLICIT UNDEFINED (A-Z)

C
C The following variables are used to keep track of
C Dino's feeding schedule.
C
      DATAPOOL /FLINTSTONE/ FRED,WILMA,PEBBLES
      DATAPOOL /RUBBLE/ BETTY,BARNEY
      REAL FRED
      COMPLEX BETTY
      INTEGER BARNEY,
     +       BAM
      DOUBLE PRECISION WILMA
C
C Wilma and Fred take the same days
C
      EQUIVALENCE (WILMA,FRED)
      INTEGER*4 PEBBLES
```

```
C
C So do Barney and Betty
C
      EQUIVALENCE (BARNEY,BETTY)
C
C Can't forget Bam-Bam!!
C
      DATAPOOL /RUBBLE/ BAM(2)
      !End of definition file. END not required.
```

The preceding example shows declarations similar to standard Fortran declarations, however, a name is required in the DATAPOOL statements. There is no blank DATAPOOL. Explicit initialization at declaration is allowed.

Multiple DATAPOOL lines with the same datapool name put the named variables into the same datapool.

Generating a Datapool Dictionary (H)

A datapool dictionary is created by running the Concurrent Fortran compiler over a datapool definition file. Concurrent Fortran produces a dictionary object file for each datapool definition file. The names of the object file and definition file should be the same except for the suffix, which must be **.o** for the object file and **.dp** for the datapool definition file. Users can intermix datapool definition files and Fortran source files in the command line—Concurrent Fortran automatically links the dictionary object with the executable unless otherwise specified. The dictionary object file remains in the current directory and is not erased. When users wish to use a datapool, they must link the desired dictionary with the program object files. The datapool is placed in the appropriate memory space and all datapool references in the program are resolved. Special **ld(1)** commands are not required unless the datapool is to be used in shared memory.

Referencing a Datapool (H)

Datapools are referenced in the Fortran program via the DATAPOOL statement. Multiple datapools can be referenced provided that the variables in each DATAPOOL statement are uniquely named. Variables contained in a datapool, but not included in a DATAPOOL statement, are not accessible from the program. Variables declared with similar names would be considered local variables. A sample program referring to variables from the dictionary defined above:

```
PROGRAM SAMPLE
DATAPOOL /RUBBLE/ BARNEY,BETTY,BAM
DIMENSION BAM (2)
COMPLEX BETTY
INTEGER BARNEY,BAM
EQUIVALENCE (BARNEY,BETTY)
PRINT *,BETTY, BARNEY, BAM(1), BAM(2)
END
```

Equivalencing a datapool variable to a subprogram variable results in an error. However, equivalencing between datapool variables within a Fortran source file (if it duplicates the equivalencing specified in a datapool definition file) provides the compiler with essential reference information and assures that correct code is generated for datapool variable references. Users are encouraged to include their datapool definition files en masse into source files by using the `INCLUDE` statement. Incorrect equivalence information about datapool variables usually causes incorrect program results.

If a datapool is being placed in shared memory and optimization is enabled, the datapool name must be placed in a `VOLATILE` statement.

Placing a Dictionary in Shared Memory (H)

The file name of the datapool dictionary is given as part of the input to **`shmdefine(1)`** using the statement

```
INCLUDE "filename.o"
```

Refer to the `COMMON` statement in “Shared Memory Interface (H)” on page 4-10 in this manual and also “Interprocess Communication” in the *PowerMAX OS Programming Guide* for detailed information on accessing a shared memory region.

DIMENSION Statement

A DIMENSION statement identifies symbolic names for arrays, defines the number of dimensions in each array, and defines the number of elements in each dimension.

SYNTAX

```
DIMENSION a(d) [, a(d)] ...
```

DESCRIPTION

a — Specifies the symbolic name of an array.

a(*d*) — Specifies an array declarator of the form:

a (*d1*[,*d2*,...,*d7*])

d — Specifies from one to seven dimensions for array *a*. Each *d* is an integer constant or a dimension expression. A dimension expression, if present, must be an integer constant expression. An upper and lower bound range for each dimension of the following form can be specified:

a ([*el* :] *eu*,...)

where *el* and *eu* specify the lower and upper bound, respectively. The upper and lower bound is an integer constant or a dimension expression. The rightmost dimension in an array declaration in a subprogram may have an asterisk (*) as the upper bound, indicating that the dimension is unknown.

If a range is not specified, the dimension *d* is the upper bound of the array, and the lower bound is 1.

The size of array *a* is the product of all elements in each dimension of the array. All memory locations for the elements of the array are contiguous and are of the same size (as determined by the data type of the array).

Arrays can also be declared in COMMON, DATAPOOL, AUTOMATIC, and STATIC statements and in explicit type statements.

A symbolic name defined as an array can be declared only once within a program unit; i.e., the dimensions and upper and lower bounds of an array cannot be redefined in any way in the same program unit in a COMMON, DATAPOOL, AUTOMATIC, STATIC or explicit type statement or in another DIMENSION statement. The array name can be used in a COMMON, DATAPOOL, AUTOMATIC, STATIC, or explicit type statement to declare it as an array in common, or to give the array an explicit data type, but no array declarator can be specified.

When an element of the array is referenced, the same number of subscripts must be used as there are dimensions in the array, unless the array name is used in a context that does not require any subscripts or is used in an EQUIVALENCE statement.

Array declarators cannot contain variable names in a main program; however, they can contain variable names in a subprogram (see Chapter 8).

Examples:

Example	Explanation
`DIMENSION VECTOR(25)`	Correct
`DIMENSION MATRIX(5,6)`	Correct
`DIMENSION Z(-2:2,5,15)`	Correct
`DIMENSION A(-5:-3)`	Correct
`DIMENSION X(8), Y`	Dimensions omitted for Y
`DIMENSION B(0)`	Zero dimensions invalid
`DIMENSION X(-3)`	Upper bound is less than lower bound

EQUIVALENCE Statement

The EQUIVALENCE statement allows you to refer to the same sequence of contiguous storage locations with two or more symbolic names. (This statement does not establish an equivalence in the mathematical sense of the word.)

An EQUIVALENCE statement is used to conserve memory, to set up a "structure" of data, or to add more storage units at the end of a common block but not at the beginning of a common block. EQUIVALENCE allows the same storage locations to be used for different purposes with different symbolic names. An EQUIVALENCE statement applies only to the program unit in which it is defined, unless one of the equivalenced entities is also in a common block.

SYNTAX

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

DESCRIPTION

list (REQUIRED) Specifies a list of variable names, array names, array element names, or character substring names. Each parenthesized *list* is separated by commas, and items within each *list* are separated by commas. All items in a *list* are allocated beginning at the same storage location. Parentheses must enclose each equivalence list.

Example:

```
CHARACTER *3 I, J
CHARACTER *7 K, L, M*2
EQUIVALENCE (I, K), (J, L, M)
```

Associates the variables I, J, K, L, and M to storage units as follows:

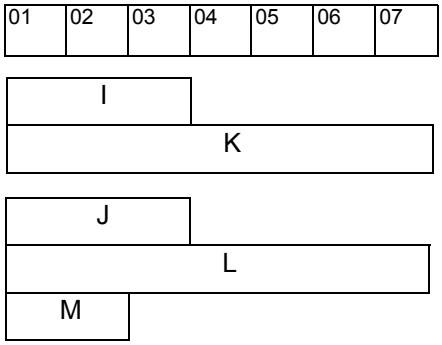

Figure 4-1. Equivalencing Scalars

Dummy argument names must not appear in an equivalence list in a subprogram. In addition, if a variable name is also the name of a function, that name must not appear in an equivalence list.

Numeric, character, and logical entities of any data type can be equivalenced (i.e., symbolic names representing character, numeric, or logical data can appear in an equivalence list), and the length of the entities need not be the same. Where the lengths of the entities are not the same, the entities are partially associated. Also, if entities of different data types are equivalenced, conversion of data types is not performed.

Variables can be equivalenced with arrays and neither takes on the properties of the other (e.g., a variable equivalenced with an array element is not part of any array nor is the array element a variable).

An `EQUIVALENCE` statement must not specify that the same storage location is to occur more than once in a storage sequence.

Example:

```
INTEGER A, B (2)
EQUIVALENCE (B(1), A), (B(2), A)
```

defines `B(1)` and `B(2)` to be at the same storage location.

Also, two storage locations that must be consecutive cannot be made non-consecutive with an `EQUIVALENCE` statement. For example,

```
CHARACTER *1 A(5), B(2), C
EQUIVALENCE (A(1), B(1)), (A(5), B(2))
```

is an illegal use of `EQUIVALENCE`. It causes the first element of `A` to be equivalenced to the first element of `B` and the second element of `B` to be equivalenced to the fifth element of `A`. The preceding equivalence is illegal because it indicates that an array is to be stored non-linearly in memory.

Associating one entity with another in an `EQUIVALENCE` statement can also cause the association of other entities. For example, if an array element name is equivalenced with another symbolic name in a storage sequence, other elements of the array are also associated with storage units in the storage sequence.

For example,

```
INTEGER *2 A(3), B(5), C, D
EQUIVALENCE (A(2), B(5)), (C, D, B(4))
```

associates the variables `C` and `D` and the element of arrays `A` and `B` as follows:

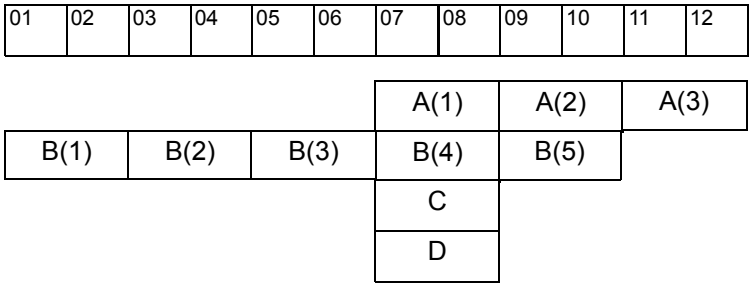

Figure 4-2. Equivalencing Arrays

Note that adjacent elements in an array are also associated even though the elements were not specifically equivalenced in an equivalence list (e.g., `A(1)` is equivalenced to `B(4)`). Note also that two arrays of equal size could occupy the same storage space if an `EQUIVALENCE` statement caused the first two elements of each array to be equivalenced.

Partial association of entities occurs when some but not all of the storage units of the entities share the same storage. For example,

```
CHARACTER *4 X, Y, Z*3
EQUIVALENCE (X(2:4), Y, Z)
```

associates `X`, `Y`, and `Z` as follows:

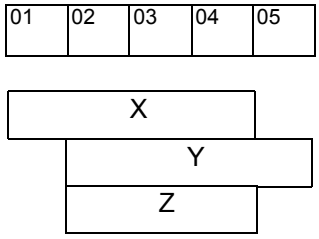

Figure 4-3. Equivalencing with Partial Association

Symbolic names defined as being in common blocks are used in `EQUIVALENCE` statements, but not more than one of the elements in an equivalence list can be in a common block. Two components in the same or different common blocks cannot be equivalent.

An `EQUIVALENCE` statement can be used to extend a common block beyond its original boundaries, but only beyond the last element in the common block. The ordering of common blocks cannot be forced with an `EQUIVALENCE` statement. For example, the following `EQUIVALENCE` is not allowed:

```
DIMENSION ARRAY(2)
COMMON /COM1/ C1
COMMON /COM2/ C2
EQUIVALENCE (ARRAY(1), C1), (ARRAY(2), C2)
```

A single (one-dimensional) subscript may be used in EQUIVALENCE statements with array elements that were previously defined as arrays with two or more dimensions. A single subscript represents the linear element number in the bounds of the array. However, the lower bound of all dimensions in a multi-dimensional array must be 1 or a single subscript cannot be used in an EQUIVALENCE statement. If the lower bounds of any dimension in the multi-dimensional array is not 1, the correct number of dimensions as defined for the array must be specified in the EQUIVALENCE statement.

Example:

```
CHARACTER *1 X(3,2,2), Y(2,3)
EQUIVALENCE (X(6),Y(2))
```

the arrays are aligned as follows:

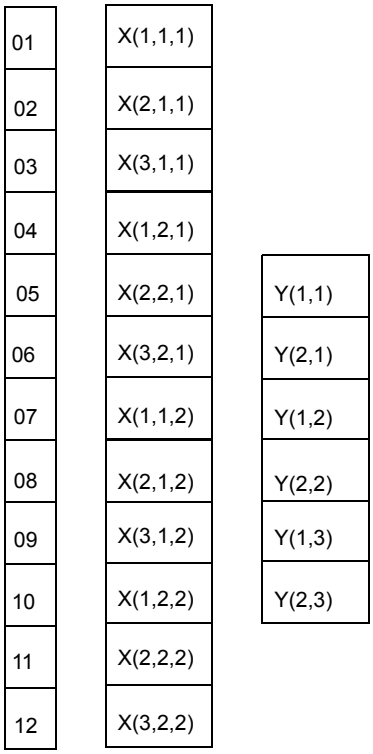

Figure 4-4. Equivalencing Arrays with One Subscript

EXTERNAL Statement

An EXTERNAL statement allows a user-defined function or subroutine name, a dummy procedure name, or a block data subprogram name to be used as an actual argument in a subroutine call or function reference. If procedure names that are passed as actual arguments to subprograms are not declared in an EXTERNAL statement, the procedure names are interpreted as variables. An EXTERNAL statement is permitted wherever a specification statement is allowed.

SYNTAX

```
EXTERNAL name [, name] ...
```

DESCRIPTION

name (REQUIRED) Specifies the symbolic name of an external procedure, dummy procedure, or block data subprogram. The same symbolic name appears only once in any EXTERNAL statement in a program unit.

If a complete function reference (including function name and arguments) is used in a function reference or subroutine call, then the name of the complete function reference need not appear in an EXTERNAL statement.

Example:

```
CALL SUB (X,Y,SQRT(Z))
```

SQRT(Z) represents an actual value and SQRT need not be specified in an EXTERNAL statement.

If the name of an intrinsic function appears in an EXTERNAL statement, that name becomes the name of an external procedure and the intrinsic function is no longer available for reference in the program unit. Using intrinsic function names in EXTERNAL statements permits a programmer to override intrinsic function names, so that the same name can be used for a user-defined function. Note, however, that the generic properties of any name appearing in an EXTERNAL statement are not lost.

A statement function name must not appear in an EXTERNAL statement. Example:

```
PROGRAM MAIN
EXTERNAL DSQUARE
DOUBLE PRECISION D
. . .
X = DSQUARE (Y)    ! statement function declaration
. . .
END

FUNCTION DSQUARE (ROOT)
. . .
END
```

Statement X = DSQUARE (Y) references a user-defined function called DSQUARE.

IMPLICIT Statement

The IMPLICIT statement overrides or confirms the implied data type for symbolic names. A program unit can have more than one IMPLICIT statement, and all IMPLICIT statements must precede all other specification statements. The names of intrinsic functions are not affected by the IMPLICIT statement.

SYNTAX

```
IMPLICIT type [*n](a [, a]...) [,type [*n](a [, a] ...)] ...

IMPLICIT NONE
```

DESCRIPTION

a Specifies a single letter or a list of single letters in any order. A range of letters is denoted as follows:

a1-a2

a1 and *a2* must be in alphabetical order.

Parentheses must enclose the letters defined for a particular type. Each single letter or range of letters is separated by commas.

The same letter cannot appear explicitly or in a range of letters in another IMPLICIT statement within the same program unit.

type Specifies one of the following:

BYTE	COMPLEX
INTEGER *1	COMPLEX *8
INTEGER *2	COMPLEX *16
INTEGER *4	DOUBLE COMPLEX
INTEGER	LOGICAL *1
REAL	LOGICAL *2
REAL *4	LOGICAL *4
REAL *8	LOGICAL
DOUBLE PRECISION	CHARACTER *n
	UNDEFINED

Defaults for all data types are as defined in Chapter 2. If UNDEFINED is selected as the type, then there is no implicit type for variables starting with the letters specified. If IMPLICIT NONE is used, then there is no implicit type of any variable. The compiler issues a warning for each variable without an implicit type that is used but which does not appear in a type statement.

An IMPLICIT statement applies only to the program unit that contains it.

The appearance of a symbolic name in an explicit type statement overrides any typing and length defined by an IMPLICIT statement. An explicit data type specified in a FUNCTION statement overrides an IMPLICIT statement for the name of the function subprogram. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

Examples:

```
IMPLICIT DOUBLE PRECISION (A-E)
IMPLICIT DOUBLE PRECISION (A, B, C, D, E)
IMPLICIT INTEGER*2  (P-R, T, X-Z)
IMPLICIT REAL (K, J, I)
IMPLICIT CHARACTER *25 (F-H), INTEGER (I), COMPLEX (C, P)
```

INTRINSIC Statement

An INTRINSIC statement identifies a symbolic name as the name of an intrinsic function, and permits the symbolic name of an intrinsic to be used as an actual argument in a function reference or subroutine call.

SYNTAX

```
INTRINSIC fname [, fname] ...
```

DESCRIPTION

fname (REQUIRED) Specifies the generic or specific name of an intrinsic function. A symbolic name appears only once in any INTRINSIC statement within a program unit, and a symbolic name cannot appear in both an EXTERNAL and an INTRINSIC statement in the same program unit.

If the symbolic name of an intrinsic function is used as an actual argument in a function reference or subroutine call in a program unit, the symbolic name must be identified in an INTRINSIC statement in the program unit.

The name of a statement function cannot be used in an INTRINSIC statement.

Use of the generic name of a function in an INTRINSIC statement does not cause the name to lose its generic property.

Example:

```
PROGRAM TEST
INTRINSIC SQRT
REAL VALUE, RESULT
. . .
CALL RTINE (SQRT, VALUE, RESULT)
. . .
END


SUBROUTINE RTINE (FN, V, R)
REAL V, R, FN
R = FN(V)
. . .
RETURN
END
```

NAMELIST Statement (H)

The NAMELIST statement defines a list of variables or array names and associates the list with a unique group name. The group name is used in namelist-directed I/O statements.

SYNTAX

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

DESCRIPTION

/name/ (REQUIRED) Specifies the slash-enclosed symbolic name of the namelist group. It cannot be the name of a function, subroutine, or any other symbolic entity in the program.

namelist Specifies a comma-separated list of variable names or unsubscripted array names. Each name in this list is associated with the preceding group name. A variable can appear in more than one namelist. Each *namelist* is appended to the end of any existing *namelist* of the same group name in the same program unit.

The namelist associated with a group name is used by a namelist-directed I/O statement in place of an I/O list. The unique group name identifies a list whose entities can be read or written. Only the entities in the specified group can be read or written during a namelist I/O operation. It is not necessary for the input records of a namelist-directed read operation to define every entity in the associated namelist.

The namelist entities can be of any data type and can be explicitly or implicitly typed. Array elements and character substrings are not permitted in the namelist. Array elements can be specified in a namelist-directed I/O operation.

The order of the entities in the NAMELIST statement determines the order in which the values are written in a namelist-directed write operation. Input values can be in any order.

A variable name can appear in zero or more namelists. Dummy arguments cannot appear in a namelist. Variable and assumed-size arrays are not allowed. Refer to Chapter 6 for more information on namelist-directed I/O operations.

Namelist group names and namelist entity variable names may contain embedded underscores (_); however, embedded dollar signs ($) are not allowed. The dollar sign is a special delimiter in namelist data files. The Concurrent Fortran compiler issues a warning for group names and variable names that violate this restriction within NAMELIST statements. Unpredictable results, including I/O errors and invalid data, may result if this guideline is not followed.

Example:

```
CHARACTER *10 C10
INTEGER   IARRAY(5), INT2
REAL      REALVAL, ARRAY(3), REAL2, REAL3, REAL4
NAMELIST /BLK1/ C10,IARRAY,INT2,REALVAL,REAL2,REAL3
NAMELIST /BLK2/ INT2, C10   /BLK1/ ARRAY, REAL4
```

This namelist statement defines two groups, `BLK1` and `BLK2`. `BLK1` contains the entities `C10`, `IARRAY`, `INT2`, `REALVAL`, `REAL2`, and `REAL4`. `BLK2` contains `INT2` and `C10`.

PARAMETER Statement

A PARAMETER statement gives a symbolic name to a numeric, character, Hollerith, or logical constant. Once defined in a PARAMETER statement, the name of a constant can be used wherever the constant is used except in a format specification. A PARAMETER statement may be used to generalize array and character length declarations so that an array or character entity can be expanded or contracted simply by changing the value of the symbolic constant. PARAMETER statements are evaluated at compilation time and are non-executable statements.

SYNTAX

```
PARAMETER (p1=e [, p2=e] ...)
```

DESCRIPTION

p1, p2... Specify symbolic names.

e Specifies a constant expression. Parentheses must enclose the list of assignments. The data type of the symbolic name is as defined in an explicit type statement or is determined implicitly by the first character of the name. If *p1* is numeric, then *e* must be an arithmetic constant expression or Hollerith value. If *p1* is of type CHARACTER or LOGICAL, then *e* must be a character constant expression or a logical constant expression, respectively. If the data type of *e* differs from that of *p1*, then the data type of the value of *e* is converted to the data type of *p1* in accordance with the rules for assignment statements.

A symbolic name of a constant that appears in expression *e* must have been previously defined in the same or a different PARAMETER statement in the same program unit. The symbolic name of a constant applies only to the program unit in which it is defined. The symbolic name of a constant must not become defined more than once in a program unit.

The default length for the symbolic name of a constant is the same as the default length for the data type. If a different length (or data type) is desired, it must be defined in an explicit type statement or in an IMPLICIT statement before being used in a PARAMETER statement.

The length or data type of the symbolic name of a constant cannot be changed by subsequent statements.

PARAMETER statements can appear anywhere in a program unit, but they must define the symbolic name of a constant before the constant is used.

Example:

```
INTEGER A, B, X
PARAMETER (A=100, B=0)
DIMENSION X(A)
DATA X /A*B/
. . .
END
```

`A`, the symbolic name for integer constant `100`, is used to declare the dimension of vector `X` and is used as a repetition factor in the `DATA` statement. `B`, the symbolic name for integer constant `0`, is used as the initial value to be assigned to all elements of array `X`.

POINTER Statement (H)

The POINTER statement associates a pointer variable and a block of based variables whose locations are determined as offsets from an address held in the pointer variable. A pointer block is useful for implementing dynamic and other data structures otherwise difficult in Fortran.

SYNTAX

```
POINTER  /name/  list  [  [,]  /name/  list  ] ...
```

DESCRIPTION

/name/ (REQUIRED) Specifies the pointer variable which holds the base address of the pointer block. The pointer variable must be declared separately, and requires INTEGER*4 type to hold a full system address. It must be scalar, and may be a local variable, a dummy argument, a member of a common block or datapool, or a member of another pointer block.

list (REQUIRED) Specifies the comma-separated list of one or more variables or array names associated with the pointer variable name. No storage is allocated for these based variables. Based variables may appear in the VOLATILE statement and may be equivalenced to local variables. Like common block variables, all based variables in a pointer block form a storage sequence.

Storage for pointer block members is supplied by the user and the address is assigned to the pointer variable. Once the pointer variable contains a valid address, the associated based variables may be used. Use of a based variable when its associated pointer variable does not contain a valid address usually results in a run-time error and a core dump.

A chunk of storage may be obtained dynamically via the **malloc(3F)** package, or statically by declaring a dummy array and using the %LOC() intrinsic to determine its address.

The address of a based variable is the current value of its associated pointer variable, plus the offset as determined from the member list. Offsets of based variables are the same as offsets computed for an identical list of common block variables, and like common block variables, it is the offset rather than the name that is used for accessing the storage area. A different name may be used for a based variable in a separate subprogram as long as type and size information remains the same.

The total size of a pointer block may be larger than the sum of the sizes of all member variables, due to data-type alignment restrictions. The **sizeofblock(3F)** intrinsic takes as its argument a pointer variable and returns the total size in bytes of the associated pointer block.

Example 1:

```
PROGRAM FUDGE

INTEGER*4 PTR
POINTER /PTR/ A, B(2)
REAL*4 A, B

EXTERNAL MALLOC
INTEGER*4 MALLOC

PTR = MALLOC(SIZEOFBLOCK(PTR))

B(2) = 1.0
B(1) = 2.0
A = B(1) + B(2)

PRINT *, A, B

END
```

Example 2:

```
...
INTEGER*4 PTR
POINTER /PTR/ A, B(2)
REAL*4 A, B

! Static storage allocation.

INTEGER*4 DUMMY(3)

PTR = %LOC(DUMMY) ! %LOC(DUMMY(1)) is also valid.
...
```

SAVE Statement

A SAVE statement in a program unit identifies a list of local entities that do not become undefined when a RETURN or END statement is executed. If a SAVE statement identifies a local entity that is not in a common block and that is defined when a RETURN or END statement is executed, that entity has the same value at the next reference of the subprogram.

The use of a common block name in a SAVE statement is effectively a null operation. However, a saved common block can become undefined or redefined in another program unit.

SYNTAX

```
SAVE [name [, name] ] ...
```

DESCRIPTION

name (OPTIONAL) Specifies a common block name enclosed with slashes, a variable name, or an array name. A symbolic name cannot be specified more than once in a SAVE statement in a program unit. Names of particular entities in a common block, dummy argument names, and procedure names cannot appear in a SAVE statement.

The values of saved common block entities are made available to the next program unit which specifies that common block name in a COMMON statement. If the common block is specified in a main program unit, the current values of saved common block entities are made available to each subprogram that uses the labeled common block, and a SAVE statement in the subprogram has no effect.

The definition status of each entity in the named common block storage sequence depends on the association that has been established for the common block storage sequence.

Example:

```
INTEGER I, J, K(3)
COMMON /COM/ J, K

SAVE COM, I        ! I, J, K are saved
```

STATIC Statement (H)

Variables and array names may be declared of static storage. The values in the variables and array names are preserved between invocations of the routine containing the declarations, as with the SAVE statement.

SYNTAX

```
STATIC v [, v ] ...
```

DESCRIPTION

v (REQUIRED) Specifies a variable name or an array name (with an optional array declarator). If more than one name is specified, each name is separated by commas.

Example:

```
INTEGER I, J(3,4,4), K

STATIC I, J, K     ! I, J, K are static
```

Statement Function Definitions

A statement function definition defines a statement function that is local to the program unit in which it is defined. Statement function definitions may or may not have a dummy argument list, and once the function is evaluated, it must return a value at the point of reference. Statement functions resemble assignment statements and are referenced like intrinsic functions.

SYNTAX

name ([*d* [, *d*] ...]) = *e*

DESCRIPTION

name (REQUIRED) Specifies the symbolic name of the statement function.

d (OPTIONAL) Specifies a dummy argument.

e Specifies an arithmetic, character, relational, or logical expression. The name of the statement function has a data type, and if the result of *e* is not the same data type, the result is converted to the data type of the function name. The conversion rules applied are the same as those for assignment statements.

The data type of the *name* can be declared explicitly in a type statement or can be defined implicitly by the first letter of the name.

Statement function definitions follow other specification statements and precede any executable statements in a program unit.

Statement functions can be referenced only within the program unit in which they are defined. A statement function name cannot be used as a procedure name in an actual argument list.

Each dummy argument in a statement function is a variable name, and all arguments in the list need not have the same data type. When a statement function is referenced, actual arguments are associated with dummy arguments on a one-to-one basis. The same dummy argument can be used as a dummy argument in more than one statement function definition in the same program unit or in other program units, and can also be used as a variable in the same program unit. Constants, subscripted array names, and unsubscripted array names are not valid as dummy arguments in a statement function definition.

The expression *e* can contain constants, symbolic names of constants, dummy argument names, subscripted array names, intrinsic and external function references, dummy procedure references, references to other statement functions previously defined in the same program unit, or an expression in parentheses.

A statement function definition can be continued on more than one line.

A statement function cannot be referenced in another statement function unless it has been previously defined.

If a statement function is defined in a function subprogram, the name of the function subprogram or the name of an entry point in the function subprogram cannot be referenced in expression *e*.

Example:

```
PROGRAM STFU
SQUARE (X) = X ** 2
FUNC (A, B, C) = A * B + MIN(A,B,C)
. . .
END
```

VOLATILE Statement (H)

The VOLATILE type qualifier denotes local variables or common block, datapool or pointer block members which may be modified asynchronously to the execution of the modules in which they are declared. A shared-memory program, for example, could affect a variable in this manner when one process modifies a shared-memory variable that is also used by another process. The VOLATILE keyword limits the kinds of optimizations that may take place on expressions involving the volatile variable.

If one variable of an equivalence class appears in a VOLATILE statement, the entire equivalence class is considered volatile. Individual member variables of common blocks, datapools, or pointer blocks may be volatile without declaring the entire block volatile, though there is also VOLATILE syntax for declaring an entire block. A pointer block and its associated pointer variable must be declared volatile separately.

SYNTAX

VOLATILE *volatile-entity* [, *volatile-entity* ...]

DESCRIPTION

volatile-entity (REQUIRED) One of the following *name* or */block-name/* program elements.

name

Symbolic Name	Volatile Impact
Local variable	The variable is volatile
Member variable of a common block	The variable is volatile
Member variable of a datapool	The variable is volatile
Member variable of a pointer block	The variable is volatile
Pointer variable for a pointer block	The pointer variable is volatile
Common block	Every member variable is volatile
Datapool	Every member variable is volatile

/block-name/

Symbolic Name	Volatile Impact
Common block	Every member variable is volatile
Datapool	Every member variable is volatile
Pointer block	Every member variable is volatile

Example:

```
COMMON   /COM/ R, I
POINTER  /PTR/ S, T
DATAPOOL /DP/ A, B
INTEGER  I, J, PTR
REAL     R, S, T(10,10)
COMPLEX  A, B

VOLATILE /COM/       ! R and I volatile
VOLATILE PTR, /PTR/  ! PTR, S and T volatile
VOLATILE J, A        ! J and A volatile, B is not
!
VOLATILE /J/   ! ERROR: J is not a valid block name
```

5
Control Statements

5
Control Statements

General Description

Control statements affect the order of statement execution in the source program. Statements are executed sequentially, unless control statements indicate a transfer of control or establish an iterative procedure, in which the same sequence of statements are executed zero or more times, depending on the result of a condition. All control statements except CALL and RETURN are discussed in this chapter. The CALL and RETURN statements are discussed in Chapter 8.

A source program is executed sequentially, one statement after another, unless a transfer of control indicates otherwise. GO TO statements permit branching, which is the transfer of control from one statement to an executable statement elsewhere in the same program unit.

There are three types of GO TO statements:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO

The IF statement permits conditional execution of one or more statements and is one of three types:

- Arithmetic IF -- conditional branching
- Logical IF -- conditional statement execution
- Block IF -- conditional execution of a block of statements

The SELECT CASE construct permits selective execution of one of several blocks of statements.

This chapter presents information on the execution of DO loops, IF blocks, and SELECT CASE constructs followed by detailed alphabetical descriptions of the control statements.

Execution of a DO Loop

An active DO loop executes as follows:

1. If the DO loop is being executed for the first time, the iteration count is established based on the initial value and terminal value of the DO variable, and the number by which the DO variable is incremented with each iteration of the loop. The iteration count is then tested, and the first statement after the DO statement is executed if the iteration count is not zero. If the iteration count is zero or negative, the loop is not executed, and execution continues with the first statement after the terminal statement of the DO loop. However, if other DO loops sharing the same terminal statement are active, execution continues with the DO statement for the next outermost DO loop.
2. For subsequent executions of the loop, the value of the DO variable is incremented by the value of the increment parameter, and the iteration count is decreased by 1. If the iteration count is positive and is not zero, execution continues with the statement following the DO statement; otherwise, the loop becomes inactive.

A loop becomes inactive when:

- Its iteration count is tested and determined to be zero or negative.
- A RETURN or STOP statement is executed within the loop's range.
- A statement within the loop transfers control to a statement outside the loop.
- Execution is abnormally terminated.

Transfer into the range of a DO loop from outside the loop is not permitted. Control is transferred only to the DO statement of a DO loop. Control is transferred only from an inner DO loop to any executable statement in an outer DO loop's range. Control may not be transferred from a statement in an outer loop to a statement in an inner loop. If two or more loops share the same terminal statement, control is transferred from an inner loop to that statement, not from an outer loop to that statement.

The terminal statement of a DO loop may not be any of the following statements:

unconditional GO TO	EXIT DO
assigned GO TO	LOOP
arithmetic IF	EXIT LOOP
block IF	END LOOP
ELSE IF	FOR
ELSE	EXIT FOR
END IF	END FOR
RETURN	WHILE
STOP	EXIT WHILE
END	END WHILE
DO	SELECT CASE
END SELECT	CASE

If the terminal statement is a logical IF statement, the object of the logical IF may contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, FOR, LOOP, WHILE, SELECT CASE or another logical IF.

A function reference or subroutine call within the range of the loop does not cause the loop to become inactive unless an alternate return specifier in a subroutine returns control to a statement outside the range of the loop.

A DO loop can be executed zero times.

Example:

```
        M = 0
        DO 50 J = 3, 1
50      M = M + 1
```

The loop is not executed, leaving M=0 and J=3.

When a DO loop becomes inactive, the DO variable retains its last defined value.

Examples:

```
        DO 15 I = 0, 16, 2
        . . .
15      CONTINUE


        DO 111 R = 1.0, 3.5, .1
        . . .
111     CONTINUE


        DO 40 X = INT(HI), INT(LO), -3
        . . .
40      CONTINUE
```

Nested DO Loops

DO loops may be nested to any level; that is, a DO loop can be included in another DO loop, which can be included in another DO loop, etc. A DO loop within the range of another DO loop must be totally within the range of the outer DO loop. Note that more than one DO loop may have the same terminal statement.

A DO loop in an IF block must be wholly contained within that block. An IF, SELECT CASE, WHILE, FOR, LOOP, DO WHILE, or DO-UNTIL block within the range of a DO loop must be totally within that DO loop. For example:

Correct nesting of three DO loops

```
        DO 100 I = 1, 15
        . . .
        DO 30 J = 1, 5
        DO 30 K = 1, 5
        . . .
30      CONTINUE
        . . .
100     CONTINUE
```

Incorrect nesting of three DO loops

```
        DO 100 I = 1, 15
        . . .
        DO 30 J = 1, 5
        DO 35 K = 1, 5
        . . .
100     CONTINUE
        . . .
30      CONTINUE
        . . .
35      CONTINUE
```

Execution of an IF Block

If the expression in the initial IF statement of an IF block is true, the range of statements between the IF statement and the first ELSE IF or ELSE statement, that has the same IF control level as the block IF statement, is executed, and then control passes to the statement after the END IF. If the expression in the initial IF statement is false, processing continues with the next ELSE IF, ELSE, or END IF statement. A THEN clause may be empty.

If the expression in an ELSE IF statement is true, the range of statements between the ELSE IF and the next ELSE IF or ELSE statement, that has the same IF control level as the ELSE IF block, is executed, and then control passes to the statement after the END IF. If the expression in an ELSE IF statement is false, processing continues with the next ELSE IF, ELSE, or END IF statement. An ELSE IF clause may be empty.

If an ELSE statement is present and all previous tests are false, the range of statements between the ELSE and the END IF statement, that have the same IF control level as the ELSE statement, is executed, and then control passes to the statement after the END IF. An ELSE clause may be empty.

Nested IF Blocks

Block IF statements may be nested in other block IF statements. If so, inner block IF's must be wholly contained within the block IF, ELSE IF, or ELSE section in which the inner block IF statement appears.

The IF control level of a statement *s* is:

n1 – n2

where *n1* is the number of initial block IF statements from the beginning of the program unit up to and including a particular statement *s*, and *n2* is the number of END IF statements in the program unit up to but not including statement *s*. The IF control level of every statement *s* must be 0 or positive, and the IF control level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF control level of the END statement of each program unit must be zero, i.e., all active IF blocks must be inactive when the program unit completes execution.

For each block IF, there must be a corresponding END IF at the same IF control level in the same program unit.

The statement label, if any, of an ELSE IF or ELSE statement must not be referenced by any other statement.

Transfer of control into an IF block, ELSE IF block, or ELSE block from outside the IF, ELSE IF, or ELSE block is prohibited.

An ELSE IF statement at the same IF control level as an ELSE statement may not appear after the ELSE statement.

Statements in outermost IF, ELSE IF, or ELSE blocks have a higher IF control level than statements in innermost blocks. A transfer of control from within an IF block, ELSE IF block, or ELSE block is permitted as long as the transfer is to a statement that has a lower IF control level than the IF, ELSE IF, or ELSE statement. A statement within a single clause may transfer control to another statement within the same clause but not to a statement in another clause in the same IF block, unless the transfer is to a statement that is the initial statement of a clause. A transfer of control out of the clause to a statement in another clause at the same IF level or at a higher IF level is prohibited.

Example:

```
                                                    IF Control
                                                    Level for
                                                    Each
                                                    Statement
CHARACTER *3 ICOM                                       0
...
IF (ICOM .EQ. 'ADD') THEN                               1
     OP1 = OP1 + OP2                                    1
     PRINT*, OP1                                        1
ELSE IF (ICOM .EQ. 'SUB') THEN                          1
     OP1 = OP1 - OP2                                    1
     PRINT*, OP1                                        1
ELSE IF (ICOM .EQ. 'MUL') THEN                          1
     OP1 = OP1 * OP2                                    1
     IF(OP1 .GT.100000.) THEN                           2
          PRINT*, "GREATER THAN 100000."                2
     ELSE                                               2
          PRINT*, OP1                                   2
     END IF                                             2
ELSE                                                    1
     PRINT*, "ERROR"                                    1
END IF                                                  1
END                                                     0
```

An `IF` block in a block or in a `DO` loop must be wholly contained within that block or loop.

Execution of a SELECT CASE Construct (H)

The SELECT CASE construct allows selective execution of one of several blocks of statements. The user provides a selection expression in the SELECT CASE statement, which is evaluated and compared sequentially to the constant expression provided in each of subsequent CASE statements. The block of statements following the first CASE that matches the selection expression is evaluated. If no CASE expression matches the selection expression, and a CASE DEFAULT or ELSE statement is provided, the block of statements following the CASE DEFAULT or ELSE statement is executed. Control is then transferred to the first executable statement following the END SELECT statement.

The selection expression may be of any data type, including CHARACTER. It must evaluate to a scalar value. The constant expression in each CASE statement may express any number of values or ranges of values. The exact format of the constant expression is described in "CASE (H)" on page 5-28. The value of the selection expression is compared with each CASE expression in the order the CASE expressions appear in the source.

The block of statements executed on matching a CASE expression may be zero or more statements long and includes all statements between the CASE statement and a following CASE, CASE DEFAULT, ELSE, or END SELECT statement. IF blocks, DO and other loop constructs, and other SELECT CASE constructs may be present but must be wholly contained within the block of statements.

While not required, there may be at most one CASE DEFAULT or ELSE statement per SELECT CASE construct. If ELSE is used, it must appear at the same nesting level as other CASE statements in the same construct. No constant expression is associated with CASE DEFAULT or ELSE. The block of statements following the CASE DEFAULT or ELSE is executed if the selection expression matches no other CASE expression.

There are no restrictions to the sequence of CASE, CASE DEFAULT or ELSE statements within a SELECT CASE construct.

Transfer of control into a SELECT CASE construct is prohibited. Transfer of control out of a SELECT CASE construct is allowed at any time.

CONTINUE Statement

The CONTINUE statement has no effect when executed. It is used as a point to which control can be transferred from elsewhere in the program unit or as the terminal statement of a DO loop.

SYNTAX

[*label*] CONTINUE

DESCRIPTION

label Specifies an optional statement number. If the CONTINUE statement is the terminal statement of a DO loop or is a point to which control is transferred from elsewhere in the program unit, the label must be present.

A CONTINUE statement may be used to terminate a DO loop that would otherwise end in a statement that may not be the terminal statement of a DO loop.

DO Statements

Simple DO

The DO statement defines a loop called a DO loop which is a sequence of executable statements to be executed zero or more times based on control information defined in the DO statement.

SYNTAX

```
      DO label [,] v = e1, e2 [, e3]
      ...
label ...
```

or

```
      DO v = e1, e2 [, e3]
      ...
      END DO
```

DESCRIPTION

label Specifies the statement number of an executable statement that appears below the DO statement in the same program unit.

v Specifies the DO variable and is of type integer, real, or double precision.

e1 Specifies the initial value of the DO variable.

e2 Specifies the terminal value of the DO variable.

e3 Specifies the number by which the DO variable is incremented with each iteration of the loop.

The labeled statement is called the terminal statement of the DO loop. If the *label* specification is omitted, then the END DO statement is the terminal statement of the DO loop. The range of the DO loop is the sequence of statements from the DO statement up to and including the terminal statement of the loop.

The control parameters *e1*, *e2*, and *e3* are integer, real, or double precision expressions. If necessary, the data type of the value of *e1*, *e2*, and *e3* is converted to the data type of the DO variable. If *e3* is omitted, its value is 1. The control parameters have a positive or negative value, but the value of *e3* may not be zero. If *e1*, *e2*, or *e3* is an expression, it is evaluated the first time the DO statement is processed but is not reevaluated on subsequent iterations of the loop. The control parameters *e1*, *e2*, and *e3* should not be changed during the execution of the loop.

The number of times the loop is to be processed, the iteration count, is determined when the DO statement is executed for the first time. The iteration count is defined to be:

```
MAX(INT((e2 - e1 + e3) / e3), 0)
```

(The intrinsic functions `MAX` and `INT` are explained in Chapter 9.)

The iteration count is zero whenever:

e1 > *e2* and *e3* > 0

or

e1 < *e2* and *e3* < 0

During program execution, a `DO` loop is either active or inactive. The `DO` loop becomes active when its `DO` statement is executed. The `DO` loop becomes inactive when it has exited.

DO-UNTIL (H)

A DO-UNTIL block is a loop that executes at least once. The loop remains active or becomes inactive based on a test at the end of the range of the DO-UNTIL loop.

SYNTAX

```
DO
. . .
UNTIL (e)
```

DESCRIPTION

e Specifies a relational or logical expression. The parentheses must be specified as shown. The range of the loop is executed at least once, and a test is made at the end of the loop in the UNTIL statement. If the test is true, control is transferred to the statement that follows the UNTIL statement. If the test is false, the range of the DO-UNTIL loop is executed again.

Control may not be transferred to a statement within the range of the DO-UNTIL loop from outside the loop.

A DO-UNTIL loop in an IF, WHILE, FOR, DO WHILE, or LOOP block or in a DO loop must be wholly contained within that block or loop. An IF, WHILE, FOR, DO WHILE, or LOOP block or a DO loop within the range of a DO-UNTIL loop must be wholly contained within the range of the DO-UNTIL loop.

Example:

```
CHARACTER *1 BLANK, ICHAR
DATA BLANK /"␣"/
. . .
DO
CALL GETCHR(ICHAR)
UNTIL (ICHAR .NE.  BLANK)
```

DO WHILE (H)

A DO WHILE block executes as long as the specified condition in the DO WHILE statement is true.

SYNTAX

```
        DO WHILE (e)
        . . .
        END DO
```

or

```
        DO label [,] WHILE (e)
        . . .
label   . . .
```

DESCRIPTION

e Specifies a relational or logical expression. Execution of the DO WHILE statement causes evaluation of expression *e*. If the value of *e* is true, normal sequential execution continues and execution of the range of the DO WHILE begins. If the value of *e* is false, control is transferred to the statement immediately following the END DO or labeled statement that closes the DO WHILE block.

label Specifies the statement number of an executable statement that appears below the DO WHILE statement in the same program unit.

The DO WHILE block executes repeatedly as long as the specified condition is true. The test for the condition is made before each execution of the range of the DO WHILE; thus a DO WHILE block can be executed no times. DO WHILE blocks may be nested to any level but overlapping cannot occur. A DO WHILE block in an IF, DO-UNTIL, FOR, or LOOP block or within a DO loop must be wholly contained within that block or loop. An IF, DO-UNTIL, FOR, or LOOP block or DO loop within the range of a DO WHILE block must be totally within that DO WHILE block.

Example:

```
        DO WHILE (ICOUNT .NE.  0)
            . . .
            DO 10, WHILE (.NOT.  EOF)
                . . .
10          CONTINUE
            . . .
        END DO
```

EXIT DO (H)

Allows a conditional or unconditional exit from a DO-UNTIL block.

The EXIT DO statement allows a conditional or unconditional exit from a DO-UNTIL block but may not be used to exit from any other type of DO loop.

SYNTAX

```
EXIT DO [IF (e)]
```

DESCRIPTION

e Specifies the relational or logical expression that permits the DO-UNTIL block to be exited conditionally depending on whether the value of *e* is true or false. The parentheses surrounding *e* must be specified as shown.

One or more EXIT DO statements may be placed in a DO block. If an EXIT DO is encountered during execution, the innermost DO-UNTIL block becomes inactive, and control is transferred to the statement following the UNTIL statement that closes the DO-UNTIL block. If an IF phrase is not specified, the exit is unconditional. If an IF phrase is specified and the logical expression *e* is true, the DO-UNTIL loop is exited. If *e* is false, an exit is not taken and the loop execution continues with the next statement after the EXIT DO.

Example:

```
LOGICAL TEST
DO
. . .
EXIT DO IF (TEST)
. . .
UNTIL (I .GE.  25)
```

FOR Statements (H)

FOR (H)

A FOR block functions exactly like a DO loop except the terminal statement of the FOR block is an END FOR statement rather than a labeled statement, and a label is not specified in the FOR statement.

SYNTAX

```
FOR v = e1, e2 [, e3]
    ...
END FOR
```

DESCRIPTION

v Specifies the FOR variable and is of type integer, real, or double precision.

e1 Specifies the initial value of the FOR variable.

e2 Specifies the terminal value of the FOR variable.

e3 Specifies the number by which the FOR variable is incremented with each iteration of the loop.

FOR blocks may be nested to any level; that is, a FOR block may be included in another FOR block, which may be included in another FOR block, etc. Overlapping of FOR blocks is not permitted. A FOR block within the range of another FOR block must be totally within the range of the outer FOR block. An END FOR statement closes only the innermost FOR block; thus, an END FOR must be present for each FOR statement. Examples:

```
FOR I = 1, 10
    . . .
    FOR INVAR = J, K, L
    . . .
        FOR J = 10, -10, -1
            . . .
        END FOR
    END FOR
END FOR
```

A FOR block in an IF, WHILE, DO-UNTIL, DO WHILE, or LOOP block or in a DO loop must be wholly contained within that block or loop. An IF, WHILE, DO-UNTIL, DO WHILE, or LOOP block or a DO loop within the range of a FOR block must be totally within that FOR block.

```
DIMENSION A (10,10)
. . .
FOR I = 1, 9
    I1 = I + 1
    FOR J = I1, 10
        TEMP = A(I,J)
        A(I,J) = A(J,I)
        A(J,I) = TEMP
    END FOR
END FOR
```

This computes the transpose of a 10 by 10 matrix A. Note that the two `END FOR` statements are needed to close the two `FOR` blocks; the first `END FOR` statement closes the inner `FOR` block with J as its loop variable, and the second `END FOR` statement closes the outer `FOR` block with I as its loop variable.

EXIT FOR (H)

The EXIT FOR statement allows a conditional or unconditional exit from a FOR block.

SYNTAX

```
EXIT FOR [IF (e)]
```

DESCRIPTION

e Specifies the relational or logical expression that permits the FOR block to be exited conditionally depending on whether the value of *e* is true or false. The parentheses surrounding *e* must be specified as shown.

One or more EXIT FOR statements may be placed in a FOR block. If an EXIT FOR is encountered during execution, the innermost FOR block becomes inactive, and control is transferred to the statement following the END FOR statement that closes the FOR block. If an IF phrase is not specified, the exit is unconditional. If an IF phrase is specified and the logical expression *e* is true, the FOR loop is exited. If *e* is false, an exit is not taken and the loop execution continues with the next statement after the EXIT FOR.

GO TO Statements

Unconditional GO TO

An unconditional `GO TO` statement transfers control to another statement in the same program unit.

The statement is called *unconditional* because a transfer of control occurs every time the `GO TO` is executed.

SYNTAX

`GO TO` *label*

DESCRIPTION

label Specifies the statement number of an executable statement elsewhere in the program unit. The labeled statement may precede or follow the unconditional `GO TO` statement. Normal execution continues with the statement whose statement number is label.

Examples:

```
      GO TO 105
      . . .
      GO TO 105
      . . .
      GO TO 105
      .   . .
105   CONTINUE
      . . .
      END
```

Control is immediately transferred to the statement labeled 105 if any of the `GO TO` statements is executed.

Computed GO TO

A computed GO TO permits a user to define alternate locations to branch to based on the evaluation of an expression.

SYNTAX

```
GO TO (s [, s] ...)[,] e
```

DESCRIPTION

s Specifies a statement label that appears elsewhere in the same program unit. Each *s* must be separated by a comma. The labeled statements may precede or follow the computed GO TO statement. The same statement label may appear more than once in the list of statement labels. The list of statement labels must be enclosed in parentheses.

e Specifies an arithmetic expression.

If the value of *e* is a non-integer numeric value, the number is converted to an integer.

When *e* is evaluated, control is transferred to the statement whose statement label is the *e*th statement label in the list of statement labels. For a computed GO TO to have any effect, the value of *e* must be between 1 and the total number of labels specified in the list of statement labels. If *e*'s value is 0, a negative number, or a number greater than the number of labels in the list, the computed GO TO has no effect, and execution continues with the statement that follows the computed GO TO.

Example:

```
GO TO (10, 20, 30, 30, 40) I + J / 2
```

Transfers control to the statement labeled 20 if I+J/2 equals 2, to the statement labeled 30 if I+J/2 equals 3 or 4, etc.

Assigned GO TO

Assigned `GO TO` statements transfer control to a location in the same program unit based on the value assigned to a variable in an `ASSIGN` statement (see "ASSIGN Statement" on page 3-30).

SYNTAX

```
GO TO i [ [,] (s [, s] ...)]
```

DESCRIPTION

i Specifies a four-byte integer variable name defined in an `ASSIGN` statement.

s Specifies the label of an executable statement that appears elsewhere in the same program unit, either above or below the assigned `GO TO` statement. If one or more statement labels are specified, the labels must be enclosed in parentheses, and the labels must be separated by commas.

The variable *i* must have been previously defined in an `ASSIGN` statement in the same program unit, and the destination of the transfer of control is the statement label identified in the most recently executed `ASSIGN` statement that specified the variable *i*.

The same statement label can appear more than once in the list of statement labels. If a list of labels is not specified, then the list of possible labels is assumed to be all statement numbers that appear in an `ASSIGN` statement. If a list of labels is specified, then the value of *i* must be one of the labels in the list; if not, a warning is not issued, but the compiled program can be erroneous.

Examples:

```
ASSIGN 5371 TO LOCUS
GO TO LOCUS
GO TO LOCUS, (117, 56, 101, 5371)
```

The two `GO TO` statements transfer control to the statement labeled 5371.

IF Statements

Arithmetic IF

An arithmetic `IF` branches to one of three specified locations depending on the result of a condition.

SYNTAX

```
IF (e) s1, s2, s3
```

DESCRIPTION

e Specifies an integer, real, or double precision expression, enclosed in parentheses, that returns a positive, negative, or zero value. A complex or double complex expression for *e* is not permitted.

s1,s2,s3 Each specifies a statement label of an executable statement that appears elsewhere in the same program unit. The same statement label may appear more than once in the arithmetic `IF`.

Once *e* is evaluated, control is transferred to the statement labeled *s1* if *e* is less than 0, to statement *s2* if *e* equals 0, or to statement *s3* if *e* is greater than 0. The transfer may be to a statement that precedes or follows the arithmetic `IF` in the program unit.

Examples:

```
IF (I / J) 10, 20, 30
```

Caution should be exercised if *e* is a real or double precision expression, and the expression involves a computation whose mathematical value is zero. A small number that is not exactly zero is likely to result and a branch to *s2* would not be taken.

Statement	Value of *e*	Transfers to Statement
IF (K) 1,2,3	47802	3
`IF (3 * M(J) - 7) 76,4,3`	-6	76
IF (C(J,10) + A/4) 23,10,12	0.0002	12
IF (K*N**2 - 14*LIMIT) 78,444,78	-1000	78
IF (Z-B-3.1416 + SQRT(X-2)) 3,3,7	23.40669	7

Logical IF

A logical IF statement executes a statement conditionally, depending on whether the logical IF is true or false.

SYNTAX

IF (*e*) *stmt*

DESCRIPTION

e — Specifies a relational or logical expression enclosed in parentheses.

stmt — Specifies any executable statement except a DO, DO-UNTIL, DO WHILE, END DO, LOOP, END LOOP, WHILE, END WHILE, FOR, END FOR, SELECT CASE, block IF, END IF, ELSE IF, ELSE, END, or another logical IF.

NOTE: The symbolic name "THEN" must not be used as a keyword with a logical IF statement.

If *e* results in a true value, *stmt* is executed and then the next sequential statement after the logical IF is executed (unless *stmt* is a transfer of control). If *e* results in a false value, *stmt* is not executed, and execution continues with the next sequential statement after the logical IF.

The execution of a function reference in *e* can affect entities in *stmt*.

Examples:

```
IF (FLAG .OR.  L) GO TO 3000
IF (W .OR.  N .LT.  U/S + X3(J,K)) R(J-8) = Q * ABS(X)
IF (OCTT * TRR .LT.  5.334E4) CALL THERML (N, Y(L,5))
```

Block IF

A block `IF` statement defines a sequence of statements to be executed conditionally if a condition is true, defines an `IF-THEN-ELSE` sequence, or defines a case selection facility.

SYNTAX

```
!  The following form defines a sequence of statements
!  to be executed conditionally if a condition is true
IF (e)  THEN
.  . .
END IF



!  The following form defines an IF-THEN-ELSE sequence
IF (e)  THEN
.  . .
ELSE
.  . .
END IF



!  The following form defines an IF-THEN-ELSEIF facility
IF (e1) THEN
    !   "Then" clause (Case 1)
ELSE IF (e2) THEN
    !   "Else if" clause (Case 2)
.
.
ELSE IF (en) THEN
    !  "Else if" clause (Case n)
ELSE
    !  "default" clause (if no previous test is true)
END IF
```

DESCRIPTION

e Specifies a relational or logical expression enclosed in parentheses.

EXIT IF (H)

The EXIT IF statement allows a conditional or unconditional exit from any clause in an IF block.

SYNTAX

```
EXIT IF [IF (e)]
```

DESCRIPTION

e Specifies a relational or logical expression that permits the IF block to be exited conditionally depending on whether the value of *e* is true or false. The parentheses surrounding *e* must be specified as shown.

One or more EXIT IF statements can be placed in an IF block. If an EXIT IF is encountered during execution, the innermost IF block becomes inactive, and control is transferred to the statement following the next END IF statement at the same IF control level as the EXIT IF statement. If an IF phrase is not specified, the exit is unconditional. If an IF phrase is specified and the logical expression *e* is true, the IF block is exited. If *e* is false, an exit is not taken and the loop execution continues with the next statement after the EXIT IF.

An EXIT IF statement can be used in a THEN, ELSE IF, or ELSE clause to exit the IF block.

LOOP Statements (H)

LOOP (H)

A LOOP block executes a specified number of times or executes repeatedly until the loop is exited from within.

SYNTAX

```
LOOP [ (e) ]
.  . .
END LOOP
```

DESCRIPTION

e Specifies a constant, the symbolic name of a constant, a variable, an array element name, or an arithmetic expression. The *e* defines the number or times the range of the loop is to be executed, i.e., the iteration count. If the value of *e* is a real or double precision number, it is converted to an integer. If *e* has a negative or zero value, the loop is not executed. A complex or double complex value for *e* is not permitted. If specified, *e* must be enclosed in parentheses. If *e* is omitted, the range of the loop executes repeatedly until a statement from within the loop transfers control outside the loop.

Function references or subroutine calls do not transfer control out of the loop unless an alternate return is executed. LOOP blocks can be nested to any level but overlapping cannot occur.

Example:

```
LOOP (K / J)
     . . .
     LOOP (100)
          . . .
          LOOP
               . . .
          END LOOP
     END LOOP
END LOOP
```

A LOOP block in an IF, WHILE, DO-UNTIL, DO WHILE, or FOR block or within a DO loop must be wholly contained within that block or DO loop. An IF, WHILE, or FOR block or DO loop within the range of a LOOP block must be totally within that LOOP block.

EXIT LOOP (H)

The `EXIT LOOP` statement allows a conditional or unconditional exit from a `LOOP` block.

SYNTAX

```
EXIT LOOP [IF (e)]
```

DESCRIPTION

e Specifies a relational or logical expression that permits the `LOOP` block to be exited conditionally depending on whether the value of *e* is true or false. The parentheses surrounding *e* must be specified as shown.

One or more `EXIT LOOP` statements may be placed in a `LOOP` block. If an `EXIT LOOP` is encountered during execution, the innermost `LOOP` block becomes inactive, and control is transferred to the statement following the `END LOOP` statement that closes the `LOOP` block. If an `IF` phrase is not specified, the exit is unconditional. If an `IF` phrase is specified and the logical expression *e* is true, the loop is exited. If *e* is false, an exit is not taken and the loop execution continues with the next statement after the `EXIT LOOP`.

An `EXIT LOOP` statement may not be used to exit from any other iterative procedure.

PAUSE Statement

The PAUSE statement suspends execution of the source program. More than one PAUSE statement may appear in a program unit.

SYNTAX

```
PAUSE [literal]
```

DESCRIPTION

literal Specifies a string of decimal digits or a character string. If the character string contains blanks or special characters, the string must be enclosed in single or double quotes. If the literal begins with a digit, only an INTEGER number may be specified. The literal is written to the user's terminal.

The resumption of execution occurs when the user responds to the PAUSE prompt at the terminal. Once execution resumes, processing continues as if CONTINUE were specified, and there had been no effect on the execution of the program.

SELECT CASE Statements (H)

SELECT CASE (H)

The SELECT CASE statement begins the SELECT CASE construct. It contains a selection expression to be evaluated.

SYNTAX

```
SELECT CASE expression
CASE case-expression
   . . .
CASE case-expression
   . . .
CASE DEFAULT
   . . .
END SELECT
```

DESCRIPTION

expression
: Specifies a selection expression of any data type, including CHARACTER. The expression must represent a scalar value.

case-expression
: A comma-separated list of expressions against which the selection expression is compared. See "CASE (H)" on page 5-28 for format details.

When SELECT CASE is executed, *expression* is evaluated. The value of *expression* is compared with each *case-expression* in the order the *case-expressions* appear in the source. Control is transferred to the first statement following a matching CASE *case-expression*. If no matching *case-expression* is found, control is transferred to the first statement following CASE DEFAULT or ELSE. If no CASE DEFAULT or ELSE statement is present, control is transferred to the first statement following the END SELECT statement.

CASE (H)

The CASE statement appears within a SELECT CASE construct. The CASE statement precedes a block of zero or more statements to be selectively executed.

The CASE statement contains one or more constant expressions representing a value or range of values. The value of the selection expression is compared with each value or range of values in left-to-right order of their appearance in the source. If the value of the selection expression matches a value or falls within a range of values, control is transferred to the first statement following the CASE statement containing the matching constant expression. When the final statement of the block is executed, or if there are no statements within the block, control is transferred to the first executable statement following the END SELECT statement of the SELECT CASE construct.

The data type of the constant expressions should match the data type of the corresponding SELECT CASE selection expression. If the data types do not match, data type conversion rules are followed. (See "Data Type Conversions (Mixed Modes)" on page 3-5 for details.)

SYNTAX

CASE *value-or-range-of-values* [,*value-or-range-of-values*]
statement-list

DESCRIPTION

value-or-range-of-values
(REQUIRED) Represents a single value or an inclusive or exclusive range of values to compare with the selection expression. Ranges are valid for all numeric data types, and CHARACTER, for which the comparisons are lexical. Ranges are not valid for LOGICAL data type. The following forms are recognized.

value-or-range-of-values	Selection expression matches if...
value	Equal to *value*.
lower : *upper*	Greater than or equal to *lower*, and less than or equal to *upper*.
(*lower* : *upper*)	Greater than *lower*, and less than *upper*.
(*lower* : *upper*	Greater than *lower*, and less than or equal to *upper*.
lower : *upper*)	Greater than or equal to *lower*, and less than *upper*.
lower :	Greater than or equal to *lower*.
(*lower* :	Greater than *lower*.
: *upper*	Less than or equal to *upper*.
: *upper*)	Less than *upper*.

statement-list

Zero or more statements to be executed if the `CASE` statement is executed and the selection expression matches any of *value-or-range-of-values*.

Example:

```
INTEGER ARR(50), I
PARAMETER (ERR = -1)
   .  .  .
SELECT CASE ARR(I)
CASE ERR
   PRINT *, "ARR(I) matches ERR"
CASE 0, 3:5, 17
   PRINT *, "ARR(I) is one of 0, 3, 4, 5, 17"
CASE (6:6*2)
   PRINT *, "ARR(I) is between 6 and 12 but not 6 or 12"
CASE :ERR)
   PRINT *, "ARR(I) is less than ERR"
CASE DEFAULT
   PRINT *, "ARR(I) did not match any case expression"
END SELECT
```

CASE DEFAULT or ELSE (H)

An CASE DEFAULT or ELSE statement appears within a SELECT CASE construct. The CASE DEFAULT or ELSE statement precedes a block of statements to be executed if no matching *value-or-range-of-values* (see "CASE (H)" on page 5-28) is found.

There may be one optional CASE DEFAULT or ELSE statement per SELECT CASE construct. If no matching *value-or-range-of-values* is found, control is transferred to the first executable statement following the CASE DEFAULT or ELSE statement. When the final statement of the block is executed, control is transferred to the first executable statement following the END SELECT statement of the SELECT CASE construct.

SYNTAX

```
CASE DEFAULT
    statement-list
```

statement-list
Zero or more statements to be executed if no other CASE statement is executed.

See "CASE (H)" on page 5-28 for an example.

END SELECT (H)

An `END SELECT` statement terminates a `SELECT CASE` construct.

When the final statement of a block of statements associated with a `CASE`, `CASE DEFAULT` or `ELSE` statement is executed, control is transferred to the first executable statement following the `END SELECT` statement of the `SELECT CASE` construct.

See "CASE (H)" on page 5-28 for an example.

STOP Statement

The STOP statement terminates the execution of a running program. More than one STOP statement may appear in a program unit.

SYNTAX

STOP [*literal*]

DESCRIPTION

literal Specifies a string of decimal digits or a character string. If the character string contains blanks or special characters, the string must be enclosed in single or double quotes. If the literal begins with a digit, only an INTEGER number may be specified. The literal is written to the user's terminal.

When STOP is executed, the message "STOP literal statement executed" is written to stderr. *literal* is the integer or character string supplied with the STOP statement. If the literal is an integer, the exit status of the program is the integer value, otherwise the exit status is 0.

WHILE Statements (H)

WHILE (H)

A WHILE block executes as long as the specified condition in the WHILE statement is true.

SYNTAX

```
WHILE (e)
. . .
END WHILE
```

DESCRIPTION

e Specifies a relational or logical expression enclosed in parentheses. Execution of the WHILE statement causes evaluation of expression *e*. If the value of *e* is true, normal sequential execution continues and execution of the range of the WHILE begins. If the value of *e* is false, control is transferred to the statement immediately following the END WHILE or labeled statement that closes the WHILE block.

The WHILE block executes repeatedly as long as the specified condition is true. The test for the condition is made before each execution of the range of the WHILE; thus a WHILE block cannot be executed.

WHILE blocks can be nested to any level, but overlapping cannot occur. A WHILE block in an IF, DO-UNTIL, DO WHILE, FOR, or LOOP block or within a DO loop must be wholly contained within that block or loop. An IF, DO-UNTIL, DO WHILE, FOR, or LOOP block or DO loop within the range of a WHILE block must be totally within that WHILE block.

Examples:

```
WHILE (ICOUNT .NE.  0)
    . . .
    WHILE (.NOT.  EOF)
        . . .
        WHILE (T .LE.  TMAX)
            . . .
        END WHILE
        . . .
    END WHILE
    . . .
END WHILE


INTEGER I, SUM
I = 50
SUM = 100
WHILE (I .LT.  100)
    SUM = SUM + I
    I = I + 1
END WHILE
.  . .
END
```

The second example computes the sum of the integers between 50 and 100, inclusive.

EXIT WHILE (H)

The EXIT WHILE statement allows a conditional or unconditional exit from a WHILE block.

SYNTAX

```
EXIT WHILE [IF (e)]
```

DESCRIPTION

e Specifies a relational or logical expression that permits the WHILE block to be exited conditionally depending on whether the value of *e* is true or false. The parentheses surrounding *e* must be specified as shown.

One or more EXIT WHILE statements can be placed in a WHILE block. If an EXIT WHILE is encountered during execution, the innermost WHILE block becomes inactive, and control is transferred to the statement following the END WHILE statement that closes the WHILE block. If an IF phrase is not specified, the exit is an unconditional exit. If an IF phrase is specified and the logical expression *e* is true, the WHILE loop is exited. If *e* is false, an exit is not taken and the loop execution continues with the next statement after the EXIT WHILE.

An EXIT WHILE statement may not be used to exit from any other iterative procedure.

6
Fortran Input/Output

6
Fortran Input/Output

General Fortran I/O Information

Fortran input statements transfer (read) data stored on an external storage medium (e.g., disk storage) into memory, or transfer data from an internal file to memory. Input is performed using the `READ` statement. The non-standard `ACCEPT` statement is functionally and syntactically equivalent to the `READ` statement; however, its use is discouraged.

Fortran output statements transfer (write) data from memory to an external storage medium, or transfer data from memory to an internal file. Output is performed using the `WRITE` or `PRINT` statements. The non-standard `TYPE` and `PUNCH` statements are functionally and syntactically equivalent to the `WRITE` and `PRINT` statements; however, their use is discouraged.

Input and output statements are classified as:

- Formatted (sequential and direct access)
- Unformatted (sequential and direct access)
- List-directed (free-format)
- Namelist-directed

Table 6-1 compares the syntaxes of the American National Standard (ANSI) Fortran input statements. Table 6-2 compares the syntaxes of the ANSI Fortran output statements. Both tables refer to several specifiers. For information about the following specifiers, see the corresponding section.

u	Specifies a unit number or character entity. See "UNIT Specifier" on page 6-15
f	Specifies a format specifier. See "Format Specifier" on page 6-10
g	Specifies the group-name of a namelist. See "Namelist Specifier (H)" on page 6-14
`END=`*s*	Indicates a statement label to branch to on an end-of-file condition. See "END Specifier" on page 6-9
`ERR=`*s*	Indicates a statement label to branch to on an I/O error condition. See "ERR Specifier" on page 6-9
`IOSTAT=`*ios*	Indicates the status of the last I/O operation. See "IOSTAT Specifier" on page 6-11

REC=*n*	Specifies the next record number to be read or written. See "REC Specifier" on page 6-14
list	Specifies an input or output list of variables, subscripted array names, unsubscripted array names, or expressions. See "Input/Output Lists" on page 6-15

Table 6-1. Comparison of Input-Statement Syntaxes

Category	Syntax	Page Number
Formatted Sequential	READ (*u*, *f* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*] READ *f* [,*list*]	6-19
Unformatted Sequential	READ (*u* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*]	6-22
List-Directed	READ (*u*, * [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*] READ * [,*list*]	6-24
Namelist-Directed	READ (*u*, *g* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*])	6-28
Formatted Direct Access	READ (*u*, *f*, REC=*n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*] READ (*u'n*, *f* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*]	6-32
Unformatted Direct Access	READ (*u*, REC=*n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*] READ (*u'n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*]	6-34

Table 6-2. Comparison of Output-Statement Syntaxes

Category	Syntax	Page Number
Formatted Sequential	WRITE (*u*, *f* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*] PRINT *f* [,*list*]	6-20 6-21
Unformatted Sequential	WRITE (*u* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*]	6-23
List-Directed	WRITE (*u*, * [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*]) [*list*] PRINT * [,*list*]	6-26
Namelist-Directed	WRITE (*u*, *g* [,ERR=*s*] [,END=*s*] [,IOSTAT=*ios*])	6-31
Formatted Direct Access	WRITE (*u*, *f*, REC=*n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*] WRITE (*u'n*, *f* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*]	6-33
Unformatted Direct Access	WRITE (*u*, REC=*n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*] WRITE (*u'n* [,ERR=*s*] [,IOSTAT=*ios*]) [*list*]	6-35

Records

Any input or output statement that transfers data processes one or more "records" with each execution of the input or output statement. A data *record* is a sequence of one or more data values. The length of a record is the number of bytes (i.e., characters) in the record.

The maximum length of a record depends on the medium on which the records are stored. For example, records read from or written on cards cannot exceed 80 characters, while records printed on a line printer are limited to the size of a line that the printer can print.

An end-of-file record denotes the end of a data file and is written by an `ENDFILE` statement. An end-of-file record does not contain data and is usually the last record of a file. When an end-of-file record is read, no data is transmitted, but an end-of-file condition is raised if the end-of-file record is the last record in the file.

The Fortran standard does not permit I/O after an `ENDFILE` operation unless it is preceded by a backspace or a `REWIND` operation. The Fortran I/O library permits write operations to be performed after an `ENDFILE`. The `ENDFILE` truncates the file, but there is no physical embodiment of an end-of-file record.

A null record contains no data and is neither defined nor undefined. Unformatted `WRITE` statements without an output list write null records. A null record can be read only with an unformatted `READ` statement that contains no input list.

External and Internal Files

A *file* is a collection of records and is external or internal.

External files are read from or written to an external device (e.g., a line printer, magnetic tape drive, magnetic disk drive, etc.). An external file can be empty.

When an external file is to be read or written, the input or output statement for the file must refer to a unit number which references a file on an external device defined for your Concurrent system.

An *internal file* is used to transfer data from one location in memory to another location in memory and is used to convert data from one form to another (e.g., from numeric to character form). A `READ` or `WRITE` statement specifying a character entity as the unit specifier (see "I/O Statements for Reading and Writing" on page 6-7) is used in conjunction with a format specification to edit and convert the data.

Units

Neither input nor output can be performed on a file until the file is "connected" to a unit.

With Concurrent Fortran, a file is *preconnected* to certain units by the compiler or is connected explicitly when an `OPEN` statement is executed. An `OPEN` statement can be specified for files that are preconnected, but the `OPEN` statement is not necessary.

Units 5, 6, and 0 are preconnected when the program starts. Unit 5 is connected to the standard input (`stdin`), unit 6 is connected to the standard output (`stdout`), and unit 0 is connected to the standard error (`stderr`) unit. To preserve error reporting, it is illegal to close unit 0, although it may be reopened to another file.

Redefining the standard units may impair console I/O. An alternative is to use shell redirection to externally redefine the above units. To redefine default blank control or format of the standard input or output files, use the `OPEN` statement specifying the unit number and no file name.

`stdin`, `stdout`, and `stderr` are not actual file names, and cannot be used for opening these units. `INQUIRE` does not return these names and indicates that the above units are not named unless they are open for real files.

All other units are also preconnected when execution begins. Unit *n* is connected to a file named **`fort.n`**. These files need not exist nor are they created unless their units are used without first executing an `OPEN`. All default preconnections are for sequential formatted I/O. The `ioinit` subroutine may be used to override this convention; see the man page for **`ioinit(3F)`**.

All input/output statements refer to units that are connected. The `OPEN`, `CLOSE`, and `INQUIRE` statements are the only statements that refer to unconnected units and to files that have not been created.

A unit is connected only to one file at a time; however, a file can be disconnected from one unit and then reconnected to the same unit or to another unit.

A unit is disconnected when a `CLOSE` statement for the unit is executed, when an `OPEN` statement associating the same unit to a different file is executed, or when program execution is terminated.

Once a unit is connected or disconnected, it is connected or disconnected for all program units in the source program.

The maximum number of units that a program may open at one time is same as the system limit. Unit numbers must be in the range of 0 to 255.

Vertical Format Control

If a unit is open for sequential access with `FORM='print'` specified, then control codes 0 and 1 are replaced in the output file with `\n` and `\f`, respectively. The control character + is not implemented and, like any other character in the first position of a record written to a print file, is dropped. Vertical format control is not recognized for direct formatted output or list-directed output.

File Organization

Every external file is defined to have a certain file organization which controls the order in which records of the file are retrieved or written. This organization is called the file's *access method* and is dependent on the properties of the storage medium. Fortran permits both sequential access and direct access files.

Sequential Access

Records in *sequential access* files are written one after the other and are retrieved in the order in which they were written. The records are either all formatted or all unformatted. Records of different lengths must not be read or written except with sequential I/O statements.

List-directed I/O statements are sequential access I/O statements.

Direct Access

Records in *direct access* files must have the same lengths and can be retrieved or written only by direct access I/O statements. List-directed input and output statements cannot read or write direct access records.

Records in direct access files are written or read in any order. For example, record 5 can be written first, followed by record 1, followed by record 23, etc. Any record can be retrieved as long as it has previously been written.

Each direct access record is identified in an input or output statement by its record number. The record number is a positive integer. Once a record is created, the record number cannot be changed, and the record cannot be deleted; however, the record can be rewritten.

If a sequential file can be accessed directly, its end-of-file, if any, is not considered to be part of the file while it is connected for direct access.

File Position

A file that is connected to a unit is positioned at a certain location in the file. This position can change when various input or output statements are executed.

The *initial point* of a file is the position just before the first record of the file.

The *terminal point* of a file is the position just after the last record of the file.

The *current record* is the record at which the file is currently positioned for reading or writing. If the file is positioned at the initial or terminal point, there is no current record.

The *preceding record* is the record just before the current record. If the first record is the current record or if the file is positioned at its initial point, there is no preceding record. If the file is positioned at its terminal point, the last record of the file is the preceding record.

The *next record* is the record just after the current record. If the last record is the current record or if the file is positioned at the terminal point, there is no next record. If a file is positioned at its initial point, the first record is the next record.

Input and Output Using Internal Files

Output can be written to or read from files in memory. Such I/O is performed using internal files.

An *internal file* is a character variable, character array, character array element, or character substring specified as the unit in a `READ` or `WRITE` statement. The record length is the length of the variable, array element, array, or substring.

If the internal file is a character variable, character array element, or substring, the file consists of a single record. If the internal file is an entire array, each element of the array is a record of the file, each record has the same length, and the order of the records in the internal file is the same as the order of the elements in the array.

A variable, array element, or substring becomes defined by writing the record. A value that is too long to fit the record results in an error. A value that is too short results in blanks being appended to the end of the record to fill the record length. A record in an internal file is read only if it has been defined. Note that a record in an internal file can become defined or undefined in ways other than an output statement (e.g., with an assignment statement).

Unformatted I/O is not allowed for internal files.

The `OPEN`, `CLOSE`, `INQUIRE`, `ENDFILE`, `BACKSPACE`, and `REWIND` statements cannot refer to an internal file.

Example:

```
      INTEGER Q
      CHARACTER *16 INTFILE (100)
      DO 10 Q= 1,100
      WRITE (UNIT=INTFILE (Q), FMT='(I15)') Q*Q
10    CONTINUE
      DO 90 I=1,100,1
      PRINT, I, INTFILE(I)
90    CONTINUE
      END
```

Stores the squares of the whole numbers from 1 to 100 in the records of internal file `INFILE` and then prints the contents of `INTFILE`.

I/O Statements for Reading and Writing

Formatted I/O Statements

Records read using *formatted* input and output statements are edited on both input and output, and a format specification (see "Format Specification" on page 7-1) must be referenced in the formatted input or output statement. Formatted I/O statements read or write records in both sequential and direct access files.

Formatted I/O statements include:

- Formatted `READ` (sequential and direct access)
- Formatted `WRITE` (sequential and direct access)
- Formatted `PRINT` (sequential access only)

Data in records read or written by formatted I/O statements can be numeric, character, logical, or a mixture of the three types.

Rounding occurs on output for real, double precision, complex, and double complex data.

Unformatted I/O Statements

Unformatted input and output statements read or write records that consist of binary data; each record is a string of binary digits. No data translation or editing is performed when unformatted I/O statements are used; thus, a format specification is not specified.

Because data translation or editing is not performed, unformatted I/O is usually faster than formatted I/O. Also, unformatted I/O permits greater accuracy for numeric data because the representation of the data is identical.

Unformatted I/O statements read or write records in both sequential and direct access files.

Unformatted I/O statements include:

- Unformatted `READ` (sequential and direct access)
- Unformatted `WRITE` (sequential and direct access)

The length of an unformatted record is not restricted. For direct access files, unformatted records are fixed in length, but as many records as are necessary are read or written to accommodate the input or output list.

List-Directed I/O Statements

List-directed I/O (also called *free-format I/O*) permits data on one or more records to be read or written until all items in an input or output list are satisfied. List-directed I/O state-

ments do not require a FORMAT statement. Data editing is performed by the compiler and cannot be changed by the user.

List-directed input and output statements include:

- List-directed READ (sequential access only)
- List-directed WRITE (sequential access only)
- List-directed PRINT (sequential access only)

List-directed I/O statements cannot be used to read or write direct access files.

Internal list-directed I/O has been implemented. During internal list reads, bytes are consumed until the input list is satisfied or until the end-of-file is reached. During internal list writes, records are filled until the output list is satisfied. The length of an internal array element should be at least 20 bytes to avoid logical record overflow when writing double precision values. The internal list READ is implemented to make command line decoding easier. The internal list WRITE should be avoided.

Namelist-Directed I/O Statements (H)

Namelist-directed reads cause a series of data records to be read from the specified unit. Each record may contain a list of variable-name/value specifications, causing the variable to assume the specified value. Variables that may be updated in this manner are defined in a NAMELIST statement.

When a namelist-directed READ statement is executed, records are read sequentially from the specified unit until a header block with the specified group name is detected. The variables listed in the following data records are then set to the specified values, until a terminating block is located.

When a namelist-directed WRITE statement is executed, output is written to the specified unit in the format used by the namelist-directed READ statement.

Control Information List

The control information list consists of one or more control specifiers that appear in a READ or WRITE statement after the READ or WRITE keyword. The list is enclosed in parentheses in most cases, and control specifiers are separated by commas in the list.

Not all control specifiers apply to all types of READ and WRITE statements. The control specifiers that pertain to an input or output statement are shown in subsequent sections that discuss the syntax of particular I/O statements.

The term *specifier* denotes a control list value that must be specified in a particular position in the list of control information or denotes a specification in keyword form:

keyword=value

END Specifier

The end-of-file specifier identifies a statement label within the program unit, where control is to be transferred if an end-of-file condition is detected in an I/O statement. An end-of-file condition is raised on input when an attempt is made to read an end-of-file record. On output, an end-of-file condition is raised when a file becomes full (e.g., if a file system fills up).

SYNTAX

`END=`*s*

DESCRIPTION

s Specifies the statement number of an executable statement in the same program unit.

The end-of-file specifier is always optional. If `END` is specified, control is transferred to the statement identified by *s* if an end-of-file record is detected. The program is aborted if an end-of-file condition is detected and the program has not specified `END=` or `IOSTAT=`.

ERR Specifier

The error specifier identifies a statement label within the program unit, where control is to be transferred if an error occurs during the execution of an I/O statement.

SYNTAX

`ERR=`*s*

DESCRIPTION

s Specifies the statement number of an executable statement in the same program unit.

The error specifier is optional. If `ERR` is specified, control is transferred to the statement identified by *s* if an error occurs. If `ERR` is not specified, execution of the user's program terminates if an error is detected.

Format Specifier

A format specifier identifies a format for the input or output statement.

SYNTAX

f

FMT=*f*

DESCRIPTION

f Specifies an input or output format specification (see Chapter 7) and is one of the following:

- The statement number of a FORMAT statement in the same program unit.
- An integer variable (not an array name) whose value is the statement number of a FORMAT statement as defined in an ASSIGN statement in the same program unit.
- A character constant enclosed in single or double quotes.
- The name of a character variable, subscripted character array, or unsubscripted character array.
- A subscripted or unsubscripted array name containing Hollerith data.
- An asterisk (*) (list-directed I/O only).
- A character expression, except a character expression that concatenates an operand whose length specification is an asterisk, unless the operand is the symbolic name of a constant.

The format specifier must be the second item of the control information list if the FMT keyword is not specified, and the unit specifier, without the UNIT keyword, must be the first item of the list. If the FMT keyword is specified, the format specifier may appear anywhere in the control information list.

Example:

```
READ "(I5)", I
PRINT '("b", I5)', I
STOP
END
```

Reads an INTEGER constant from the input and prints it.

IOSTAT Specifier

The I/O status specifier returns a positive or zero integer indicating the status of the last I/O operation.

SYNTAX

`IOSTAT=`*ios*

DESCRIPTION

ios Specifies an integer variable or array element name whose value is:

- 0 I/O operation was successful (i.e., no error or end-of-file was detected).
- +*n* *n* is the positive Fortran error number. See the next section for details.

I/O Library Error Messages

The I/O library generates the following error messages. The error numbers are returned in the `IOSTAT=`*variable* if the `ERR=`*return* is taken. Error numbers less than 100 are generated by the kernel.

100 "error in format"
See error message output for the location of the error in the format. Can be caused by more than ten levels of nested () or an extremely long format statement.

101 "illegal unit number"
It is illegal to close logical unit 0. Negative unit numbers are not allowed. The upper limit is system-dependent.

102 "formatted i/o not allowed"
The logical unit was opened for unformatted I/O.

103 "unformatted i/o not allowed"
The logical unit was opened for formatted I/O.

104 "direct i/o not allowed"
The logical unit was opened for sequential access or the logical record length was specified as zero.

105 "sequential i/o not allowed"
The logical unit was opened for direct access I/O.

106 "can't backspace file"
The file associated with the logical unit cannot seek. It may be a device or pipe.

107 “off beginning of record”
The format specified a left tab beyond the beginning of an internal input record.

108 “can’t stat file”
The system cannot return status information about the file. Perhaps the directory is unreadable.

109 “no * after repeat count”
Repeat counts in list-directed I/O must be followed by an * with no blank spaces.

110 “off end of record”
A formatted write tried to go beyond the logical end-of-record. An unformatted read or write will also cause this error.

111 “truncation failed”
The truncation of an external sequential file on `CLOSE`, `BACKSPACE`, `REWIND`, or `ENDFILE` failed.

112 “incomprehensible list input”
List input must be correct.

113 “out of free space”
The library dynamically creates buffers for internal use. You ran out of memory for buffers. Your program is too big.

114 “unit not connected”
The logical unit was not open.

115 “read unexpected character”
Certain format conversions cannot tolerate non-numeric data. Logical data must be T or F.

116 “blank logical input field”

117 “’new’ file exists” You tried to open an existing file with `STATUS='NEW'`.

118 “can’t find ’old’ file”
You tried to open a non-existent file with `STATUS='OLD'`.

119 “unknown system error”
Should not happen, but ...

120 “requires seek ability”
Direct access requires seek ability. Sequential unformatted I/O requires seek ability on the file due to the special data structure required. Tabbing left also requires seek ability.

121 “illegal argument”
Certain arguments to `OPEN`, etc., must be legal. Often Fortran looks only for non-default forms.

122 “negative repeat count”
The repeat count for list-direct input must be a positive integer.

123 “illegal operation for unit”
An operation was requested for a device associated with the logical unit (which was not possible). This error is returned by the tape I/O routines if attempting to read past end-of-tape, etc.

149 “cannot keep a scratch file”
An attempt was made to keep a scratch file upon its closing.

150 “could not print file upon closing”
The file could not be printed upon its closing.

151 “could not print and delete file upon closing”
The file could not be printed and/or deleted upon its closing.

152 “cannot write to a read-only file”
An attempt was made to write to a read-only file.

153 “record number greater than maximum”
A record number larger than the specified maximum was used on a direct access file.

154 “cannot print a scratch file”
An attempt was made to print a scratch file.

155 “cannot delete a read-only file”
An attempt was made to delete a read-only file.

156 “unrecognizable data type”
The namelist dictionary references an unknown data type.

157 “invalid number of dimensions”
An invalid number of dimensions was specified in the namelist record.

158 “unable to find this name in the namelist block”
A name is present in the data file that is not a member of the namelist.

159 “bad value or value format”
Unable to read a valid value where one is expected.

160 “bad variable name or name format”
Unable to read a valid variable name, or an error was found in a subscript format.

161 “no terminating block for this namelist input record”
No terminating block (e.g., “`&END`”, “`$end`”, etc.) was found for this namelist record.

162 “unable to locate namelist header block”
Could not find a valid namelist header block.

Namelist Specifier (H)

A namelist specifier identifies the group-name and associates the namelist to be used in that I/O statement.

SYNTAX

NML=*g*

g

DESCRIPTION

g Specifies a group name of a namelist that has already been defined in a namelist statement.

The group name must be defined in the same program unit. The keyword NML= is optional only if the namelist specifier is the second parameter in the control list, and the first parameter is a logical unit specifier without the optional keyword UNIT=. A namelist specifier cannot be used in a statement that contains a format specifier. See "Namelist-Directed READ (H)" on page 6-28 and "Namelist-Directed WRITE (H)" on page 6-31.

REC Specifier

The record specifier identifies the next record to be read or written for direct access I/O.

SYNTAX

REC=*n*

'*n*

DESCRIPTION

n Specifies a positive integer, or an arithmetic expression indicating the absolute record number of the record to be read or written.

The maximum number of records in a direct access file cannot exceed the maximum size of a disk area on a disk pack.

The REC specifier must be present for direct access READ and WRITE statements. The REC specifier cannot be specified for files being accessed sequentially.

Refer to the sections in this chapter on direct access READ and WRITE for examples of the proper use of '*n*.

UNIT Specifier

The unit specifier identifies a unit number for external files or a character entity for internal files.

SYNTAX

u

UNIT=*u*

DESCRIPTION

u Specifies an asterisk (*), or the logical input file or output file.

For external files, the unit specifier is an asterisk or an integer arithmetic expression that results in a value from 0 through 255. An asterisk can be used only in READ, WRITE, and PRINT statements and means that the implicit unit number for input or output is to be used.

For internal files, the unit specifier is the name of a character variable, character array, character array element, or character substring.

Usually, the unit specifier appears as the first control specifier in the list and the UNIT keyword is omitted. If the unit specifier is not the first control specifier in the list, the UNIT keyword must be specified.

The UNIT keyword must be omitted in some cases (See "Format Specifier" on page 6-10).

A given unit specifier has the same meaning in all program units.

Input/Output Lists

An I/O list is a sequence of entities specified in an input or output statement that is to be read or written. Entities in an input or output list are separated by commas.

No entity or part of an entity in an input or output list can contain a function reference that references a function containing I/O statements.

Once variables are defined in the input or output list, they may be used as subscripts later in the input or output list. For example,

```
DIMENSION J(5)
READ, I, J(I)
PRINT, I, J(I)
END
```

A value is read for variable I and that value is used as a subscript reference for array J.

The subscript of an array can be an arithmetic expression in both input and output statements.

The name of an assumed-size array cannot appear in an input or output list without subscripts.

Input Lists

Zero or more entities are specified in an input list. An entity in an input list is a variable name, array name, array element name, or character substring name. If an unsubscripted array name is specified, values are read into the array as if each element were specified one after another in the order in which they are stored internally. Values are read into each entity in left-to-right order. Not specifying an input list causes an input record to be skipped.

Output Lists

Zero or more entities can be specified in an output list. An entity in an output is:

- A numeric, character, or logical constant
- A variable name
- A subscripted array name
- An unsubscripted array name
- A character substring name
- An expression, except a character expression that concatenates an operand whose length specification is an asterisk.

If an unsubscripted array name is specified, the array is treated as if each element were specified one after another in the order in which they are stored internally. Values are output in left-to-right order in the I/O list.

Implied-DO Lists

Implied-DO lists are used in input or output lists to establish a loop within the I/O list. Implied-DO lists are useful in reading or writing part of an array or in reading or writing elements of an array in a different order from the manner in which they are stored internally.

SYNTAX

```
(dlist, i = e1, e2 [, e3] )
```

DESCRIPTION

dlist Specifies an input or output list or another implied-DO list

i Specifies the name of an integer variable (the implied-DO variable).

e1,e2,e3 Specify integer expressions which contain implied-DO variables or variables from outer implied-DO lists that have this implied-DO within their ranges.

The range of an implied-DO is the list dlist. An iteration count and the values of the implied-DO variable are established from *e1*, *e2*, and *e3*, exactly as for a DO loop.

When an implied-DO list appears in an I/O statement, the list items in *dlist* are specified once for each iteration of the implied-DO list, with the appropriate substitution of values for any occurrence of the implied-DO variable *i*.

Expressions are permitted for subscripted array names or function references in *dlist*, and the expressions can contain the implied-DO variables of any implied-DO list that has the subscript expression within its range.

Nested implied-DO lists must be enclosed in parentheses and be wholly contained in the surrounding implied-DO list.

Sample Implied-DO Lists	Equivalent Simple List
`( X(I), I = 1, 4 )`	`X(1), X(2), X(3), X(4)`
`( A(J), B(J), J = 1, 3 )`	`A(1), B(1), A(2), B(2), A(3), B(3)`
`( G(2*N), N = 3, 9, 2 )`	`G(6), G(10), G(14), G(18)`
`T, ( C(J), J = 3, 5 ), E, LENGTH`	`T, C(3), C(4), C(5), E, LENGTH`
`( (A(I,J), I = 7, 9), J = 1, 3)`	`A(7,1), A(8,1), A(9,1), A(7,2), A(8,2), A(9,2), A(7,3), A(8,3), A(9,3)`
`( R, T(K), K = 2, 3 )`	`R, T(2), R, T(3)`

The DO variable can be used as a list item. For example:

```
( K, A(K), K = 1, 3 ), G(K)
```

is equivalent to the list:

```
1, A(1), 2, A(2), 3, A(3), G(4)
```

Implied-DO lists function like a DO statement and therefore can be executed no times. For example, the implied-DO:

```
( J, X(J), J = 10, 9, 1 )
```

results in data not being read or written.

Sequential I/O Statements

The peculiar requirements of sequential unformatted files make it unlikely that they will ever be read or written by any means except Fortran I/O statements. Each record is preceded and followed by an integer containing the record's length in bytes.

During a read, the Fortran I/O system breaks sequential formatted files into records after each newline. The 1977 standard does not define the result if a program reads past the end of a record. In general, the I/O system treats the record as being extended by blanks.

Logical records in sequentially accessed external files may be of arbitrary and variable length. The logical record length for unformatted sequential files is determined by the size of items in the input or output list. The requirements of this form of I/O cause the external physical record size to be somewhat larger than the logical record size. For formatted WRITE statements, the logical record length is determined by the format statement interacting with the output list at execution time.

While printing, the I/O system writes a newline at the end of each record. It is also possible for the program to write newlines. If a program writes a newline, no error occurs. The only effect, however, is that each record you write is treated as multiple records during subsequent records or backspacing.

Formatted Sequential READ

The formatted sequential READ statement transfers data in specified formats from the next record in an input data file to memory locations identified in the READ statement input list. On transfer, data is translated to internal form and is edited based on a format specification.

SYNTAX

```
READ f [, list]

READ ( u, f [, END=s] [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u — An input unit specifier. Format 1 does not specify a unit and the implied unit is unit number 5.

f — A format specifier other than an asterisk. The FMT keyword cannot be specified for format 1.

list — An input list. If the input list is omitted, the next record is skipped. Once a record is read or skipped by a sequential access READ statement, it cannot be read again until the file is repositioned.

If a record contains more data than the formatted READ statement needs, additional data is ignored. Each READ causes a new record to be read. If the READ statement's input list requires more data than is available on an input data record, an error results. The format specification can indicate that more than one record is to be read.

Examples:

```
READ 100
READ 100, A, B, C
READ (11, FMT=ARRAY) TOTAL
READ ( UNIT=*, FMT='(A)' ) CHRS
READ (FMT=1001, UNIT=7, END=99) X, Y, Z
```

Formatted Sequential WRITE

The formatted sequential WRITE statement transfers the values of entities in an output list in specified formats to the next record in a specified external or internal file. On transfer, data is translated to external form and is edited based on a format specification.

SYNTAX

```
WRITE ( u, f [, END=s] [, ERR=s] [, IOSTAT=ios] ) [list]
```

DESCRIPTION

u An output unit specifier.

f A format specifier other than an asterisk.

list An output list. If the output list is omitted, data stored in the format specification is written. If data is not present in the format specification, a blank record is written. Once a record is written by a sequential access WRITE statement, it cannot be rewritten or read until the file is repositioned. All entities in the list must be defined on output or an error occurs and data is not written.

The END specifier can be used to trap an end-of-file condition that results when a disk pack fills up or a file becomes too large.

Examples:

```
WRITE (6, 102, ERR=99, IOSTAT=IOS) A, B, C, 4*D+C
WRITE (6, FMT=10) VALUE
WRITE (6, 500)
WRITE (END=99, UNIT=6, FMT=ARRAY) (I(M),J(M),M = 1,15,2)
WRITE ( *, FMT='("b", A)' ) CSTRING
```

Formatted PRINT

The formatted `PRINT` statement functions like the formatted sequential `WRITE` statement except the unit specifier is omitted and defaults to unit 6.

SYNTAX

```
PRINT f [, list]
```

DESCRIPTION

f A format specifier other than an asterisk. The `FMT` keyword cannot be specified.

list An output list.

The unit specifier is omitted and the implied unit is unit number 6. All rules and restrictions pertaining to the formatted sequential `WRITE` statement apply to formatted `PRINT` statements.

Examples:

```
PRINT 10, A * B * C, L
PRINT '(A)', CSTRING
PRINT 100
```

Unformatted Sequential READ

The unformatted sequential READ statement transfers (without editing) binary data from the next record in an input file to specified memory locations. Unformatted READ statements read only records previously written by unformatted WRITE statements.

SYNTAX

```
READ ( u [, END=s] [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u A unit specifier.

list Specifies an input list. The amount of data read from the input record is determined by the length of the variables and array elements in the input list. If the input list is omitted, one input record is skipped.

Each unformatted READ reads exactly one record, and each read causes a new record to be read. If data in the record is not needed by the input list, additional data are ignored. If the input list requires more data from the input record than is available, an error results.

Examples:

```
READ (6) M, N, O
READ (END=99, UNIT=6) IN1, KN2, LN3
```

Unformatted Sequential WRITE

The unformatted sequential `WRITE` statement transfers (without editing or translation) binary data from specified program variables to the next record in an output file. Unformatted `WRITE` statements write only records to storage media capable of handling binary data. Records written by unformatted `WRITE` statements should be read only by unformatted `READ` statements.

SYNTAX

```
WRITE ( u [, END=s] [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u A unit specifier.

list An output list. The size of the written record is the total length of all variables and array elements in the output list. If the output list is omitted, a null record is written.

Each unformatted `WRITE` writes exactly one unformatted record.

Examples:

```
WRITE (UNIT=11, END=99) A, B, C
WRITE (6, IOSTAT=IOS) OUTFIELD
```

List-Directed READ

The list-directed (free-format) READ statement transfers (without editing) symbolic data (i.e., ASCII characters) from the next record in an input file to specified memory locations.

SYNTAX

```
READ ( u, * [, ERR=s] [, END=s] [, IOSTAT=ios] ) [ list ]

READ * [, list]
```

asterisk (*) indicates the omission of a format specifier.

DESCRIPTION

u A unit specifier.

list Specifies an input list. The amount of data read from the input record is determined by the length of the variables and array elements in the input list. If the input list is omitted, one input record is skipped.

Examples:

```
READ *, A, B, C
READ, A, B, C
READ (7, *) CL, CS
```

Format of List-Directed Input Data Records

Data values that are read using a list-directed READ statement are separated by a comma, one or more blanks, or a comma and one or more blanks.

The following considerations apply when list-directed reading is done:

Input data must be of the same data type as the variable or array into which it is being read.

Character data can be read only into character variables or arrays. Character values may be enclosed in single or double quotes. A character value cannot be empty. A single quote can be included in a string enclosed by single quotes, by specifying two adjacent single quote marks. Similarly, a double quote can be included in a string enclosed by double quotes, by specifying two adjacent double quote marks. If the character value is smaller or larger than its corresponding entity in the input list, blanks are appended or additional characters are truncated, respectively.

The list-directed READ has been extended to allow input of a string not enclosed in quotes. The string must not start with a digit and cannot contain a separator (, or /) or blank (space or tab). A newline terminates the string unless escaped with \\. Any string not meeting the above restrictions must be enclosed in single or double quotes.

Data are read from one or more records until every item in the input list is satisfied. Character values may be split over more than one input record as needed. Complex numbers may be split between lines only if the split does not occur within the real or imaginary number, or immediately before the comma. Real, integer, and logical data may not be split between lines since the digits at the end of the first line and the remaining digits at the beginning of the next line are considered to be two separate values. Blanks are never used as zeroes, and embedded blanks are not permitted in constants, except within character and complex constants. The end of a record has the effect of a blank, except when it appears within a character constant.

Integer data must not contain a decimal point.

Real, double precision, complex, and double complex data may be input in any of the forms acceptable under a formatted `READ` statement. If an exponent is present in the data, it must contain an `E` or `D`. The decimal point is optional and, if omitted, is assumed to be to the right of the last digit of the datum.

Complex items are input in one of the following forms:

(r,i)
r,i

without the parentheses. The real or imaginary part of a complex constant can be preceded or followed by one or more blanks.

If a logical variable or array is to be true, it must read a value whose first character is a `T`; a value that begins with any other character results in the variable or array element being false.

A new input record is read each time a list-directed `READ` statement is executed.

Two consecutive commas on an input record denote a null value. A null value indicates the contents of the variable is not changed.

If a slash (`/`) is read as input, execution of the list-directed `READ` statement stops, and the remaining entities retain their previous values. If the remaining entities have not been previously initialized, their values continue to be null. A complex constant can be represented by a null value, but the individual real or imaginary part cannot be null.

Input values specified using a repeat specification have the following form:

$r*c$

where *r* is an unsigned non-zero integer constant denoting r repetitions of value *c*. The following results in repetitions of the null value:

$r*$

List-Directed WRITE and PRINT Statements

The ANSI list-directed (free-format) `WRITE` and `PRINT` statements transfer (without editing) data from specified program variables to the next record in an output file.

SYNTAX

```
WRITE ( u, * [, ERR=s] [, END=s] [, IOSTAT=ios] ) [list]

PRINT * [, list]
```

DESCRIPTION

u A unit specifier.

list An output list.

Examples:

```
PRINT, I, J, K
PRINT *, "K= ", K
WRITE (*, *) X, Y, Z
```

Format of List-Directed Output Records

Data	Format
INTEGER *1	I5
INTEGER *2	I5
INTEGER	I10
REAL	1PG14.5
DOUBLE PRECISION	1PG24.15
LOGICAL*1	L1
LOGICAL*2	L1
LOGICAL	L1
COMPLEX	(1X,1P,'(',G13.8,','G13.8,')')
DOUBLE COMPLEX	(1P,(1X,G23.18,1X,',',1X,G23.18')')
CHARACTER *n	An

The formats are explained in Chapter 7.

A printed line is 81 characters containing 80 print positions. The first position or character in every output record is interpreted as carriage control and is not printed. The carriage control character is provided by the compiler for list-directed output statements, and all

output is single spaced. If other carriage control characters are desired, a formatted `WRITE` or `PRINT` statement must be used.

If the current line being printed does not accommodate the values being printed, a new line is started (e.g., three complex numbers cannot be printed on the same line since the total field would be greater than 80 characters).

The output accuracy for real, double precision, complex, and double complex values is such that all full digits of accuracy for the value are output. The least significant portion is not output in order to avoid printing numbers such as 1.99999999 when the value is effectively 2.0000.

Each execution of a list-directed `WRITE` or `PRINT` statement causes the printer carriage to advance to a new line before printing the output.

Numeric data is written right-justified in an output field, whereas character and logical data are printed left-justified in the print field. Character constants are split over two lines if insufficient room is available on a single line. A `T` is printed for a true value, and an `F` is printed for a false value.

Namelist-Directed READ (H)

The namelist-directed READ statement transfers (without editing) and assigns symbolic data from the next record in an input file to the entities that appear in the specified name-list.

SYNTAX

```
READ (u, g [,ERR=s] [,END=s] [,IOSTAT=ios])
```

DESCRIPTION

g Group name that has been defined in a namelist statement.

The namelist read statement reads data from external records accessed under sequential access mode.

If the unit is connected to a file, a namelist read scans forward until it finds the matching group name then begins reading input records.

The read continues until an ampersand or dollar sign (&,$) is detected on input, and everything after the terminator on the current input record is ignored.

The namelist read translates the data from external to internal form using the data types of the entities in the corresponding namelist statement. Namelist reads provide editing as in list-directed reads.

The namelist read assigns the translated data to the specified entity in the order in which they appear in the input records. In the case of arrays, a value or list of values can be assigned beginning at a specified array offset.

Example:

```
NAMELIST /BLK1/TITLE,COLUMNS,LINES,PIXELS,STOP
CHARACTER*80 TITLE
INTEGER COLUMNS,LINES,PIXELS,STOP,IOS
READ(10,NML=BLK1,IOSTAT=IOS)
.
.
.
```

In this example, The namelist statement associates five entities with the group name BLK1. The read statements reads input data and assigns values to the specified namelist entities.

Syntax Rules of Namelist-Directed Input Data Records (H)

Namelist-directed data records consist of an identifying header block, one or more *variable_name*/value combinations, and a closing block.

The header block consists of an ampersand or dollar sign followed by the group name of the namelist block, and must begin in column two.

The header is followed by one or more combinations of variable names and values to be assigned to the variable, in the format:

variable_name `[(`*index_list*`)]` = *list_of_constant_values*

where *variable_name* has been listed in a `NAMELIST` statement in the program unit doing the namelist read. If the specified variable is scalar, only one value may be present. Array subscripts may be specified. If subscript are not present for an array name, the list of values is loaded sequentially into the array. If fewer values are listed than allocated for the array, the remaining array elements are not modified. Using namelist data records to store values beyond the end of an array causes undefined results, and should be avoided. For character variables, substring specifications may be used.

Variable names must be contained on a single data record and not have embedded spaces. Variable names may have embedded underscores (_), however, embedded dollar signs (`$`) are not allowed. The Concurrent Fortran compiler allows these special characters within normal variable names, but the dollar sign is a special delimiter in namelist data files. The equals sign may be preceded or followed by one or more spaces.

The format of the values is the same as in list-directed input records. Assigned values and subscripts (or substring delimiters) must be constant values. The use of symbolic constant names (parameters) is <u>not</u> permitted.

Variable_name/value combinations are separated by a comma and zero or more spaces. Data records are read sequentially and processed until a terminating block is detected.

A terminating block consists of an ampersand or dollar sign optionally followed by the text "`END`". The terminating block must begin in column two of the record.

Data records with non-blank characters in column 1 are assumed to be comments. They may appear anywhere in the namelist data records. As in Fortran program statements, the case of variable names is <u>not</u> significant.

Example:

```
 &BLK1
C
C   Many variables are listed in the NAMELIST block, but
C   often only some of them appear in the input file.
C
    REALVAL=12.456, IARRAY=1,2,3, ARRAY(3)=5.0,
    REAL2=4, INT2=5.0E4
 &END
```

This example should be compared with the NAMELIST statement example in "NAMELIST Statement (H)" on page 4-30. Note the comment characters appearing in column 1, and the implicit type conversion performed for the values of REAL2 and INT2.

Namelist-Directed WRITE (H)

The namelist-directed `WRITE` statement transfers (without editing) data from entities in a specified namelist to the next record in an output file. In the case of arrays, the entire array is output.

SYNTAX

```
WRITE (u, g [,ERR=s] [,END=s] [,IOSTAT=ios])
```

DESCRIPTION

g Group name that has been defined in a namelist statement.

The namelist-directed `WRITE` retrieves data specified in the namelist specifier and translates it from internal to external form by using the data types of the list of entities in the corresponding namelist statements.

The name of the variable and equals sign followed by the value of the variable is written to the output unit in sequential access mode.

The order of values is dictated by the order of variables in the `NAMELIST` statement. Each variable displayed begins on a new line.

Direct Access I/O Statements

A logical record in a direct access external file is a string with the number of bytes specified when the file is opened. Read and write statements must not specify logical records longer than the original record size definition. Shorter logical records are allowed. Unformatted direct access writes leave the empty part of the record undefined. Formatted direct access writes tend to pad the empty record with blanks.

Formatted Direct Access READ

The formatted direct access READ statement transfers data in specified formats from a specific record in a direct access input file to specified program variables. On transfer, data is converted to internal form and is edited based on a format specification.

SYNTAX

```
READ ( u, f, REC=n [, ERR=s] [, IOSTAT=ios] ) [ list ]

READ ( u'n, f [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u Specifies a unit specifier. The unit must have previously been opened for direct access input by an OPEN statement.

f Specifies a format specifier other than an asterisk.

REC=*n*

'n Specifies a record specifier indicating the number of the record to be read. This number must be an existing record number for the direct access file or an error results. The record specifier may not be omitted. The record number must be a positive number in the range of INTEGER values.

list Specifies an input list. If the input list is omitted, the file is positioned at the specified record but no data is read.

If a record contains more data than the formatted READ statement needs, additional data is ignored. Each READ causes a new record to be read. An error results and all entities in the input list become undefined if the input list and format specification require more data than is available on the input data record.

An attempt to read a record outside the file's boundaries results in an end-of-file condition with an I/O status of -1. If the file is being read in a sequential fashion (e.g., in a loop in which the record number is being incremented), the READ should be followed by a check for a negative I/O status.

Examples:

```
READ (REC=5, UNIT=7, FMT=100, ERR=99) A, B, C
READ (7, 100, REC=1) I, J, K
READ (*, REC=25, FMT=ARRAY)
READ (7, REC=RECPOS, FMT=1099) Z
```

Formatted Direct Access WRITE

The formatted direct access `WRITE` statement transfers data in specified formats from specified program variables to a specific record in a direct access output file. On transfer, data is converted to ASCII characters and is edited based on a format specification.

SYNTAX

```
WRITE ( u, f, REC=n [, ERR=s] [, IOSTAT=ios] ) [ list ]

WRITE ( u'n, f [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u — A unit specifier. The unit must have previously been opened for direct access output by an `OPEN` statement.

f — Specifies a format specifier other than an asterisk.

`REC=`*n*

'n — A record specifier indicating the number of the next record to be written. Only the one record can be written, and the output list and format must not cause additional records to be written.

All records that are written must not exceed the maximum record length defined for the direct access file in an `OPEN` statement, if a maximum record length was specified. If insufficient data is written to fill a record, blanks are automatically appended to fill the record's length. An error results if more data is written than can be contained in the record. The `REC` specifier cannot be omitted. The record number must be a number in the range of `INTEGER` values.

list — Specifies an output list. If the output list is omitted, output data is taken from the format specification or, if none exists, a blank record is written.

An attempt to write a record beyond the file's boundaries results in an error.

Examples:

```
WRITE (6, 100, REC=10) A, B, C
WRITE (FMT=555, UNIT=6, ERR=99, REC=15) IX, IY, IZ
WRITE ( 11, '(A)', REC=14) LINE
WRITE (*, 1000, REC=50)
WRITE (7, REC=RECPOS, FMT=1099) Z
```

Unformatted Direct Access READ

The unformatted direct access READ statement transfers (without editing) binary data from a specified record in a direct access input file to specified program variables. When transferred, the data is not edited or translated in any way.

SYNTAX

```
READ ( u, REC=n [, ERR=s] [, IOSTAT=ios] ) [ list ]

READ ( u'n [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u — A unit specifier. The unit must have previously been opened for direct access input by an OPEN statement.

REC=*n*
'n — A record specifier indicating the number of the record to be read. This number must be an existing record number for the direct access file or an error results. The record number must be a positive integer in the range of INTEGER values. The REC specifier cannot be omitted.

list — Specifies an input list. If the input list is omitted, the file is positioned at the specified record but no data is read.

If a record contains more binary data than the unformatted READ statement needs, additional data is ignored. Each READ statement causes a new record to be read. An error results and all entities in the input list become undefined if the input list requires more binary data than is available in the input data record.

An attempt to read a record beyond the file's boundaries results in an end-of-file condition with an I/O status of -1.

Examples:

```
READ (7, ERR=99, REC=I) A, B, C
READ (ERR=99, UNIT=10, REC=58) K, L, M
READ (UNIT=7, REC=40)
READ (*, REC=1) J, JJ, JJJ
```

Unformatted Direct Access WRITE

The unformatted direct access `WRITE` statement transfers (without editing) binary data from specified program variables to a specified record in a direct access output file. No data translation or editing is performed when the data is written.

SYNTAX

```
WRITE ( u, REC=n [, ERR=s] [, IOSTAT=ios] ) [ list ]

WRITE ( u'n [, ERR=s] [, IOSTAT=ios] ) [ list ]
```

DESCRIPTION

u — A unit specifier. The unit must have previously been opened for direct access output with an `OPEN` statement.

`REC=`*n*
'n — A record specifier indicating the number of the next record to be written. Each unformatted `WRITE` writes exactly one record. The record number must be a number in the range of `INTEGER` values.

All records that are written must not exceed the maximum record length defined for the direct access file in an `OPEN` statement. If insufficient data is written to fill a record, the remainder of the record is undefined. The `REC` specifier cannot be omitted.

list — Specifies an output list. If the output list is omitted, a null record is written.

An attempt to write a record outside the file's boundaries results in an error.

Examples:

```
WRITE (6, REC=15) A, B, C
WRITE (REC=1, UNIT=6, ERR=99) JJ, KK, LL
WRITE (*, REC=J)
```

OPEN Statement

An OPEN statement is used to connect a unit, create and assign a file and connect it to a unit, or change certain specifiers of a connection between a file and a unit. An OPEN statement must specify a unit and can optionally specify a file name that corresponds to a system file name.

If a sequential READ or WRITE statement is executed and an OPEN statement for the file has not been executed, the file is implicitly opened. An explicit OPEN statement must be executed before data can be written to or read from a direct access file.

When a file is opened, the following file properties are established explicitly or implicitly:

- An access method (sequential or direct access).
- The form of the file (formatted, unformatted, or print).
- A record length (for direct access files).
- A blank significance specifier for numeric data (applies only to formatted input files).
- A file status (indicating whether the file is old, new, or a scratch file).

SYNTAX

```
OPEN ( [UNIT=]u [ specifiers ] )
```

where *specifiers* are as follows:

```
ACCESS=acc
ASSOCIATEVARIABLE=asv
BLANK=blnk
BLOCKSIZE=bsize
BUFFERCOUNT=bcount
CARRIAGECONTROL=cc
DEFAULTFILE=name
DISP=disp
DISPOSE=disp
ERR=s
EXTENDSIZE=esize
FILE=name
FORM=fm
INITIALSIZE=isize
IOSTAT=ios
MAXREC=max
NAME=name
NOSPANBLOCKS
ORGANIZATION=org
READONLY
RECORDSIZE=len
RECORDTYPE=rtype
RECL=len
SHARED
```

STATUS=*sta*
TYPE=*sta*
USEROPEN=*proc*

DESCRIPTION

All *specifiers* are optional except the unit specifier. The RECL specifier must be present if a file is being connected for direct access.

In the following specifier descriptions, a *character entity* can be a character constant enclosed in single or double quotes, a character variable, a character array element, or a character expression. The data type requirements and meaning of each specifier is provided in Table 6-3.

Table 6-3. OPEN Statement Specifiers, Data Types, and Meaning

Specifier	Data Type	Meaning
u	INTEGER	Identifies a required unit number and must be the first specifier in the list unless the UNIT keyword is used. Once the unit is connected, it can be referenced in any program unit of the source program.
ACCESS=*acc*	CHARACTER	Identifies the access method of the file connected to unit *u*. The *acc* must be a character entity whose value is "SEQUENTIAL", "DIRECT", or "APPEND". The default access method is sequential. For an existing file, the specified access method must be a permissible access method for the file.
ASSOCIATEVARIABLE=*asv*	INTEGER	Specifies an integer variable which is updated after each I/O call to the file to reflect the record number of the next record in the file. The variable must not be a dummy argument. This specifier may be used only in an OPEN for direct access; otherwise, it is ignored.
BLANK=*blnk*	CHARACTER	Indicates how blank characters in formatted numeric input fields are to be handled. This specifier applies only to formatted input files. The *blnk* must be a character entity whose value is either "NULL" or "ZERO". If "NULL" is specified, all blanks in a formatted numeric input field are ignored, and a field of all blanks has a value of zero. If "ZERO" is specified, all blanks except leading blanks are treated as zeroes. The default setting is "NULL".
BLOCKSIZE=*bsize*		This specifier has no effect. It is included to facilitate compatibility with other compilers and is accepted but ignored and a warning is issued.
BUFFERCOUNT=*bcount*		This specifier has no effect. It is included to facilitate compatibility with other compilers and is accepted but ignored and a warning is issued.
CARRIAGECONTROL=*cc*	CHARACTER	Indicates how carriage control is to be handled. The *cc* must be a character entity whose value is "Fortran", "LIST", or "NONE". The default setting is "Fortran".

Table 6-3. OPEN Statement Specifiers, Data Types, and Meaning (Cont.)

Specifier	Data Type	Meaning
DEFAULTFILE=*name*	CHARACTER	*name* must be a character entity. This specifier has effect only when a FILE or NAME parameter is not supplied. The meaning of *name* is the same as for the FILE parameter in this case.
DISPOSE=*disp*	CHARACTER	Indicates what will happen to the file when it is closed. The *disp* must be a character entity whose value is "KEEP", "SAVE", "DELETE", "PRINT", or "PRINT/DELETE". If "KEEP" or "SAVE" is specified, the file remains after it is closed. If "DELETE" is specified, the file is removed when closed. If "PRINT" is specified, the file is submitted to the system line printer and is not deleted unless "PRINT/DELETE" is specified. The keyword DISP may be used in place of DISPOSE. A READONLY file may not be deleted. A scratch file may not be saved or printed.
ERR=*s*	Statement label	An error specifier (see "ERR Specifier" on page 6-9).
EXTENDSIZE=*esize*	INTEGER	This specifier has no effect. It is included to facilitate compatibility with other compilers and is accepted but ignored and a warning is issued.
FILE=*name*	CHARACTER	Identifies the name of the file connected to the unit. If the specified name represents an environment variable, the value of that variable is used as the name of the file connected to the unit. Otherwise, the name itself is used as the file name. The keyword NAME may be used in place of FILE.
FORM=*fm*	CHARACTER	Indicates whether the file is being connected for formatted or unformatted I/O, or vertical format print control. The *fm* is a character entity whose value is either "FORMATTED", "UNFORMATTED" or "PRINT". If the FORM specifier is omitted for a new file, the compiler assumes "UNFORMATTED" for files connected for direct access and assumes "FORMATTED" if the file is connected for sequential access. If vertical format control is desired, the unit must be open for sequential access with the value "PRINT" (see "Vertical Format Control" on page 6-4).
INITIALSIZE=*isize*	INTEGER	This specifier has no effect. It is included to facilitate compatibility with other compilers and is accepted but ignored and a warning is issued.
IOSTAT=*ios*	INTEGER	An input/output status specifier (see "IOSTAT Specifier" on page 6-11).

Table 6-3. OPEN Statement Specifiers, Data Types, and Meaning (Cont.)

Specifier	Data Type	Meaning
MAXREC=*max*	INTEGER	Indicates the maximum number of records permitted in a direct access file. The *max* must be an integer expression. This specifier may be used only in a file opened for direct access.
NAME=*name*	CHARACTER	This specifier is a non-standard synonym for FILE.
NOSPANBLOCKS		This specifier is has no effect. It is included to facilitate compatibility with other compilers.
ORGANIZATION=*org*		This specifier has no effect. It is included to facilitate compatibility with other compilers.
READONLY		Indicates that a file may be used only for reads, regardless of whether the user has write access to the file.
RECORDTYPE=*type*		This specifier has no effect. It is included to facilitate compatibility with other compilers.
RECL=*len*	INTEGER	Identifies the length of each record of a file connected for direct access. The *len* must be a positive integer constant or integer expression. This specifier must be present for files connected for direct access. The length is defined in bytes (characters) for both formatted and unformatted I/O. For an existing file, *len* must be the record length defined for the file. For a new file, the user must specify the record length or an error results. The keyword RECORDSIZE may be used in place of RECL.
SHARED		This specifier has no effect. It is included to facilitate compatibility with other compilers.

Table 6-3. OPEN Statement Specifiers, Data Types, and Meaning (Cont.)

Specifier	Data Type	Meaning
STATUS=*sta*	CHARACTER	Indicates the status of the file on opening. The *sta* is a character entity whose value is either "NEW", "OLD", "SCRATCH", or "UNKNOWN". If "OLD" is specified, the Fortran I/O system assumes the file exists. If "NEW" is specified, the Fortran I/O system assumes the file does not exist and creates it. If "UNKNOWN" is specified, the Fortran I/O system simply opens an existing file or creates the file if it does not exist. If "SCRATCH" is specified, the Fortran I/O system assigns the unit number to a temporary file with a name of the form **tmp.F***xxxxxx*, where *xxxxxx* is a process id number and opens the file. When the unit is closed or the program terminates, the scratch file is deleted. "SCRATCH" must not be specified with a named file. The FILE specifier must be present if "NEW" or "OLD" is specified explicitly. The default status is "UNKNOWN". Any other status specifier without an associated file name opens a file named **fort.***N*, where *N* is the specified unit number.
TYPE=*sta*	CHARACTER	The keyword TYPE may be used in place of STATUS.
USEROPEN=*proc*		This specifier has no effect. It is included to facilitate compatibility with other compilers.

If a unit is connected to a file, an OPEN statement specifying a different unit for the same file is not allowed until the previous file is disconnected from the unit.

If the unit being connected to the file is different from a file already connected to the unit, the new file is connected and the old file is disconnected.

The same specifier or unit must not be specified more than once in the same OPEN statement.

By default, a file is positioned at the beginning upon opening. An external file opened for sequential access is positioned at the end if it is opened with APPEND access. Existing files are never truncated on opening.

CLOSE Statement

A CLOSE statement disconnects a file from a unit.

SYNTAX

```
CLOSE u

CLOSE ([UNIT=]u [, ERR=s] [, IOSTAT=ios] [, STATUS=sta])
```

All specifiers are optional except the unit specifier. No specifier can be used more than once in a single CLOSE statement.

DESCRIPTION

u	The required unit specifier. The unit specifier identifies the external unit and must be the first specifier in the list unless the UNIT keyword is used. A unit is disconnected whenever a CLOSE statement specifying that unit is executed.
ERR=*s*	An error specifier (see "ERR Specifier" on page 6-9).
IOSTAT=*ios*	An input/output status specifier (see "IOSTAT Specifier" on page 6-11).
STATUS=*sta*	Determines the disposition of the file on closing. The status is a character entity whose value is "KEEP", "SAVE", "DELETE", "PRINT", or "PRINT/DELETE". "SAVE" is equivalent to "KEEP". If "DELETE" is specified, the file is eliminated after closing. If "KEEP" is specified, the file is not deleted. If "KEEP" is specified and the file does not exist, the file will not exist after closing. "KEEP" may be specified for temporary files; if you want to open it again later, get the scratch file's real name using INQUIRE. If "PRINT" is specified, the file is submitted to the system line printer and is not deleted unless "PRINT/DELETE" is specified. The default is "DELETE" for scratch files and "KEEP" for all other files. The keywords DISP and DISPOSE may be used in place of STATUS.

A CLOSE statement for a unit need not appear in the same program unit in which an OPEN statement for the unit was specified.

Execution of a CLOSE statement on a unit or file that does not exist has no effect.

Sequentially accessed external files are truncated to the current file position during a CLOSE if the last access to the file was a WRITE.

INQUIRE Statement

The INQUIRE statement is used to inquire, either by file name or by unit number, about the characteristics of an external file. An INQUIRE statement may be executed before, while, or after a file is connected to a unit. If an inquiry by file name is performed, the file, but not the unit number, is specified. If an inquiry by unit is performed, the unit number, but not the file name, is specified.

SYNTAX

Name statement:

```
INQUIRE ( FILE=fname [ specifiers ] )
```

Unit statement:

```
INQUIRE ( UNIT=u [ specifiers ] )
```

DESCRIPTION

The name is specified in the same manner as it is in the OPEN statement (see "OPEN Statement" on page 6-36).

The *specifiers* are as follows:

```
ACCESS=acc
BLANK=blnk
CARRIAGECONTROL=cc
DEFAULTFILE=fn
DIRECT=dir
ERR=s
EXIST=ex
FORM=fm
FORMATTED=fmt
IOSTAT=ios
NAME=fn
NAMED=nmd
NEXTREC=nr
NUMBER=num
OPENED=od
RECL=len
RECORDTYPE=rtype
SEQUENTIAL=seq
UNFORMATTED=unf
```

Specifiers are separated by commas, and no specifier is used more than once in a single INQUIRE statement. More than one INQUIRE statement can be present in a single program unit. The data type requirements and meaning of each specifier is provided in Table 6-4. Each specifier is a variable or subscripted array of the indicated data type.

Table 6-4. INQUIRE Statement Specifiers, Data Types and Meaning

Specifier	Data Type	Meaning
ACCESS=*acc*	CHARACTER	*acc* has the value "SEQUENTIAL" if the file is connected for sequential access or "DIRECT" if connected for direct access. *acc* becomes undefined if the file is not connected.
BLANK=*blnk*	CHARACTER	*blnk* has the value "NULL" if null blank control is in effect or "ZERO" if zero blank control is in effect for a connected file. *blnk* is undefined if the file is not connected, or if the connection is not for formatted I/O.
CARRIAGECONTROL=*cc*	CHARACTER	*cc* has the value "Fortran", "LIST", or "NONE", depending on how the file was opened. *cc* is undefined if the file is not connected, or if the connection is not for formatted I/O.
DEFAULTFILE=*name*	CHARACTER	*name* is the name supplied to the DEFAULTFILE parameter when the file was opened.
DIRECT=*dir*	CHARACTER	*dir* has the value "YES" if the file can be direct access, "NO" if the file cannot be direct access, or "UNKNOWN" if the access is indeterminate.
ERR=*s*	Statement label	Transfers control to statement number *s* if an error on the file or unit exists. (See "ERR Specifier" on page 6-9.)
EXIST=*ex*	LOGICAL	*ex* contains .TRUE. if the file exists or .FALSE. otherwise.
FORM=*fm*	CHARACTER	*fm* has the value "FORMATTED" if the file is connected for formatted I/O, "UNFORMATTED" if the file is connected for unformatted I/O, and "PRINT" if the file was opened with vertical format control. *fm* is undefined if no connection exists.
FORMATTED=*fmt*	CHARACTER	*fmt* has the value "YES" if the file can be a formatted file, "NO" if the file cannot be a formatted file, or "UNKNOWN" if *fmt* is indeterminate.
IOSTAT=*ios*	INTEGER	*ios* is zero if an error or end-of-file does not exists, or a non-zero value if an error or end-of-file exists (see "IOSTAT Specifier" on page 6-11).
NAME=*fn*	CHARACTER	*fn* is the external file name for an INQUIRE by unit.
NAMED=*nmd*	LOGICAL	*nmd* contains .TRUE. if the file has a name or .FALSE. otherwise.
NEXTREC=*nr*	INTEGER	If *n* is the current record number for a direct access file, *nr* is *n* + 1. If records have not been read or written, *nr* is 1. *nr* is undefined if the file is not connected, or if the connection is not for direct access.
NUMBER=*num*	INTEGER	*num* contains the value of the external unit identifier of the currently connected unit. *num* is undefined if a unit is not connected.

Table 6-4. INQUIRE Statement Specifiers, Data Types and Meaning (Cont.)

Specifier	Data Type	Meaning
OPENED=*od*	LOGICAL	*od* contains .TRUE. if the file name is connected to some unit, or if the specified unit is connected to some file, or.FALSE. otherwise.
RECL=*len*	INTEGER	*len* contains the fixed length in bytes of records in the file. *len* is undefined if a connection does not exist or if the file is connected for sequential access.
RECORDTYPE=*rtype*	CHARACTER	*rtype* always has the value "STREAM_LF". It is included to facilitate compatibility with other compilers.
SEQUENTIAL=*seq*	CHARACTER	*seq* has the value "YES" if the file can be accessed sequentially, "NO" if the file cannot be accessed sequentially, or "UNKNOWN" if the access is indeterminate.
UNFORMATTED=*unf*	CHARACTER	*unf* has the value "YES" if the file can be an unformatted file, "NO" if the file cannot be unformatted, or "UNKNOWN" if the form is indeterminate.

All specifier values become undefined if an error occurs during the execution of INQUIRE. The EXIST and OPENED specifiers are always defined unless an error occurs.

A variable or array element that becomes undefined as a result of its use as an INQUIRE specifier must not be used in more than one specifier in the same INQUIRE statement.

The use of an INTEGER or LOGICAL variable or array element that is not of four-byte size within an INQUIRE specifier that may modify it, causes the compiler to generate an assignment prior to completion of the INQUIRE statement.

The following rules apply to INQUIRE by file name statements:

- The following specifiers are assigned values only if the value of the FILE specifier is an acceptable file name or if the file exists by that name: DIRECT, CARRIAGECONTROL, FORMATTED, NAME, NAMED, SEQUENTIAL, and UNFORMATTED.
- The NUMBER specifier becomes defined only if the OPENED parameter has a true value. Also, the specifier values for ACCESS, BLANK, FORM, NEXTREC, RECL, and RECORDTYPE become defined only if the value of OPENED is true.

The following rule applies to INQUIRE by unit statements:

The following specifiers become defined only if the specified unit exists and if a file is connected: ACCESS, BLANK, CARRIAGECONTROL, DIRECT, FORM, FORMATTED, NAME, NAMED, NEXTREC, NUMBER, RECL, RECORDTYPE, and SEQUENTIAL. Otherwise, they become undefined.

Example:

```
CHARACTER *15 ACC, DIR, FM, FMATED, SQ, UFMATED
LOGICAL EXST, NMED
PRINT, "THIS IS A TEST"
INQUIRE (UNIT=6, ACCESS=ACC)
INQUIRE (UNIT=6, DIRECT=DIR)
INQUIRE (UNIT=6, EXIST=EXST)
INQUIRE (UNIT=6, FORM=FM)
INQUIRE (UNIT=6, FORMATTED=FMATED)
INQUIRE (UNIT=6, NAMED=NMED)
INQUIRE (UNIT=6, SEQUENTIAL=SQ)
INQUIRE (UNIT=6, UNFORMATTED=UFMATED)
PRINT, "ACCESS = ", ACC
PRINT, "DIRECT = ", DIR
PRINT, "EXIST = ", EXST
PRINT, "FORM = ", FM
PRINT, "FORMATTED = ", FMATED
PRINT, "NAMED = ", NMED
PRINT, "SEQUENTIAL = ", SQ
PRINT, "UNFORMATTED = ", UFMATED
END
```

FLUSH Subroutine (H)

The C library `fflush` function transfers buffered data to an output file. It can be invoked on Fortran 77 logical units, through a subroutine interface.

SYNTAX

```
CALL FLUSH [(n)]
```

DESCRIPTION

n A unit specifier. If omitted, `FLUSH` flushes all logical units.

BACKSPACE Statement

A BACKSPACE statement positions a sequential access external file at the previous sequential record.

SYNTAX

```
BACKSPACE u

BACKSPACE ([UNIT=]u [, IOSTAT=ios] [, ERR=s] )
```

DESCRIPTION

u Is the required unit specifier.

If a preceding record does not exist in the file, the position of the file does not change, and the IOSTAT variable, if any, is set to 0.

BACKSPACE cannot be used to backspace a direct access file, a connected sequential file that does not exist, or a file created using list-directed formatting.

Sequentially accessed external files are truncated to the current file position during a BACKSPACE if the last access to the file was a WRITE.

ENDFILE Statement

An ENDFILE statement truncates a sequential access external file to the current file position.

SYNTAX

```
ENDFILE u

ENDFILE ([UNIT=]u [, IOSTAT=ios] [, ERR=s] )
```

DESCRIPTION

u Is the required unit specifier.

Execution of an ENDFILE statement for a connected file that does not exist creates the file.

The Fortran standard does not permit I/O after an ENDFILE operation unless it is preceded by a BACKSPACE or a REWIND operation. The Fortran I/O library permits write operations to be performed after an ENDFILE. The ENDFILE truncates the file, but there is no physical embodiment of an end-of-file record.

REWIND Statement

A REWIND statement repositions a sequential access external file at the initial point of the file.

SYNTAX

```
REWIND u

REWIND ([UNIT=]u [, IOSTAT=ios] [, ERR=s] )
```

DESCRIPTION

u Is the required unit specifier.

Execution of REWIND for a file that is already at the initial point or for a connected file that does not exist has no effect.

Sequentially accessed external files are truncated to the current file position during a REWIND if the last access to the file was a WRITE.

7
Formatted Input and Output

7
Formatted Input and Output

Format Specification

If formatted input and output statements are used, all formatting is under the control of the programmer. Formatted input and output statements contain a format specifier that is a reference to a `FORMAT` statement or that is a Hollerith array or character entity defining a format specification.

A *format specification* defines the size of input and output fields, what type of data is being read or written (numeric, character, Hollerith, or logical) and how the data is to be edited.

SYNTAX

`(`[*fs*]`)`

DESCRIPTION

fs Specifies a list of editing descriptors separated by commas. The parentheses must be specified even if the format specification is empty. The format specification is empty only if the input or output statement referencing the format specification does not contain an input or output list.

The format descriptors consist of the following categories:

- Integer editing descriptor (`I`)
- Octal editing descriptor (`O`)
- Hexadecimal editing descriptor (`Z`)
- Real, double precision, complex and double complex editing descriptors (`F`, `E`, `D`, and `G`)
- Character editing descriptors (`A`, the apostrophe, and the double quote)
- Hollerith editing descriptors (`A` and `H`)
- Logical editing descriptor (`L`)
- Positional editing descriptors (`X`, `T`, `TL`, and `TR`)
- Format control descriptors (`/`, `:`, `R`, `S`, `SS`, `SP`, `SU`, `B`, `BN`, and `BZ`)
- Miscellaneous non-standard descriptors (`Q`,`$`)

On input, editing descriptors specify the size of fields to be read from an input data record. Each numeric, character, Hollerith, or logical field is matched with an entity in the input list of the READ statement on a left to right basis.

On output, editing descriptors define output fields for an output record, and each entity in the output list of the WRITE or PRINT statement is matched on a left to right basis with the editing descriptor defining the output field.

The commas separating editing descriptors are omitted before or after a slash descriptor, before or after a colon descriptor, and before or after a scaling factor.

I/O formatting operates in the following manner:

- When a formatted input statement is executed, a new input data record is read. When an output statement is executed, construction of a new output data record is begun.
- Attempting to read or write more characters on a record than are (or can be) physically present does not cause a new record to be begun. On input, extra characters are ignored. On output, characters are lost.
- During input and output operations, a record is terminated whenever a slash descriptor or the closing right parenthesis of the format specification is sensed, or when the format specification requests an item from the input or output list, and there are not any remaining items in the list.
- Each list item is matched to one descriptor or to one repeated descriptor except for a complex list item, which is matched with two such descriptors.
- Each formatted input or output statement containing an I/O list must refer to a format specification that contains at least one numeric, logical, or character editing descriptor. If this condition is not met, the format specification is processed, but an error results.
- Whenever a numeric, logical, or character editing descriptor is encountered, a new input or output list item is processed, data is converted as necessary, and the next item in the format specification is processed. If conversion is impossible because of a conflict between an editing descriptor and a data type, an error results. If the next descriptor is a format control or format positioning descriptor, the descriptor is processed whether or not items remain in the input or output list.

  Example:

```
      WRITE (6, 100)
100 FORMAT ( /, /, /, 'ABCD' )
```

  Produces three blank records and one record containing ABCD before reaching the closing right parenthesis of the format specification. When there are no more items remaining in an input or output list, and the closing right parenthesis is encountered or a conversion descriptor (i.e., A, D, E, F, G, I, O, or Z) has been found, the current record is terminated and the I/O operation is complete.

- When the closing right parenthesis of the format specification is encountered, a test is made to determine if all items in the input or output list have been processed. If no list items remain, the current record is

terminated, and the I/O operation is complete. If list items remain, a new record is processed, and the format specification is rescanned as follows:

- If group specifications are not present, the entire format specification is rescanned.
- If group specifications are present, rescanning begins with the group specification whose right parenthesis was the last one encountered before the closing right parenthesis of the format specification.

Group Specification

A *group specification* is a list of editing descriptors that are enclosed within parentheses in the format specification. Editing descriptors in the group specification are separated by commas.

Zero or more group specifications can be specified in a format specification, and group specifications can contain other group specifications. Group specifications can be nested to a depth of ten levels. Group specifications can be intermixed with single editing descriptors in the same format specification.

An editing descriptor that normally cannot be repeated can be enclosed in parentheses in a group specification and be repeated by preceding the group specification with a repetition factor.

Example:

```
READ ( UNIT=11, FMT='(I5, A3, 2(F10.3, I4))' )I,J,A,K,B,M
WRITE ( 6, '(I5, /, A3, /, (F15.3, /, I4))' ) I,J,A,K,B,M
```

Input Record:

```
1234567890b1234567890b1234567890b1234567890b1234567890
```

Output:

```
12345
678
     9001234.563
 890
   1234567.875
 123
```

Repetition Factor

An optional repetition factor of the following form precedes certain editing descriptors or precedes a group specification:

SYNTAX

rd

r (*gs*)

DESCRIPTION

r	Specifies an unsigned integer constant between 1 and 32767 denoting a repetition factor.
d	Specifies an editing descriptor for which a repetition factor is permitted.
gs	Is a group specification.

If a repetition factor is not specified, an editing descriptor or group specification is used once. The editing descriptors for which a repetition factor is valid are given in Table 7-1.

Examples:

```
READ (11, '(3I5)') I, II, III
WRITE (6,'(3(2X,I5),/,"  SUM= ",I10)')I,II,III,I+II+III
```

Input Record:

```
1234567890b1234567890b1234567890b1234567890b1234567890
```

Output:

```
12345bb67890bb1234
SUM=      81469
```

Scaling Factor

An optional scaling factor of the following form can precede the F, E, D, or G editing descriptors:

SYNTAX

[*c*]P[,]*d*

[*c*]P[,]*rd*

DESCRIPTION

c — Specifies an optionally signed integer indicating the number of places the decimal point is to be scaled to the left or right. If *c* is omitted, then the scale factor is set to the default value of zero.

P — Denotes that a scaling factor is present.

r — Specifies an optional repetition factor.

d — Specifies one of the editing descriptors F, E, D, or G. An optional comma can follow P.

When a scaling factor is encountered in a format specification, it remains in effect for all subsequent F, E, D, and G field descriptors.

The scaling factor is treated as a multiplier of the form 10**-*c* for input and 10***c* for output. On output, the decimal point is shifted to the right *c* places. For the D and E descriptors, the exponent field is reduced by *c* for output. Thus, for output under D, E, and G control, the number is equal to the rounded internal value; for output under F control, the number is not equal to the internal value unless *c* is zero.

The scaling factor is set to zero at the beginning of a formatted I/O operation.

The scaling factor can be changed in a format specification any number of times. Each new scaling factor is in effect for the remainder of the format specification unless another scaling factor is defined. If an entire format specification or a portion of a format specification is rescanned to satisfy all items in the input or output list, the scaling factor in effect remains in effect for the rescanning.

FORMAT Statement

A FORMAT statement is a non-executable statement appearing anywhere in a program unit. FORMAT statements define a format specification for use by a formatted input or output statement.

SYNTAX

label FORMAT ([*fs*])

DESCRIPTION

label Specifies a unique statement number.

fs Denotes a format specification.

A statement label must be present so the format can be referenced. FORMAT statements are the only statements that require a label. A formatted READ, WRITE, or PRINT statement refers only to a FORMAT statement that appears in the same program unit.

Examples:

```
      DOUBLE PRECISION   DA
      . . .
      READ ( 11, 104 ) IA, RA, DA
104   FORMAT ( I3, F15.6, F20.11 )
      WRITE ( 6, 105 ) IA, RA, DA
105   FORMAT ( I3 / F20.7 / F25.12 )
```

Input Record:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

Output:

```
123
   456789024.0000000
   901234567.890123360000
```

Character Format Specifications

The format specifier of a formatted I/O statement is a character expression, variable, or constant; a Hollerith constant; or a subscripted array name or unsubscripted array name containing Hollerith data (refer to “Format Specifier” on page 6-10). A format specification is derived from a character expression or read as input data. The format specification must be stored in the proper form in the character or Hollerith entity; i.e., the

beginning left parenthesis, closing right parenthesis, and list of editing descriptors (and group specifications if any), separated by commas, must be present.

Example:

```
DOUBLE PRECISION DA
INTEGER HFS(5)
CHARACTER *21 CFS
. . .
DATA HFS /19H(I3, F15.6, F20.11)/
DATA CFS /'(I3 / F20.7 / F25.12)'/
. . .
READ ( UNIT=11, FMT=HFS ) IA, RA, DA
WRITE ( UNIT=6, FMT=CFS ) IA, RA, DA
```

Input Record:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

Output:

```
123
   456789012.3400000
   901234567.890000000000
```

Editing Descriptors

Editing descriptors are summarized in Table 7-1. Lowercase letters in the table have the following meanings:

a	Denotes an ASCII character.
*c*P	Denotes a scaling factor.
d	Denotes the number of digits in the field that are to be placed to the right of the internal decimal point. *d* must be an unsigned integer between 0 and 32767, inclusive.
e	Denotes an exponent field. *e* must be an unsigned integer in the range 1 to 127, inclusive.
h	Denotes a Hollerith character.
m	Denotes the exact number of digits in a field to be written to an output field, including leading zeroes (0 <=*m* <= *w*).
r	Denotes a repetition factor. *r* must be an unsigned integer between 1 and 32767, inclusive.

w Denotes the total width of a numeric, character, or logical field. *w* must be an unsigned integer between 1 and 32767, inclusive.

Table 7-1. Summary of Editing Descriptors

Descriptor Form	Function	Repetition Factor	Scaling Factor
'a1...aw'	The apostrophe defines a character input/output field.	--	--
/	On input, causes a new record to be read. On output, causes the current record to be written.	--	--
:	Terminate output if no remaining output list items exist.	--	--
$	Suppress carriage return on output records.	--	--
A*w*	Define a character or Hollerith field. The *w* can be omitted for character data on both input and output. The *w* must be present for the input and output of Hollerith data.	*r*A*w*	--
B	Return to default mode of blank interpretation.	--	--
BN	Ignore blank characters in a numeric input field.	--	--
BZ	Treat blank characters in a numeric input field as zeroes.	--	--
D*w.d*	Define a real or double precision field.	*r*D*w.d*	*c*PD*w.d*
E*w.d* or E*w.d*E*e* or E*w.d*D*e* or E*w.d.e*	Define a real or double precision field	*r*E*w.d* *r*E*w.d*E*e* *r*E*w.d*D*e* *r*E*w.d.e*	*c*PE*w.d* *c*PE*w.d*E*e* *c*PE*w.d*D*e* *c*PE*w.d*E*e*
F*w.d*	Define a real or double precision field.	*r*F*w.d*	*c*PF*w.d*
G*w.d* or G*w.d*E*e*	Define a real or double precision field.	*r*G*w.d* *r*G*w.d*E*e*	*c*PG*w.d* *c*PG*w.d*E*e*
*w*H*h1...hw*	Define a Hollerith field for output.	--	--
I*w* or I*w.m*	Define an integer field.	*r*I*w* or *r*I*w.m*	--
L*w*	Define a logical field.	*r*L*w*	--
O*w* or O*w.m*	Define an octal field.	*r*O*w* or *r*O*w.m*	--
Q	Obtain the number of characters remaining in an input record.	--	--
*n*R	Set the radix for integers.	--	--
S or SS	Do not insert a leading plus sign on output.	--	--
SP	Insert a leading plus sign on output.	--	--
SU	Interpret integers as unsigned.	--	--
*n*T	Move to the *n*th eight-column location.	--	--
T*n*	Move current position pointer to position *n* in a data record	--	--
TL*n*	Move the current position pointer backward *n* positions.	--	--

Table 7-1. Summary of Editing Descriptors (Cont.)

Descriptor Form	Function	Repetition Factor	Scaling Factor
`TR`*n*	Move the current position pointer forward *n* positions.	--	--
w`X`	On input, skip a field of length *w*. On output, write a field of length *w* filled with blanks.	--	--
Z*w* or Z*w.m*	Define a hexadecimal field	*r*Z*w* or *r*Z*w.m*	--
`(...)`	Denote a group specification.	*r* `(...)`	--

Apostrophe (' ')

The apostrophe is a character editing descriptor that can be used in an output format specification. The apostrophe descriptor takes the form of a character constant enclosed in quotes and is used to define a character string to be read/written. On input, characters in string *s* are replaced by the same number of characters from the input record (see Example 2). The apostrophe descriptor is not permitted in an input specification.

SYNTAX

'*a1...aw*'

DESCRIPTION

a Denotes an ASCII character. The width of the output field is the number of characters in the character string excluding the opening and closing apostrophes. An apostrophe in the string is represented as two adjacent apostrophes with no intervening blanks and occupies one character position in the output field.

Characters including blanks in the character constant are output directly from the format specification, and the character constant does not correspond to an item in the output list.

Example:

```
DOUBLE PRECISION    DSUM
DSUM = 4596565605D-5 + 5765473243D-2
PRINT "(' ', 'THE ANSWER IS: ', F20.10)", DSUM
```

Output:

```
THE ANSWER IS: 57700698.0890499960
```

Double Quote (" ")

The double quote is a character editing descriptor that is functionally equivalent to the apostrophe descriptor.

Example:

```
DOUBLE PRECISION DSUM
DSUM = 4596565605D-5 + 5765473243D-2
PRINT '(" ", "THE ANSWER IS: ", F20.10)', DSUM
```

Output:

```
THE ANSWER IS: 57700698.0860499960
```

Slash (/)

The slash (/) is a control descriptor that can be used in either an input or output format specification. The slash descriptor ends data transfer on the current record and results in the positioning of the input or output data file at a new record.

SYNTAX

```
/
```

DESCRIPTION

For files connected for sequential access, a slash causes:

- A new record to be read for input (beginning with the first character position of the new input record), or
- The current record to be written and a new record begun for output (beginning with the first character position of the new output record).

The occurrence of *n* adjacent slashes in a format specification causes *n*-1 blank records to be written on output or *n*-1 records to be skipped on input. For direct access files, a slash ends data transfer on the current record and increases the record count by one.

Example:

```
INTEGER    HARRAY(7)
. . .
READ ( UNIT=11, FMT='(7(A4, /))') HARRAY
WRITE ( UNIT=6, FMT="(7(' ' ,A4))" ) HARRAY
```

Input Records:

```
ABCD
EFGH
IJKL
MNOP
QRST
UVWX
YZb
```

Output:

```
bABCDbEFGHbIJKLbMNOPbQRSTbUVWXbYZbb
```

Colon (:)

The colon is a control descriptor that can be used in an output format specification. The colon editing descriptor terminates an output operation if there are no more remaining items in the output list.

SYNTAX

:

DESCRIPTION

A colon in an input format specification is ignored.

Example:

```
DATA I, J, K, L /10, 20, 30, 40/
PRINT '(I5, I5, :, "  ""K AND L ARE:"" ", I5, I5)', I, J
PRINT '(I5, I5, "  ""K AND L ARE:"" ", I5, I5)', I, J
```

Output:

```
bbb10   20
bbb10   20 "K AND L ARE:"b
```

Dollar sign ($) (H)

The dollar sign character in an output format specification provides the same functionality as the colon descriptor, with the additional effect of suppressing the carriage return character at the end of the record output.

For terminal I/O, this means that a typed response follows the output on the same line at the terminal.

SYNTAX

```
$
```

DESCRIPTION

A dollar sign in an input format specification is ignored.

Example:

```
WRITE (6,"('What is the answer? ',$)")
READ (5,*) I
WRITE (6,*) "You entered ",I
```

Output:

```
What is the answer? 10
 You entered  10
```

A

The A descriptor is a character editing descriptor that can be used in either an input or output format specification. The A editing descriptor edits character or Hollerith data and must correspond to a character or Hollerith entity in an I/O list.

The A descriptor is repeatable.

SYNTAX

A*w*

DESCRIPTION

w Denotes the width of the input or output field in characters. The *w* can be omitted for character entities but not for Hollerith entities. If *w* is omitted, *w* is the length of the character item in the input or output list.

Input and Output of Character Data

For the input of character data, if *w* is greater than or equal to the length *n*, of the input list item, then *w* characters of the input field are read and the rightmost *n* characters are stored as the value of the character entity. If *w* is less than *n*, *w* characters are read, left-justified, with trailing blanks appended to the end of the character value.

For the output of character data, if *w* is greater than the length, *n*, of the character item in the output list, then an output field of *w* characters is written including any trailing blanks; the value is right-justified in the output field and leading blanks are supplied at the front. If *w* is less than or equal to *n*, the leftmost *w* characters of the character value of the output list item are written to the output field.

Examples:

```
CHARACTER *15 CHRS (2)
. . .
READ ( UNIT=11, FMT='(A10)' ) CHRS(1)
READ ( UNIT=11, FMT='(A20)' ) CHRS(2)
WRITE ( UNIT=6, FMT='(A)'   ) CHRS(1)
WRITE ( UNIT=6, FMT='(A)'   ) CHRS(2)
WRITE ( UNIT=6, FMT='(A5)'  ) CHRS(1)
WRITE ( UNIT=6, FMT='(A25)' ) CHRS(2)
```

Input Record:

```
ABCDEFGHIJKLMNOP
QRSTUVWXYZbbbbbbb
```

Output:

```
ABCDEFGHIJbbbbbb
STUVWXYZbbbbbbbb
ABCDE
STUVWXYZbbbbbbbb
```

Input and Output of Hollerith Data

Hollerith values can be read as the value of any numeric or logical list item using the `A` descriptor. The `A` descriptor is also used to write Hollerith values to output fields. For both the input and output of Hollerith data, the width, *w*, of the input and output field cannot be omitted.

Examples:

```
      INTEGER    IARRAY(5)
      . . .
      READ ( FMT='(5A4)', UNIT=11 ) IARRAY
      WRITE ( FMT=101, UNIT=6 ) IARRAY
101   FORMAT (' ', 5A4)
```

Input Record:

```
THE RAIN IN SPAINbb
```

Output:

```
bTHE RAIN IN SPAINbb
```

B, BN, and BZ

B, BN, and BZ are control descriptors that can be used in an input format specification. B, BN, and BZ are ignored if they appear in an output specification. These descriptors determine if blanks are treated as zeroes in a numeric input field. Once a B, BN, or BZ descriptor is specified in a format specification, it remains in effect until another blank control descriptor is encountered in the format.

SYNTAX

```
B
BN
BZ
```

DESCRIPTION

The B descriptor causes blank interpretation to return to the default mode for that unit.

The BN editing descriptor causes the compiler to ignore blank characters in a numeric input field and to right justify remaining characters as if the ignored blanks were leading blanks. A field of all blanks has a zero value.

The BZ editing descriptor causes the compiler to treat blank characters as zeroes in a numeric input field.

A BN or BZ descriptor is in effect from the moment it is detected in the format specification until another blank control descriptor is encountered. If the format specification is rescanned, any blank control setting remains intact.

In the absence of a BN or BZ descriptor, the interpretation of blanks is under the control of the BLANK specifier currently in effect for the unit.

Example:

```
READ ( 11, '(BZ, I5, BN, 3I5)' ) I, J, K, L
WRITE ( 6, '(" ", 4I7)' ) I, J, K, L
```

Input Record:

```
b1bb5b3bb741bb3bb3bb
```

Output:

```
bbbb1005bbbbbb37bbbb413bbbbbbb3
```

D

D is a numeric editing descriptor that can be used in either an input or output format specification. The D editing descriptor reads or writes real or double precision data and must correspond to a real, double precision, complex, or double complex item in an I/O list.

SYNTAX

D*w.d*

DESCRIPTION

The D descriptor operates exactly like the E*w.d* descriptor.

There is no alternative form for the D descriptor analogous to the E*w.d*E*e* form of the E descriptor.

Example:

```
DOUBLE PRECISION DN, QN
. . .
READ (11, '(BN, D10.6, BZ, D20.11, BN, D20.11)' )RN,DN,QN
WRITE ( 6, '(D30.7 / D30.12 / D30.12)' ) RN,DN,QN
```

Input Record:

```
b1537bbbE1b423.23423854599bb3829283.947569bE-3bbbb     //////
```

Output:

```
     0.1537000D-01
0.423234238546D+03
0.292839475690D+02
```

E

E is a numeric editing descriptor that can be used in either an input or output format specification. The E editing descriptor edits and converts real or double precision data in exponential form and must correspond to a real, double precision, complex, or double complex item in an I/O list.

SYNTAX

E*w.d*
E*w.d*E*e*
E*w.d*D*e*
E*w.d.e*

DESCRIPTION

w Specifies the width of the input or output field in characters.

d Specifies an unsigned integer indicating the number of digits to the right of the decimal point.

On input, a real or double precision value is read. If the value contains an exponent, the exponent must be of the form E⊖ or D*e* where *e* is a signed or unsigned power of 10 (zero is a valid power of 10, and the E or D can be omitted if the power of 10 is signed). A D exponent indicator is treated as an E exponent when input under the control of the E descriptor. The exponent cannot be omitted if E or D is present in the input value.

On input, the width indicator *w* must be large enough to include an exponent, any plus or minus signs, and an optional decimal point in the datum. A value containing a decimal point overrides a *d* specification in the E editing descriptor. A value can specify more precision than can be stored internally. Blanks in the value are interpreted as defined by the setting of the BLANK specifier for the unit or based on the BN and BZ editing descriptors.

On output, a decimal number is written with an exponent of the form E*e*, where *e* is a signed or unsigned power of 10. If the value has greater precision than can be printed, it is rounded by adding 0.5E−*d* to the magnitude of the value. The field width *w* must be large enough to write the integer portion of the number (including a leading zero and optional sign if necessary), a decimal point, any decimal portion, and an exponent of the following form.

E*snn* for all variable types.

where *s* is a plus or minus sign.

On output, a field written under E control is right justified. If the number of characters to be written exceeds the width *w*, the entire field is filled with asterisks.

Example:

```
DOUBLE PRECISION DN
. . .
READ ( 11, '(BZ, E10.6, BN, E20.11,)' ) RN, DN
WRITE ( 6, '(E30.7 / E30.12 )' ) RN, DN
```

Input Record:

```
15373b+03b423.bb233854599D-10bbbbbb38292839474169023755
```

Output:

```
     0.1537300E+30
0.423233854599E-07
```

Alternate forms of the E descriptor is written as follows:

E*w.d*E*e*
E*w.d*D*e*
E*w.d.e*

These forms are treated identically to the E*w.d* descriptor form for input and for output except that an exponent field of *e* digits is to be written. The maximum value for *e* is 7, with larger values truncated to 7. If the number of digits to be written for the exponent exceeds the width of *e*, then the entire field is filled with asterisks.

Example:

```
DOUBLE PRECISION DN
. . .
READ ( 11, '(BZ, E10.6E3, BN, E20.11E3)' ) RN, DN
WRITE ( 6, '(E30.7E3 / E30.12E3 )' ) RN, DN
```

Input Record:

```
15373b+03b423.bb233854599D-10bbbbbb38292839474169023755
```

Output:

```
     0.1537300E+030
0.423233854599E-007
```

F

`F` is a numeric editing descriptor that is used in either an input or output format specification. The `F` editing descriptor defines a real or double precision input or output field and must correspond to a real, double precision, complex, or double complex item in an I/O list.

SYNTAX

F*w.d*

DESCRIPTION

w Gives the width of the input or output field.

d Specifies the number of digits that are to be placed to the right of the decimal point.

On input, a field of *w* positions containing an integer, a number with a decimal point, or a number with an `E` or `D` exponent indicator is read and associated with the corresponding item in the input list. The input list item must be of type real, double precision, complex, or double complex. The field width *w* must allow for any decimal point, exponent field, and an optional leading sign. If a decimal point is used in the input field, it overrides the *d* specification. A value with a `D` exponent is treated the same as a value with an `E` exponent. The exponent indicator is followed by optional blanks followed by an optionally signed integer indicating a power of 10. An input value can specify more precision that can be maintained by the processor, but the number read must be in the domain of numbers capable of being assigned to real, double precision, complex, or double complex entities. Blanks in the input value are interpreted as defined by the setting of the `BLANK` specifier for the unit or based on the `BN` and `BZ` editing descriptor. A field of all blanks is read as zero.

On output, a real, double precision, complex, or double complex value is written into an output field of length *w.* If the value has greater precision than can be printed, it is rounded by adding `0.5E-`*d* to the magnitude of the value. The value is written right-justified in the output field. No leading zeroes are output except where a single zero is needed to the left of the decimal point. If the value exceeds *w* characters, the field is filled with asterisks. The width *w* must be large enough to accommodate the value, a decimal point, *d* decimal places, and an optional leading sign.

Example:

```
DOUBLE PRECISION DN
. . .
READ ( 11, '(BZ, F10.6, BN, F20.11)' ) RN, DN
WRITE ( 6, '(F30.7 / F30.12 )' ) RN, DN
```

Input Record:

```
15373b+03b423.bb233854599D-10bb
```

Output:

```
153.7300000
0.000000042323
```

G

G is a numeric editing descriptor that is used in either an input or output format specification. The G editing descriptor transmits real, double precision, complex, or double complex data whose magnitude is not known beforehand. The G descriptor must correspond to a real, double precision, complex, or double complex item in an I/O list.

SYNTAX

G*w.d*
G*w.d*E*d*

DESCRIPTION

For input, the G descriptor is the same as F descriptor input.

For output, the external representation depends on the magnitude M of the output value. The external representation is determined as follows:

Range of M	Action for Gw.d	Action for Gw.dEe
M < 0.1	cPEw.d	cPEw.dEe
0.1 <= M < 1.0	F(w-4).d,4X	F(w-(e+2)).d,(e+2)X
1.0 <= M < 10.0	F(w-4).d-1,4X	F(w-(e+2)).d-1,(e+2)X
. . .	. . .	. . .
10**(d-2) <= M < 10**(d-1)	F(w-4).1,4X	F(w-(e+2)).1,(e+2)X
10**(d-1) <= M < 10**d	F(w-4).0,4X	F(w-(e+2)).0,(e+2)X
M >= 10**d	cPEw.d	cPEw.dEe

The X descriptor indicates that 4 spaces (for G*w.d* formats) or *e*+2 spaces (for G*w.d*E*e* formats) are to follow the value in the field.

On output, a decimal number is written with an exponent of the form E*e*, where *e* is a signed or unsigned power of 10. If the value has greater precision than can be printed, it is rounded by adding 0.5E−*d* to the magnitude of the value. The field width *w* must be large enough to write the integer portion of the number (including a leading zero and optional sign if necessary), a decimal point, any decimal portion, and a three to four character exponent of the form:

E*snn*

where *s* is an optional plus sign or a minus sign if applicable. Thus, *w* should always be greater than or equal to *d*+*5*. If the output field is too small to contain the value, the field is filled with asterisks.

Example:

```
DOUBLE PRECISION DN
. . .
READ ( 11, '(BZ, G10.6, BN, G20.11 )' ) RN, DN, QN
WRITE ( 6, '(G30.7 / G30.12 )' ) RN, DN, QN
```

Input Record:

```
15373b+03b423.bb233854599D-10bb
```

Output:

```
153.7300
0.423233854590e-07
```

An alternate form of the G descriptor is written as follows:

G*w.d*E*e*

where *.d* is the number of significant digits.

This form is treated identically to the G*w.d* descriptor form for input and for output except that an exponent field of *e* digits is to be written.

H

H is a Hollerith editing descriptor that defines a Hollerith value to be output.

SYNTAX

*w*H*h1...hw*

DESCRIPTION

w Specifies the number of characters following the H in the Hollerith constant.

On output, the *w* characters *h1..hw* are written into an output field of length *w*. Care should be taken to ensure that exactly *w* characters are specified after the H, because additional characters are interpreted as another format item and generate an error, whereas fewer than *w* characters could result in subsequent format items being treated as Hollerith data. Note that two consecutive apostrophes in a Hollerith descriptor within a character constant are counted as one apostrophe.

The H editing descriptor may not be used on input.

Hollerith data can be read and written as values for list items using the A editing descriptor.

Example:

```
WRITE ( 6, '( 23HSAMPLE HOLLERITH OUTPUT )' )
```

Output:

```
SAMPLE HOLLERITH OUTPUT
```

I

I is a numeric editing descriptor that can be used in either an input or output format specification. The I editing descriptor defines an integer field and must correspond to an integer item in an I/O list.

SYNTAX

I*w*
I*w.m*

DESCRIPTION

w Specifies the width of the integer field to be read or written.

m Specifies an unsigned integer constant that must be less than or equal to the value of *w*. The m specification indicates that at least *m* digits must be written on output. The field width must be large enough to accommodate the integer as well as an optional sign.

On input, the I descriptor causes *w* positions in the input record to be read, converted to internal integer form, and associated with the corresponding item in the input list. The value read can consist of digits, blanks, and an optional leading sign, but no decimal point or exponent can be present. Blanks are interpreted as defined by the setting of the BLANK specifier for the unit or based on the BN or BZ editing descriptors. The value read must be in the domain of integer values or an error occurs. An m appearing in the input format specification is ignored.

On output, the I descriptor is associated with the corresponding item in the output list and *w* positions of the output record are written. The value is right-justified in the field, and leading blanks are output to fill the field if necessary. If *m* is specified, exactly *m* digits are written, including leading zeroes if necessary. If *m* is zero and the output is zero, a field of all blanks is written.

If the value contains too many digits to fit within the field, the entire field is filled with asterisks. If the value is not an integer, an error message is issued, and the field is filled with question marks.

Example:

```
INTEGER    I
I = 598449745
READ ( 11, '(2I5)' ) II, III
WRITE ( 6, '( 3(I10.10, 2X) / I5 )' )I, II, III, I+II+III
```

Input Record:

```
15373␣253
```

Output:

```
0598449745 0000001537 0000030253
*****
```

L

`L` is a logical editing descriptor that is used in either an input or output format specification. The `L` editing descriptor defines a logical field and must correspond to a logical item in an I/O list.

SYNTAX

L*w*

DESCRIPTION

w Specifies the width of the input or output field.

On input, *w* characters are read. If the first non-blank character read is a `T`, the value `.TRUE.` is stored internally; otherwise, `.FALSE.` is stored. An entirely blank field results in a false value.

On output, the `L` descriptor is associated with the corresponding item in the output list, which must be of type logical. The letter `T` is written right-justified in field *w* if the internal value is true, or `F` is written if the internal value is false. Leading blanks are supplied if the field width is greater than one.

O (H)

O is an octal editing descriptor that can be used in either an input or output format specification. The O editing descriptor defines an octal field and must correspond to an integer item in an I/O list. The value must be a 1- to 11-digit number formed from the digits 0, 1, 2, 3, 4, 5, 6, or 7.

SYNTAX

O*w*
O*w.m*

DESCRIPTION

w Specifies the width of the input or output field.

m Specifies an unsigned integer constant that must be less than or equal to the value of *w*. The m specification indicates that at least *m* digits must be written on output. The field width must be large enough to accommodate the integer as well as an optional sign.

On input, *w* positions of the input record are read. The value read must be unsigned and contain no decimal point, exponent indicator, or single quote. Blanks are interpreted as defined by the setting of the BLANK specifier for the unit or based on the BN or BZ editing descriptors. An *m* appearing in the input format specification is ignored.

On output, an octal number is written right-justified in the field *w* with leading zeroes, if necessary.

If *m* is specified, exactly *m* digits are written, including leading zeroes if necessary. If *m* is zero and the output is zero, a field of all blanks is written. If the value requires more than *w* positions, the entire field is filled with asterisks. No sign or leading single quote is written.

Example:

```
READ ( 11, '(BN, O5, BZ, 3O5)' ) I, J, K, L
WRITE ( 6, '( " ", O7 )' ) I, J, K, L
```

Input Record:

```
1  5 3  741  3  3
```

Output:

```
15
3007
41003
300
```

Q (H)

During a READ statement, the Q edit descriptor obtains the number of characters remaining to be transferred from the input record. The corresponding I/O list element must be of type INTEGER.

SYNTAX

```
Q
```

Example:

```
INTEGER NCHRS, IARRAY(50)

. . .

READ (5,"(F10.4,Q,50A1)") X,NCHRS,(IARRAY(I),I=1,NCHRS)
```

By placing the Q descriptor first in the format specification, the length of the input record may be obtained. On output, the Q descriptor has no effect, except that the corresponding I/O list element is skipped.

R (H)

R is a radix editing descriptor that can be used in either an input or output format specification. The R editing descriptor is patterned after P, the scale factor for floating point conversion. It defines a radix for integer I/O conversion and remains in effect until another radix is specified or format interpretation completes.

SYNTAX

[*n*]R

DESCRIPTION

n Specifies the radix to be used, where $2 < n < 36$.

The R editing descriptor should be used in conjunction with the SU editing descriptor.

S, SS, and SP

S, SS, and SP are control descriptors that determine whether plus signs are to be written in a numeric output field. S, SS, and SP are ignored if they appear in input specifications but apply to the I, F, E, D, and G descriptors during output. Once an S, SS, or SP descriptor is specified in a format specification, it remains in effect until another sign control descriptor is encountered in the format.

SYNTAX

```
S
SS
SP
```

DESCRIPTION

S and SS are identical and indicate that plus signs are to print as blanks in any numeric field that contains an optional leading plus sign. SP causes leading plus signs to print in numeric output fields.

An S or SS descriptor is in effect by default. An SP descriptor is in effect from the moment it is detected in the format specification until an S or SS descriptor is encountered. If the format specification is rescanned, any setting in effect for S, SS, or SP remains in effect.

Example:

```
DOUBLE PRECISION D, Q
. . .
READ ( 11, '(I5, F10.3, F15.4, E20.10)' ) I, R, D, Q
WRITE (6,'(SP, I10/F15.3 /F20.4/, S, F30.10') I, R, D, Q
```

Input Record:

```
320  632.3823  5927.985925537  4545645876343475763598 3D-9
```

Output:

```
+32000
+632.382
+5927.9859
45456458.7634347576
```

SU (H)

SU is a control descriptor that can be used in either an input or output format specification. The specifier indicates that integer values are to be interpreted as unsigned during input and output conversion. It remains in effect until another sign control specifier is encountered or until format interpretation completes.

SYNTAX

```
SU
```

DESCRIPTION

Unsigned integer values greater than (2**31 - 1) -- that is, any signed negative value -- cannot be read by Fortran input routines. All internal values are output correctly.

Example:

Used in conjunction with the radix editing descriptor, a binary dump could be formatted as follows:

```
200 FORMAT ( SU, 16R, 8I10.8 )
```

T, TL, and TR

`T`, `TL`, and `TR` are control descriptors that move the position pointer in an input or output data record.

SYNTAX

`T`*n*
`[`*m*`]T`
`TL`*b*
`TR`*f*

DESCRIPTION

n — Specifies an unsigned non-zero integer constant indicating the absolute character position to which the current position pointer is to be moved.

m — Specifies a tab to the next (or *n*th) eight-column tab stop. Thus, columns of data can be aligned without counting.

b — Specifies an unsigned integer constant indicating the number of positions the position pointer should be moved backward (i.e., to the left) in the current record. The *b* can be zero. If *b* is greater than or equal to the current position, the next data item is transmitted from position one of the record.

f — Specifies an unsigned integer constant indicating the number of positions the position pointer should be moved forward (i.e., to the right) in the current record. The *f* can be zero. The `TR` descriptor must not result in a forward position beyond the length of the input or output record.

The `T` descriptor moves the position pointer in an input or output data record to an absolute position forward or backward in the record.

The `T` descriptor must not specify a forward position beyond the length of the input or output record. For printed records, the first character is interpreted as a carriage control character.

The implementation of the `T` and `TL` descriptor is slightly imperfect. The implementation uses seeks, so if the unit (for example, a terminal) does not allow seeks, a run-time error message,

```
can't seek
```

is generated and the program aborts.

The `TL` descriptor moves the position pointer in an input or output data record backward to a relative position in the record.

The `TR` descriptor moves the position pointer in an input or output data record forward to a relative position in the record.

On input the T, TL, and TR descriptors permit the same portion of a record to be read more than once and possibly edited differently.

On output, the T, TL, and TR descriptors can move the position pointer so that data in an output record can be replaced. T, TL, and TR do not cause data to be replaced; however, subsequent editing descriptors can cause the data to be replaced. Positions that are skipped as a result of T, TL, or TR editing and that are not filled with any input or output data are filled with blanks.

Example:

```
    READ ( 11, 100 ) N1, N2, N3, N4, N5
100 FORMAT(T10, I5, T30, I5, TL17, I5, TR8, I5, T100, I5)
    WRITE ( 6, '(5I10)' ) N1, N2, N3, N4, N5
    WRITE ( 6, 101 ) N1, N2, N3, N4, N5
101 FORMAT(T10,I5.5,T2,I5.5,T50,I5.5,TL35,I5.5,TR11,I5.5)
```

Input Record:

```
1231 4562 3134 4821 4716 3714 9140 3800 3712 5721 4517
4533 4717
```

Output:

```
20313     40914     82104       9140         0
40914   20313     09140            00000         82104
```

X

X is a control descriptor that is used in either an input or output format specification. The X descriptor skips forward a specified number of positions in the input or output record.

SYNTAX

*n*X

DESCRIPTION

n Specifies an unsigned integer constant indicating the number of positions to skip forward from the current position.

The X descriptor must not result in a forward position that is beyond the length of the record.

Example:

```
    READ ( 11, 102 ) N1, N2, N3, N4, N5
102 FORMAT ( 3X, I5, 5X, I5, 3X, I5, 9X, I5, 4X, I5 )
    WRITE ( 6, '(5(I5, 3X))' ) N1, N2, N3, N4, N5
```

Input Record:

```
1231 4562 3134 4821 4716 3714 9140 3800 3712 5721 4517
4533 4717
```

Output:

```
31045   34048   47160    3800   20572
```

Z (H)

Z is a hexadecimal editing descriptor that can be used in either an input or output format specification. The Z editing descriptor defines a hexadecimal field and must correspond to an integer item in an I/O list. The value must be a 1- to 8-digit number formed from the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9, and the upper- or lower-case letters A, B, C, D, E, or F.

SYNTAX

Z*w*
Z*w.m*

DESCRIPTION

w Specifies the width of the input or output field.

m Specifies an unsigned integer constant that must be less than or equal to the value of *w*. The m specification indicates that at least *m* digits must be written on output. The field width must be large enough to accommodate the integer as well as an optional sign.

On input, *w* positions of the input record are read. The value read must be unsigned and contain no decimal point, exponent indicator, or single quote. Blanks are interpreted as defined by the setting of the BLANK specifier for the unit or based on the BN or BZ editing descriptors. An *m* appearing in the input format specification is ignored.

On output, a hexadecimal number is written right-justified in the field *w* with leading zeroes, if necessary.

If *m* is specified, exactly *m* digits are written, including leading zeroes if necessary. If *m* is zero and the output is zero, a field of all blanks is written. If the value requires more than *w* positions, the entire field is filled with asterisks. No sign or leading single quote is written.

Example:

```
READ ( 11, '(BN, Z5, BZ, 3O5)' ) I, J, K, L
WRITE ( 6, '( " ", Z8 )' ) I, J, K, L
```

Input Record:

```
F  5 3 a741  3  3
```

Output:

```
f5
41003
300
```

8
Subprograms and Statement Functions

8
Subprograms and Statement Functions

General Definition

Fortran *subprograms* are program units, independent of the main program, that are either written by the user or supplied with the Fortran compiler. Subprograms are classified as functions or subroutines. Functions appear in an expression and always return a value for use at the point of reference in the expression; functions also supply other values to the referencing program unit. Subroutines are invoked with a `CALL` statement and return zero or more values for use by the calling program unit.

Subprograms contain a sequence of executable statements as well as any specification statements local to the program unit. The main program and any functions or subroutines can be independently compiled or compiled together. If more than one program unit is compiled together in the same source program, only one main program can be present.

Functions are classified as:

- Intrinsic functions (those supplied with the Fortran compiler).
- Statement functions (one-statement functions that are written by the user and that pertain only to the program unit in which they are defined).
- External functions (those supplied by the user).

A data type is associated with each function when the function is defined.

A function is invoked when its name, followed by any arguments in parentheses, is *referenced* or used in an expression where the value of the function is needed.

A *subroutine* is invoked when its name, followed by any arguments in parentheses, is specified in a `CALL` statement. Unlike functions, the name of a subroutine is not assigned an explicit or implicit data type. Subroutines are either written by the user or supplied with the Fortran compiler.

A *host subprogram* is a subprogram that contains an internal subprogram. An *internal subprogram*:

- Is a subprogram defined within the body of another subprogram.
- May be a function or subroutine.
- Must not contain other internal subprograms.
- Is visible only within the host subprogram.

- May have arguments and may make uplevel references (see "Uplevel References (H)" on page 8-6) to variables that are visible within the host subprogram without additional declarations.

NOTE: Names of functions, subroutines, entry points for a subroutine or function cannot be the name of a common block or datapool.

Arguments

Arguments are a means of communicating values from one program unit to another. They are classified either as *dummy* arguments or *actual* arguments.

Dummy Arguments

Dummy arguments are used in statement function statements, `FUNCTION` statements, `SUBROUTINE` statements, or `ENTRY` statements to define an argument list for the statement function, external function, or subroutine. Dummy arguments represent the correct order, data type, and total number of actual arguments that are passed to the subprogram when the subprogram is invoked.

All dummy arguments must be variable names or unsubscripted array names (i.e., no constants, expressions, or subscripted array names are permitted). The length and data type of each dummy argument are defined explicitly in an explicit type statement in the subprogram or implicitly by the first character of the name.

A dummy argument name must not appear in an `EQUIVALENCE`, `PARAMETER`, `SAVE`, `INTRINSIC`, `DATA`, `AUTOMATIC`, `STATIC`, `COMMON` or `DATAPOOL` statement in a subprogram. The name of a common block or datapool can be the same as a dummy argument.

Dummy Arrays

Entire arrays can be passed to and from subprograms using dummy arrays. A dummy array is an unsubscripted array name that appears in the dummy argument list of an external function or subroutine. A dummy array is dimensioned in the subprogram in a `DIMENSION` statement or explicit type statement, but not in a `COMMON` or `DATAPOOL` statement. When a subscripted or unsubscripted array name is used as an actual argument, the address of the array or array element is passed to the subprogram; data values are not passed. The size of the actual array in the calling program unit can be larger than, smaller than, or equal to the size of the corresponding dummy array.

Adjustable Dimensions

A `DIMENSION` or explicit type statement in a subprogram contains fixed dimension declarators or adjustable dimension declarators. The appearance of an integer variable in a

dimension declarator instead of an integer constant or integer constant expression means that the DIMENSION statement is an adjustable DIMENSION statement. Use of integer variables as dimension declarators permits the same subprogram array to process more than one set of array data and to redefine the dimensions of an array passed in the subprogram argument list. The size of the dummy array can be smaller than or the same size as the actual array, but the dummy array cannot be larger than the actual array.

The following example illustrates the use of an adjustable DIMENSION statement in a subprogram:

Main Program:

```
      DIMENSION X(-2:2,5), Y(-2:2,10)
      INTEGER X, Y
      . . .
      CALL SUB (X, 1, 5, 3, 15)
      . . .
      CALL SUB (Y, 1, 5, 5, 25)
      . . .
      END
```

Subprogram:

```
      SUBROUTINE SUB (A, B, C, D, SIZE)
      INTEGER A, B, C, D, SIZE
      DIMENSION A (B:C,D)
      DO 99 I = 1, SIZE
      . . .
99    CONTINUE
      RETURN
      END
```

Adjustable dimensions cannot be used in array declarators appearing in a main program.

Assumed-Size Array Declarations

An array declaration in a subprogram can also be an assumed-size array declaration. Assumed-size array declarations force the compiler to use the size of the corresponding actual argument array as the size of the dummy array. The array is still declared in the subprogram, but the rightmost dimension upper bound is declared as an asterisk (*). The size of an assumed-size dummy array is the size of the corresponding actual array if the actual array is a non-character array. If the actual array is a non-character array element, the size of the assumed-size dummy array is:

$x - r + 1$

where x is the size of the actual array and r is the subscript value for the array element used as the corresponding actual argument. If the actual array is a character array, character array element, or character array substring reference, the size of the assumed-size array is:

$\mathrm{INT}((c - t + 1) / ln$

where *ln* is the length of an element of the dummy array, *c* is the total number of bytes in the actual array, and *t* is the beginning byte in the array for the corresponding actual argument.

An assumed-size array name cannot be used without subscripts in an input or output list in an I/O statement, as a unit identifier for an internal file in an I/O statement, or as a format identifier in an I/O statement.

Dummy Procedures

Dummy procedure names permit actual procedure names to be passed as arguments through several levels of program units. A dummy procedure name is a dummy argument that corresponds to an actual argument in a function reference or subroutine call. The actual argument is the name of an external function, intrinsic function, or subroutine. The dummy procedure name merely indicates that the corresponding actual argument is a subprogram name whose actual location is defined by the calling program.

Example:

```
INTRINSIC SIN, SQRT
A = DIFF (SIN, 25.)
B = DIFF (SQRT, 25.)
. . .
END


FUNCTION DIFF (F, Z)
DIFF = F(Z)
RETURN
END
```

A subprogram identifier to be passed as an argument must be identified as a subprogram. That is, it must be used as a subprogram or appear in an `EXTERNAL` statement, or both. Otherwise, it is classified as a scalar variable.

If the dummy procedure name specifies a subroutine name, the concept of data type does not apply.

The specific (but not the generic) name of an intrinsic function can be used as an actual argument.

Actual Arguments

Actual arguments are the entities that are specified in parentheses after the function or subroutine name, when the function or subroutine is invoked in a program unit. An actual argument can be:

- A constant or the symbolic name of a constant
- A variable name

- An unsubscripted or subscripted array name
- An expression
- A statement function reference
- An external function reference
- An external function name
- A subroutine name
- An alternate return specifier (see "CALL Statement" on page 8-15)

A statement function name cannot be used as a procedure name in an actual argument list.

Each actual argument must match a corresponding dummy argument defined in the initial statement or entry point of the subprogram. The correct order must be observed, and the data type of the actual argument must match that defined for the dummy argument. The same number of actual arguments must be specified as there are dummy arguments in the function or subroutine.

%VAL, %LOC, and %REF Argument List Intrinsics (H)

Three intrinsics provided to control the way Concurrent Fortran passes arguments are: `%VAL`, `%LOC`, and `%REF`.

`%VAL` (*arg*)	Passes *arg* by value rather than reference. *Arg* may be of any type and size. The value's type and size are retained. Complex values are passed as a structure with two consecutive real values. Character values are passed as a pointer to the string storage, equivalent to a C `char*`; the additional length parameter is suppressed.
`%LOC` (*arg*)	Passes the address of *arg*, resulting in the address of the address of *arg* being passed. For an `INTEGER*4` variable, it is equivalent to a C `int**`.
`%REF` (*arg*)	Undoes any previous `%VAL` action on arg. If `%VAL` was not previously used on *arg*, no action is performed.

NOTE: Six additional intrinsics, `%INT1`, `%INT2`, `%INT4`, `%LOG1`, `%LOG2`, and `%LOG4`, described in Chapter 9 may also be useful in argument lists to control the size of integer and logical constant and expression arguments.

Argument Association

When a function or subroutine is invoked, actual and dummy arguments are associated when control is transferred to the subprogram. Within the subprogram, the dummy argument name is used whenever the corresponding actual argument entity is needed. If the value of a dummy argument is changed in the subprogram, the value of the actual argument entity is also changed. This association of actual and dummy arguments permits

values to be passed back and forth between program units. Note, however, that if the corresponding actual argument is a constant, symbolic name of a constant, expression, function reference, or actual procedure name, the corresponding dummy argument cannot be used in a context in which its value would change (i.e., in an assignment statement, etc.).

Partial association of dummy and actual arguments can occur. If the length of a character dummy argument, *n*, is less than the length of the corresponding character constant, expression, variable, or array element in the actual argument list, only the leftmost *n* characters of the actual argument are associated with the dummy argument.

If an actual argument is an expression, the expression is evaluated before the actual arguments and dummy arguments become associated.

The association of actual arguments and dummy arguments is not retained between references to the function or subroutine.

CHARACTER Statements in Subprograms

A `CHARACTER` statement in a subprogram contains fixed length declarators, as defined above, or assumed-size declarators. The appearance of an asterisk (`*`) in a character variable declaration instead of an integer constant or length specification means that the character declaration is an assumed-size declaration.

If a dummy argument has a length of (`*`), the dummy argument assumes the length of the corresponding actual argument each time the subprogram is referenced. If the actual argument is an array name, the length assumed by the dummy argument is the length of a single array element in the corresponding actual array.

If an external function has a length of (`*`) declared in a function subprogram, the function name must appear as the name of a function in a `FUNCTION` or `ENTRY` statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit.

Uplevel References (H)

An internal subprogram shares the same namespace as its host subprogram and may make uplevel references to names declared within the host subprogram. Such a reference within an internal subprogram results in the reference of the identical variable within the host subprogram. If an uplevel reference to a host variable changes the value of the variable within an internal subprogram, the variable will also show the change within the host.

For each uplevel reference, type, dimension and other name specifications are identical between the host subprogram and internal subprogram.

Intrinsic Functions

Intrinsic functions (also called *built-in functions*) are supplied with the Fortran compiler. An intrinsic function reference is used in an expression and evaluated at its point of reference, and its value is made available at the point of reference when the function completes execution.

SYNTAX

name [([*a* [, *a*] ...])]

DESCRIPTION

name Specifies the specific or generic name of an intrinsic function.

a Specifies an actual argument being passed to the intrinsic function. One or more arguments must be specified, depending on the particular function, and all arguments must be of the same data type and be a valid data type for the function.

Refer to Chapter 9 for more detailed information on intrinsic functions.

Referencing Statement Functions

Statement functions are defined in statement function definitions (see Chapter 4) and are not subprograms. However, they are invoked exactly like external functions and can be referenced anywhere after their definition in the program unit in which they are defined. A statement function is referenced by name in an expression or in the actual argument list of a subprogram or of another statement function. The statement function's actual arguments, if any, are enclosed in parentheses after the name.

SYNTAX

name ([*a* [, *a*] ...])

DESCRIPTION

name Specifies the name of a statement function previously defined in the program unit.

a Specifies a valid actual argument for the statement function. Each *a* is associated with its corresponding dummy argument, and the order, number, and data type of each a must agree with the order, number, and data type of the corresponding dummy argument in the statement function definition. If *a* is an expression, the expression is evaluated before the actual arguments and dummy arguments become associated.

Once the statement function is evaluated, the value is converted to the data type of the statement function if necessary (using the type conversion rules for assignment statements), and the value is available for use at the point where the statement function was referenced.

An actual argument cannot be a character expression that concatenates an operand whose length is an asterisk, unless the operand is the symbolic name of a constant.

All components of actual arguments must be defined when the function is referenced.

A statement function can be referenced in another statement function definition as long as the statement function being referenced has been previously defined.

A statement function that does not contain any arguments is referenced as follows:

name ( )

External (User-Defined) Functions

A *user-defined function* is defined as a subprogram using a FUNCTION statement and can be referenced in an expression in the main program or in any other subprogram, but cannot reference itself either directly or indirectly. User-defined functions are external functions. Zero or more external functions can be present in a single source program.

FUNCTION Statement

A FUNCTION statement must be the first statement of an external function.

SYNTAX

```
[type] FUNCTION name ([d [, d] ...])
. . .
END
```

DESCRIPTION

type Declares the data type of the function.

name Specifies the name of the function.

d Specifies a dummy argument for the function.

The data type and optional length of a function must be declared in both the function subprogram and in any program unit that references it.

A function's data type is determined implicitly by the first letter of the function name, or is declared explicitly by specifying the data type in the FUNCTION statement or in an explicit type statement in the function subprogram.

If the type is specified in the FUNCTION statement, the type is one of the following:

```
INTEGER *1
BYTE
INTEGER *2
INTEGER *4
INTEGER
REAL
REAL *4
REAL *8
DOUBLE PRECISION
COMPLEX
DOUBLE COMPLEX
LOGICAL *1
LOGICAL *2
LOGICAL *4
LOGICAL
CHARACTER *n
```

The length specification is optional in all cases. The type specification determines the data type of the value returned at the point of reference. For data type `CHARACTER`, *n* can be any of the forms allowed for the `CHARACTER` type statement, except that an integer constant expression cannot include the symbolic name of a constant.

The length specification can alternatively be specified with the name of the function. For example:

```
INTEGER ABC*2(X,Y)
```

is equivalent to

```
INTEGER*2 ABC(X,Y)
```

If the data type of the function is not defined in the `FUNCTION` statement, the function's data type is defined by using the function name in an explicit type statement in the subprogram. The name is a global name and cannot be used to name any other program unit or dummy procedure.

The name must also be used at least once in the subprogram as a variable name (e.g., in an assignment statement) to give a value to the function. This variable can be defined and redefined and is returned to the referencing program as the function's value when a `RETURN` or `END` statement is executed in the function subprogram. If the variable is a character variable, it must not appear as an entity in a concatenation operation except in a character assignment statement.

Each *d* is a dummy argument that is a variable name, unsubscripted array name, or dummy procedure name. The dummy argument list identifies the order, number, and data type of the arguments that can be passed to the function in a function reference. If the array has adjustable dimensions, the dimensions must be declared in a `DIMENSION` or explicit type statement in the program unit. The parentheses `()` are necessary even if no dummy arguments are specified.

The function can return additional values to the referencing program, other than the value of the function, by redefining the value of dummy arguments during the execution of the function.

A dummy argument defined for an entry point is local to the sequence of statements between the entry point and the next `RETURN` or `END` statement. If one or more `ENTRY` statements appear between the entry point and the next `RETURN` or `END` statement, the `ENTRY` statements and any dummy arguments defined for the `ENTRY` statements are ignored. The same dummy argument name can be used in the argument list of more than one entry point of the function, but the name applies only to the entry point for which it was defined.

A `SUBROUTINE`, `PROGRAM` or `BLOCK DATA` statement cannot appear in a function subprogram.

A function subprogram can have more than one `ENTRY` or `RETURN` statement but only one `END` statement.

In the subprogram, if the length of the character variable representing the name of a character function is an asterisk in parentheses, the function has the length specified in the referencing program unit.

If a dummy procedure name appears as a dummy argument in a `FUNCTION` statement, then an actual procedure name must be specified as the corresponding actual argument. If a dummy procedure name specifies a function, then the specified function's data type must agree with the dummy procedure name's data type.

If an actual argument is an expression, a constant, or the symbolic name of a constant, its corresponding dummy argument must not be altered during the execution of the function.

Example:

```
      INTEGER FUNCTION FACT(N)
      FACT = 1
      DO 5 I = 2, N
5     FACT = FACT * I
      RETURN
      END
```

Referencing an External Function

SYNTAX

name ([*a* [, *a*] ...])

DESCRIPTION

name Specifies the name or entry point of an external function.

a Specifies an actual argument.

If arguments are not defined for the function, the function is referenced as follows:

name ( )

Actual arguments must agree in order, number, and data type with the corresponding dummy arguments in the FUNCTION statement or function ENTRY statement. An actual argument for an external function reference is one of the following:

- A constant or the symbolic name of a constant.
- An unsubscripted array name.
- A variable name or subscripted array name.
- An intrinsic function name. A dummy procedure name must be the corresponding dummy argument if an intrinsic function name is passed as an actual argument.
- The name of a subroutine, the name of a subroutine entry point, the name of another external function, or the name of an entry point in another external function.
- Any expression except a character expression that concatenates an operand whose length is an asterisk in parentheses, unless the operand is the symbolic name of a constant.

Functions use their dummy arguments as actual arguments when referencing other subprograms.

Example:

```
      EXTERNAL FACT
      INTEGER FACT
      PRINT, FACT(10)
      END
      INTEGER FUNCTION FACT (N)
      FACT = 1
      DO 1 I = 2, N
1     FACT = FACT * I
      RETURN
      END
```

User-Defined Subroutines

Zero or more user-defined subroutines can be defined in a source program. A subroutine subprogram is defined using a SUBROUTINE statement and is referenced using a CALL statement.

SUBROUTINE Statement

Subroutines are written as independent program units. A SUBROUTINE statement defines a subroutine, and the SUBROUTINE statement must be the first statement of the subroutine subprogram.

SYNTAX

```
SUBROUTINE name [ ([d [, d] ...]) ]
. . .
END
```

DESCRIPTION

name Specifies the symbolic name of the subroutine.

d Specifies a dummy argument for the subroutine.

The subroutine name is a global procedure name. The concept of data type does not apply to subroutine names and subroutine entry points.

Each dummy argument is a variable name, an unsubscripted array name, a dummy procedure name, or an asterisk. An asterisk denotes an alternate return (i.e., the corresponding actual argument is an alternate return specifier). An expression, subscripted array name, constant, or the symbolic name of a constant is not a valid dummy argument.

If dummy arguments are not defined for the subroutine, the subroutine is defined in either of the following ways:

```
SUBROUTINE name      SUBROUTINE name()
...                  ...
END                  END
```

A subroutine returns zero or more values to the calling program. Values are returned to the calling program by redefining the values of one or more of the dummy arguments. The value of any dummy argument can be changed as long as the corresponding actual argument is not a constant, the symbolic name of a constant, an expression, or an actual procedure name.

A CALL statement within a subroutine can call another subroutine, but it cannot call itself or call another entry point of the subroutine of which it is a part.

The symbolic name of a subroutine cannot be used as a variable in the subroutine.

A subroutine subprogram can contain any statement except a `FUNCTION`, `PROGRAM`, or `BLOCK DATA` statement.

A dummy argument name of type character whose length is an asterisk in parentheses must not appear as a concatenation operand except in a character assignment statement.

Example:

```
      INTEGER SIZE
      PARAMETER (SIZE=25)
      INTEGER VECTOR(SIZE)
      ...
      CALL FMAX (VECTOR, SIZE, MAXIMUM, LMAX)
      ...
      END


      SUBROUTINE FMAX (ARRAY, SIZE, MAXIMUM, LMAX)
      INTEGER SIZE
      INTEGER ARRAY(SIZE)
      MAXIMUM = ARRAY(1)
      LMAX = 1
      DO 10 I = 2, SIZE, 1
      IF ( ARRAY(I) .GT.  MAXIMUM ) THEN
      MAXIMUM = ARRAY(I)
      LMAX = I
      END IF
10    CONTINUE
      RETURN
      END
```

The preceding subroutine finds the maximum value in an array and the position in the array of the maximum value, and returns the two values to the calling program.

CALL Statement

A subroutine is referenced in a program unit using a CALL statement. Actual arguments, if any, appear after the subroutine name in the CALL statement and are enclosed in parentheses.

SYNTAX

```
CALL name [ ([a [, a] ...]) ]
```

DESCRIPTION

name Specifies the symbolic name of a user-defined or processor-supplied subroutine.

If arguments are not defined for the subroutine, either of the following CALL statements is permissible:

```
CALL name

CALL name ( )
```

Actual arguments must agree in order, number, and data type with the corresponding dummy arguments in the SUBROUTINE statement or subroutine ENTRY statement. If an alternate return specifier is used as an actual argument, the corresponding dummy argument must be an asterisk.

An actual argument for a subroutine call is one of the following:

- A constant or the symbolic name of a constant.
- An unsubscripted array name.
- A variable name or subscripted array name.
- An intrinsic function name.
- The name of a function, the name of a function entry point, the name of another subroutine, or the name of an entry point in another subroutine.
- Any expression except a character expression that concatenates an operand whose length is an asterisk in parentheses, unless the operand is the symbolic name of a constant.
- An alternate return specifier of the form:

  **s*

where *s* is the statement label of an executable statement in the calling program. An alternate return identifies a statement label in the calling program where execution is to continue when the subprogram completes execution. An asterisk without a statement label cannot be used as an actual argument.

Subroutines use their dummy arguments as actual arguments when referencing other subprograms.

Argument List Intrinsic Functions (H)

There are two built-in functions that govern the way arguments are passed, `%VAL` and `%REF`. They may be used only in the argument list of a `CALL` statement or function reference. For argument *a*, `%REF`(*a*) passes a by reference, which is the default. `%VAL`(*a*) passes the argument *a* by value; a 32-bit immediate value is always passed. If the argument is shorter than 32 bits, it is sign-extended. Refer to "%VAL, %LOC, and %REF Argument List Intrinsics (H)" on page 8-5 for additional information.

ENTRY Statement

An ENTRY statement is used within a function or subroutine to identify one or more alternate entry points into the function or subroutine. An ENTRY statement appears anywhere after a FUNCTION or SUBROUTINE statement but cannot appear in an IF block or within a DO loop.

SYNTAX

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

DESCRIPTION

name Specifies the symbolic name of an entry point for a subprogram.

d Specifies a dummy argument. The dummy argument list for a function or subroutine entry point follows the same rules for the dummy argument list for a FUNCTION statement.

Specification statements are permitted after an ENTRY statement, but not if any statement function or executable statements have preceded the ENTRY statement (see Table 2-1).

If an argument list is not defined, either of the following forms of the ENTRY statement is permissible:

```
ENTRY name

ENTRY name( )
```

If a function reference refers to an entry name with no arguments, the function must be referenced as follows:

```
name( )
```

Also, the function entry name must be used as a variable, in the sequence of executable statements following the ENTRY statement, to give a value to the function. The entry name can appear in an explicit type statement for a function.

If a subroutine call refers to an entry name without any arguments, the subroutine is called with either of the following:

```
CALL name

CALL name( )
```

ENTRY statements are non-executable statements.

An entry name may be referenced in any program unit except in the program unit that contains the name in an ENTRY statement.

The order, number, data type, and names of dummy arguments in an ENTRY statement may be different from the order, number, type, and names of dummy arguments in the FUNCTION statement, SUBROUTINE statement, or other ENTRY statements in the same program unit.

A dummy argument must not appear in an executable statement before it appears in an ENTRY, FUNCTION, or SUBROUTINE statement. The same dummy argument name can be used in the argument list of more than one entry point of the subprogram, but the name applies only to the entry point in which it is defined.

Within an external function, all entry names are associated with the name specified in the FUNCTION statement; therefore, if one function variable becomes defined, all associated function variables of the same type become defined and all associated function variables of a different type become undefined. The data type of the function name and of any entry names need not be the same, unless the data type is CHARACTER, in which case the function name and all entry names must be of type CHARACTER. For entry points of type CHARACTER, if one entry point is declared with length (*), all entry points must be declared with length (*); otherwise, the entry points may be of the same or different lengths.

The variable whose name is used to reference the function must be defined when a RETURN or END statement is executed in the subprogram. The variable name cannot appear in any executable statement prior to the ENTRY statement. The name cannot appear in a statement function definition unless the name appears as a dummy argument name for the statement function.

All ENTRY statements for a subprogram must appear before the END statement for the subprogram.

Within a subprogram, an entry name cannot be a dummy argument in a FUNCTION statement, a SUBROUTINE statement, or another ENTRY statement, nor can the entry name appear in an EXTERNAL statement within the subprogram.

An entry name in a function subprogram is referenced only in a function reference or as an actual argument, and an entry name in a subroutine subprogram is referenced only in a CALL statement or as an actual argument.

RETURN Statement

A RETURN statement terminates the execution of a subprogram and returns control to the referencing program unit. It appears only in a function or subroutine subprogram.

SYNTAX

```
RETURN [e]
```

DESCRIPTION

e Specifies an integer constant or integer expression.

If *e* is specified, it identifies the *e*th asterisk in the dummy argument list. Control is returned to the statement label in the calling program whose statement label is associated with the *e*th asterisk in the dummy argument list. If *e* is negative, zero, or greater than the number of asterisks specified as dummy arguments in the dummy argument list, control is returned to the next sequential statement following the referencing statement in the calling program unit.

Execution of an END statement has the same effect as a RETURN statement; therefore, a RETURN statement can be omitted. More than one RETURN statement is permitted in a single subprogram. Execution of a RETURN statement ends the association of actual and dummy arguments.

Execution of a RETURN or END statement results in all entities in the subprogram becoming undefined except:

- Entities in SAVE statements that were executed in the subprogram.
- Blank common entities.
- Entities defined in DATA statements within the subroutine or function that were not redefined or did not become undefined.
- Entities in a named common block that appear in the subprogram and appear in at least one other program unit that is referencing, either directly or indirectly, the subprogram. Entities in named common blocks defined in the main program do not become undefined when a RETURN or END statement is executed in a subprogram.

A RETURN statement cannot appear in a main program.

INTERNAL Subprograms (H)

An *internal subprogram* is a function or subroutine defined completely within the body of a subprogram, known as the *host subprogram*, and is visible only within the host subprogram. The internal subprogram may access host subprogram variables in addition to its own arguments and local variables. Zero or more internal subprograms may be present in a function, subroutine or main program.

SYNTAX

```
INTERNAL FUNCTION name ([d [, d] ...])
[ type name ]
. . .
END INTERNAL


INTERNAL SUBROUTINE name [ ([d [, d] ...]) ]
. . .
END INTERNAL
```

DESCRIPTION

type Declares the data type of the internal function. The type must not be placed on the `INTERNAL FUNCTION` line; instead, the type must be declared within the body of the internal function.

name Specifies the name of the internal function or subroutine.

d Specifies a dummy argument for the internal function or subroutine.

The `INTERNAL` keyword defines the start of the internal subprogram. The `END INTERNAL` keyword defines the end of the internal subprogram but not the host subprogram. Statements preceding the `INTERNAL` and following the `END INTERNAL` belong to the host subprogram.

An internal subprogram may be defined anywhere within the executable statements of the host subprogram. An internal subprogram may be referenced from anywhere within the host, including other internal subprograms. An internal subprogram must not directly or indirectly reference itself.

Statement labels within an internal subprogram must be unique within the host subprogram and all other internal subprograms. `FORMAT` statements anywhere within the host including other internal subprograms may be referenced by their statement labels. A `GOTO`, `ERR=`, etc. into or out of an internal subprogram is not allowed.

`ENTRY` statements, alternate returns, statement function definitions, and additional internal subprograms must not be used within an internal subprogram.

Within the body of an internal subprogram, names referenced may be dummy arguments, local variables or host variables. Dummy arguments and local variables are declared with the appropriate specification statements. Any host variables referenced are called *uplevel references* and are not declared within an internal subprogram. All attributes of uplevel

references are the same as their attributes in the host subprogram. For details, see "Uplevel References (H)" on page 8-6.

Additional considerations generally appropriate for functions and subroutines also apply to internal functions and internal subroutines.

Referencing an Internal Subprogram (H)

Host subprograms reference internal functions and internal subroutines the same way they reference external functions or user-defined subroutines. The name of an internal subprogram must not appear within an `EXTERNAL` statement in the host subprogram, or an external subprogram with the same name will be referenced instead of the internal.

The type of internal functions must be declared explicitly or implicitly within the host subprogram. Internal functions are referenced by name and return a value. See the following example.

Internal subroutines are not declared within the specification statements of the host subprogram. Internal subroutines are referenced by using the `CALL` statement. See the following example.

Non-dummy argument names declared explicitly within the internal subprogram, and implicitly declared names that are not uplevel references, are local variables within the internal subprogram. Explicitly declared names override an identical host name. Local variables of internal subprograms are not visible within the body of the host subprogram or other internal subprograms.

Internal subprograms that are not referenced or called are not otherwise executed. Execution of host statements continues from the last executable statement preceding the `INTERNAL` keyword to the first executable statement following the `END INTERNAL` keyword.

Example:

```
! The end result of calling SUB is the assignment of
! the integer value 39 to ten elements of the array
! argument.

SUBROUTINE SUB (JARR)
INTEGER D, I, JARR, IFUNC
PARAMETER (D = 10)
DIMENSION JARR(D)

I = IFUNC ()   ! references the internal function below
CALL ISUB (I)  ! calls the internal subroutine below

   INTERNAL SUBROUTINE ISUB (ARG)
   ! This internal subroutine assigns its dummy argument
   ! ARG to every element of the host array named JARR.
   ! JARR and its dimension constant D are uplevel
   ! references.  The loop index K is local to this
   ! internal subroutine.
   INTEGER ARG, K
   DO K = 1, D
      JARR(K) = ARG
   END DO
   END INTERNAL

   INTERNAL FUNCTION IFUNC ()
   ! This internal function returns an integer value.
   ! The return type is declared within the body of the
   ! internal function, as well as within the host (see
   ! above.)
   INTEGER IFUNC  ! declares the type of this function
   IFUNC = 39     ! assigns the return value
   RETURN
   END INTERNAL

END  !  This ends the subroutine SUB.
```

BLOCK DATA Subprogram

A BLOCK DATA statement is the first statement of a block data subprogram. A block data subprogram functions only at compile time and is used to define and give initial values to variables and array elements in common blocks.

SYNTAX

```
BLOCK DATA [name]
...
END
```

DESCRIPTION

name Specifies the optional name of a block data subprogram. The name, if specified, is global to the source program and must not be the name of a subprogram, main program, common block, or other block data subprogram.

Block data "subprogram" is a misnomer because a block data subprogram cannot contain executable statements and, therefore, is not a true program unit. However, the group of lines comprising a block data subprogram are placed outside the main program and are not a part of the main program.

The BLOCK DATA statement must be the first statement of a block data subprogram. Statements that can be placed in a block data subprogram include IMPLICIT, COMMON, PARAMETER, DIMENSION, SAVE, STATIC, EQUIVALENCE, DATA, END, and explicit type statements. The order of statements as defined in Table 2-1 must be observed.

There can be more than one block data subprogram in the source program.

Block data subprograms initialize entities in named or blank common blocks. If an entity in a common block is initially defined in a block data subprogram, all entities having storage units in the common block storage sequence must be specified even if they are not all initially defined. More than one common block can have entities initially defined in a single block data subprogram.

Only an entity in a common block is initially defined in a block data subprogram. Entities that are associated with an entity in a common block are considered to be in that common block.

The same storage cannot be initialized in more than one block data subprogram in the same executable program.

If an entity in a common block is initialized using a block data subprogram, a complete set of specification statements to establish the entire common block must be present even though some of the entities in the block do not appear in DATA statements.

Example:

```
DIMENSION A(5), B(5)
INTEGER C
COMMON I, J, K /ALPHA/ A, B, C
...
END
BLOCK DATA BD
DIMENSION A(5), B(5)
COMMON I, J, K /ALPHA/ A, B, C
INTEGER C
DATA I, J, K, C / 1, 2, 3, 100 /
DATA A / 5*0.  /, B / 5*1.2 /
END
```

Inter-Language Procedure Interface (H)

To write C procedures that call or are called by Fortran procedures, you must know the conventions for procedure names, data representation, return values, and argument lists followed by the compiled code.

This section discusses procedure names, data representations, common blocks and datapools and their corresponding C representations.[1]

Procedure Names (H)

The compiler appends an underscore to the name of a Fortran procedure or function and two underscores to the name of a common block. This underscore distinguishes the procedure, function or block from a C procedure or external variable with the same name. To avoid conflict with subroutine names you assign, Fortran library procedure names have embedded underscores.

Data Representations (H)

Following is a table of corresponding Fortran and C declarations.

Fortran	C
`INTEGER*1` *var*	`char` *var*;
`INTEGER*2` *var*	`short int` *var*;
`INTEGER` *var*	`long int` *var*;
`BYTE` *var*	`char` *var*;
`LOGICAL*1` *var*	`char` *var*;
`LOGICAL*2` *var*	`short int` *var*;
`LOGICAL` *var*	`long int` *var*;
`REAL` *var*	`float` *var*;
`DOUBLE PRECISION` *var*	`double` *var*;
`COMPLEX` *var*	`struct { float` *r, i*; } *var*;
`DOUBLE COMPLEX` *var*	`struct { double` *dr, di*; } *var*;
`CHARACTER*6` *var*	`char` *var*[6];

The Fortran standard specifies that `INTEGER`, `LOGICAL`, and `REAL` data occupy equal amounts of memory.

1. Users interested in interfacing Ada and Fortran can consult the *HAPSE Reference Manual*.

COMMON Blocks (H)

Accessing Fortran common blocks from C is best accomplished by using C structures. To properly access a common block's variables, the same type and order for variables must be used in the structure declaration. Consider the following example:

```
COMMON /SIMPSON/ HOMER,MARGE,BART,LISA,MAGGIE
CHARACTER*8 HOMER
COMPLEX MARGE
REAL*8 BART
LOGICAL*2 LISA
INTEGER MAGGIE
```

A C structure equivalent to this common block would be:

```
struct simpsons  {
    char homer[8];
    struct { float r, i; } marge;
    double bart;
    short int lisa;
    int maggie;
};
```

and to access the common block as a structure from C:

```
extern struct simpsons simpson__;
simpson__.maggie = 4;
...
```

For more information about common blocks, see "COMMON Statement" on page 4-9.

Datapools (H)

Outside of Concurrent Fortran, a datapool as a single storage object does not exist. Datapool storage consists of many discrete space requirements, one for each datapool variable or equivalence class. An equivalence class, as interpreted for datapool storage, is a group of all datapool variables equivalenced together; each variable name in an equivalence class is still associated with a distinct object address, and is referenced directly. This basic scheme is quite different from common block storage. Common block storage has one name, the name of the common block, and is sized to hold space for all common variables. Common block variables are referenced in object code as offsets from the beginning of common block storage and do not have a link name of their own, thus the reason for the basic Fortran requirement that all common block declarations in source files use identical types and order for variables; the proper offset must be known at compile time.

Datapool variable link names are a hybrid formation of the datapool and variable names. The basic form is "*datapool*__$*variable*". Thus, a Fortran declaration of the form

```
DATAPOOL /REM/ BUCK, STIPE
```

would, in object files, cause the datapool variables "BUCK" and "STIPE" to be referenced as "rem__$buck" and "rem__$stipe", respectively. Note the lowercase letters. This is the name, minus the leading underscore, that is used for C's extern.

Because each datapool variable has discrete storage, a datapool cannot be interpreted as a structure in C, as can common blocks. extern must be used for each desired datapool variable, but it is not required that extern be used for every variable in a datapool. All type information must still be accurate, but it is not necessary to preserve order of variables. Users accessing datapool variables from several C source files are encouraged to create a common C file containing all datapool variable declarations and use the C #include feature to guarantee consistent declarations.

For more information about datapools, see "DATAPOOL Statement (H)" on page 4-16.

Equivalenced Variables (H)

Users attempting to reference in C overlapping variables in an equivalence class should be cautious. Incorrect code may result unless the user precisely communicates the variables' interdependence to the compiler via unions or pointers. In particular, translating array equivalences that involve equivalencing to the middle of an array should be avoided unless absolutely necessary.

For datapools, the user may also specify all such variables as volatile to prevent incorrect interpretation and optimization of their usage.

Example:

```
CHARACTER*10 STR
INTEGER I, IARR(10), J
REAL R, RARR(10)
EQUIVALENCE (STR,J), (I,R), (IARR(4),RARR)
```

can be expressed in C as:

```
union { char str[10]; int j } str_j_equiv;
union { int i; float r } i_r_equiv;
    /* iarr is sized to hold both iarr and rarr */
int iarr[10 + (4 - 1)];
    /* if iarr & rarr were in a common block, just */
    /* iarr would appear in the struct declaration */
float *rarr;
rarr = &iarr[3];
```

For more information about equivalencing, see "EQUIVALENCE Statement" on page 4-22. For more information about volatiles, see "VOLATILE Statement (H)" on page 4-40.

Return Values (H)

A function of type INTEGER, LOGICAL, REAL, or DOUBLE PRECISION is declared as a C function that returns the corresponding type. A COMPLEX or DOUBLE COMPLEX function is equivalent to a C routine with an added, initial argument that points to the returned value. Thus,

```
COMPLEX FUNCTION F(...)
```

equals

```
struct { float r, i;} *temp;
f_(&temp,...)
```

A function with a character value is equivalent to a C routine with two extra initial arguments; a data address and a length. Thus,

```
CHARACTER*15 FUNCTION G(...)
```

is the same as

```
g_(result,length,...)
char result[];
long int length;
```

and could be invoked in C by

```
char chars[15];
...
g_(chars, 15L, ....);
```

The compiler invokes subroutines with alternate return values as if they are functions with INTEGER values. These values specify which return value to use. Alternate return arguments (statement labels) are not passed to the function, but are used to do an indexed branch in the calling procedure. If the subroutine does not have any entry points with alternate return arguments, the returned value is undefined. The statement

```
CALL NRET(*1, *2, *3)
```

is treated exactly as if it is the computed GOTO,

```
GOTO (1,2,3), NRET( )
```

Argument Lists (H)

All Fortran arguments are passed by address. CHARACTER arguments and dummy procedure arguments have some special characteristics. Normally, procedure arguments are passed as the address of the entry point of the procedure.

Character arguments have two components: a pointer to the storage for the string that is passed in sequence with the other arguments, and the length of that storage, which is passed by value as an extra argument after the regular arguments. Otherwise, they are contiguous with the regular arguments.

Given the following call

```
EXTERNAL F
CHARACTER*7 S
INTEGER B(3)
...
CALL SAM(F, B(2), S)
```

the equivalent in C code is normally

```
int f();
char s[7];
long int b[3];
...
sam_(f, &b[1], s, 7L);
```

Mixing C and Fortran Input/Output (H)

Files are opened in a slightly different manner in Fortran than in C. In C, the **`fopen(3S)`** function call makes explicit the desired access types and produces an error if unable to fulfill the request. Fortran's `OPEN` statement, on the other hand, is more "passive". Access types are not made explicit by the user and are determined through the user's subsequent operations on the file. A file is known to have write access only after the first Fortran `WRITE` on the unit, and read access after the first `READ`. If both are performed on the same unit, the file is opened with update access, allowing both reading and writing. Consequently, C I/O operations on a file opened from Fortran may result in an access error.

The Concurrent Fortran library provides four C-callable routines to facilitate C and Fortran I/O on the same file:

```
int _now_reading_on_unit  ( int * unit)
```

`_now_reading_on_unit()` coerces the file associated with the Fortran unit to have an access type that allows reading. The unit number is passed by address. Return status is 1 for success (file enabled for requested operation) or 0 if unable to change the requested status.

```
int _now_writing_on_unit (int * unit)
```

`_now_writing_on_unit()` coerces the file associated with the Fortran unit to have an access type that allows writing. The unit number is passed by address. Return status is 1 for success (file enabled for requested operation) or 0 if unable to change the requested status.

```
int _now_updating_on_unit  (int * unit)
```

`_now_updating_on_unit()` coerces the file associated with the Fortran unit to have an access type that allows both reading and writing. The unit number is passed by address. Return status is 1 for success (file enabled for requested operation) or 0 if unable to change the requested status.

```
FILE * _fortran_fd  (int * unit)
```

`_fortran_fd()` returns the stream pointer associated with a Fortran unit number. The unit number is passed by address.

Before performing C I/O on a file opened from Fortran, one of the access type coercion routines should be called to ensure proper stream access. Do not attempt to modify access of a Fortran file by modifying the `FILE` stream structure directly, as the Fortran library contains internal data structures associated with each unit that must remain consistent.

The coercion routines are additive. If `_now_reading_on_unit()` and then `_now_writing_on_unit()` are called for the same unit, the resulting access type for the unit is update.

Example code showing the use of `_fortran_fd()` and `_now_updating_on_unit()` follows. Note that the first write fails with a bad file number error `EBADF`. The write following the call to `_now_updating_on_unit()` completes successfully.

Fortran main program:

```
      PROGRAM IOTEST
      LFN = 10
      OPEN (UNIT=LFN, FILE="TESTFILE", STATUS="UNKNOWN")
C         call a C routine to do a write to the opened file
      CALL CSUB (LFN)
      END
```

C subroutine:

```
#include <stdio.h>
csub_(lfn)
int *lfn;
{
    FILE *fp;
    extern FILE *_fortran_fd();
    int status, fd;
    char *buf="123456789\n";

    fp = _fortran_fd( lfn );
    fd = fileno( fp );

    printf(" 1st write...");
    if (write(fd, buf, 10) < 0) {
        perror(" 1st write - error");
    }

    printf(" _now_updating_on_unit ");
    if(_now_updating_on_unit(lfn) == 0) {
        printf(" error - updating");
    }

    printf(" 2nd write...");
    if ((status = write(fd, buf, 10)) < 0) {
        perror ("2nd write - error");
        return ;
    }
```

```
    printf(" 2nd write - success nbytes= %d\n", status);
}
```

Calling C Functions Directly (H)

Separate global object code routine name-spaces are used by Fortran and C. In Fortran, global routine names end with one underscore, whereas in C, global names are not appended with underscores unless explicitly added by the user. Thus, to be able to call a C routine from Fortran, an underscore must be appended to the C routine name. System services and other C routines, for which access to the source is not available, must be called through an intermediate Fortran-callable C routine.

The Concurrent CEXTERNAL keyword, the CTOF77STR, F77TOCSTR and F77TOCSTR_TRIM intrinsics, the Concurrent implementation of the %VAL argument list intrinsic, and the documentation in this section are meant to assist the user in directly calling any C function or system service from Fortran.

See also "CEXTERNAL Statement (H)" on page 4-8, "%VAL, %LOC, and %REF Argument List Intrinsics (H)" on page 8-5, and the man pages for the CTOF77STR, F77TOCSTR and F77TOCSTR_TRIM intrinsics.

CEXTERNAL Declaration (H)

CEXTERNAL is a keyword similar in syntax and semantics to the EXTERNAL keyword with the additional action of ordering the compiler not to append an underscore to a reference of a listed name. To call the **nice(2)** routine, place the following declaration in the Fortran source program (recall that Fortran converts external names to lowercase by default):

```
CEXTERNAL NICE
```

Function Return Type Declaration (H)

Users must declare the return type of the C routines properly if the return value is to be interpreted correctly. For **nice(2)** the proper declaration is:

```
CEXTERNAL NICE
INTEGER*4 NICE
```

Passing Arguments by Value (H)

Standard Fortran passes all routine arguments by reference. Many C routines expect at least some of their arguments to be passed by value. Concurrent Fortran provides the

%VAL argument list intrinsic to pass arguments by value. To call **nice(2)** the user can write something similar to the following:

```
CEXTERNAL NICE
INTEGER*4 NICE, INCR_VAL, STATUS
PARAMETER (INCR_VAL = 8)
...
STATUS=NICE(%VAL(INCR_VAL))
```

If a C routine is actually expecting the address of a non-character argument, %VAL is not necessary.

Arguments of any type (integer, logical, real, double precision, complex, double complex, character) are accepted by Concurrent' %VAL and retain their size when passed. The address of character storage only is passed for character %VAL arguments -- the extra length parameter is suppressed.

Converting Character Arguments and Values (H)

To convert from Fortran and C character variable formats, three intrinsic functions are provided. Note the reverse sense of their arguments, that is, destination precedes origin.

```
INTEGER*4 FUNCTION  F77TOCSTR (C_DEST, F77_ORIG)
INTEGER*4 FUNCTION  F77TOCSTR_TRIM (C_DEST, F77_ORIG)
```

These copy the origin into the destination with the string storage converted to a C type null-terminated string. F77TOCSTR converts the entire value. F77TOCSTR_TRIM strips trailing blanks before converting. Both intrinsics return the equivalent of a C char* to the string storage for C_DEST. This value may be passed by %VAL to C routines which expect a char* argument -- this is useful in argument lists.

Example:

```
CEXTERNAL CFUNC
CHARACTER  F77_ORIG*10, C_DEST*11
...
CALL CFUNC(...,%VAL(F77TOCSTR_TRIM(C_DEST,F77_ORIG)),...)
```

Both arguments must be Fortran character variables. The destination must be large enough to hold the converted value -- truncation results otherwise.

```
INTEGER*4 FUNCTION  CTOF77STR (F77_DEST, C_ORIG)
```

CTOF77STR copies a C string to a Fortran string, removing the null byte and either truncating or blank padding as necessary. Only the Fortran destination string need be an actual Fortran character variable, so C functions that return char* may have their return value assigned to an INTEGER*4 variable which may then be passed by %VAL to this intrinsic, where it is treated as a char*. CTOF77STR returns the number of characters copied from the C string into the destination string.

As mentioned in the above paragraph, a Fortran routine which calls a C function returning a char* may assign the return value to an INTEGER*4 variable and then pass this by %VAL to either a C routine expecting a char* or to CTOF77STR as the origin argument.

Simulated Structures (H)

Several system services expect as an argument a pointer to an instantiation of a system-defined structure type, i.e., `struct termios`. Concurrent Fortran provides the `POINTER` statement, which may be used to construct a pointer block whose storage layout is identical in size and/or contents to a C structure. This example is for demonstration purposes; a standard-conforming method for accomplishing a call to **`tcgetattr(2)`** from Fortran may be found in the POSIX-defined **`pxftcgetattr(3F)`**.

The `POINTER` statement associates a pointer variable with a list of variables; the locations of these variables are at offsets relative to the memory address held in the pointer variable. The offsets of these based variables are determined by their ordering in the declaration list combined with type and size information. The declaration of `struct termios` from `<sys/termios.h>` is:

```
struct termios {
        tcflag_t        c_iflag;  /* unsigned long */
        tcflag_t        c_oflag;
        tcflag_t        c_cflag;
        tcflag_t        c_lflag;
#define NCCS 19
        cc_t            c_cc[NCCS];  /* unsigned char */
};
```

The equivalent pointer block declaration is:

```
INTEGER*4 TERMIOS_PTR, NCCS
PARAMETER (NCCS=19)
POINTER /TERMIOS_PTR/ IFLAG,OFLAG,CFLAG,LFLAG,CC(NCCS)
INTEGER*4 IFLAG, OFLAG, CFLAG, LFLAG, CC
CHARACTER CC
```

The following example shows a direct call to **tcgetattr(2)** from Fortran, using a pointer block to simulate `struct termios`.

```
INTEGER*4 TERMIOS_PTR, NCCS
PARAMETER (NCCS=19)
POINTER /TERMIOS_PTR/ IFLAG,OFLAG,CFLAG,LFLAG,CC(NCCS)
INTEGER*4 IFLAG, OFLAG, CFLAG, LFLAG, CC
CHARACTER CC
EXTERNAL MALLOC, FILENO
INTEGER*4 MALLOC, FILENO

CEXTERNAL tcgetattr
INTEGER*4 tcgetattr

INTEGER*4 STAT, UNIT

!  Must allocate space for the structure.
TERMIOS_PTR = MALLOC (SIZEOFBLOCK (TERMIOS_PTR))
...
! Pass the pointer variable by value, not address.
STAT = tcgetattr(%VAL(FILENO(UNIT)), %VAL(TERMIOS_PTR))
```

Pointer blocks are also useful for interpreting `struct *` return values from C functions. The C library function **localtime(3C)** returns a `struct tm *`.

```
 INTEGER*4 TM_PTR
 POINTER /TM_PTR/ TM_SEC, TM_MIN, TM_HOUR, TM_MDAY,
&   TM_MON, TM_YEAR, TM_WDAY, TM_YDAY, TM_ISDST
 INTEGER*4 TM_SEC, TM_MIN, TM_HOUR, TM_MDAY, TM_MON,
&   TM_YEAR, TM_WDAY, TM_YDAY, TM_ISDST

 CEXTERNAL localtime, time
 INTEGER*4 localtime, time

 INTEGER*4 NOW

 NOW = time(%VAL(0))
 ! Passed by address, not value.
 TM_PTR = localtime(NOW)

 ! Now that the pointer variable has a valid address, the
 ! based variables may be used.  tm.tm_yday is returned
 ! by localtime
  WRITE (*,*) TM_YDAY
```

Alternatively, common blocks or equivalence classes may be used to guarantee a specific storage layout, although the method is more clumsy and prone to error. The next example shows a translation of the **`localtime(3C)`** example implemented with common blocks.

```
 IMPLICIT INTEGER*4 (T)
 COMMON /TM_STRUCT/ TM_SEC, TM_MIN, TM_HOUR, TM_MDAY,
&   TM_MON, TM_YEAR, TM_WDAY, TM_YDAY, TM_ISDST
 CEXTERNAL LOCALTIME, TIME, MEMCPY
 INTEGER*4 LOCALTIME, TIME, MEMCPY, NOW, TM_POINTER,
&   SIZEOF_TM_STRUCT
 PARAMETER (SIZEOF_TM_STRUCT=9*4)
 ...
 NOW = TIME(%VAL(0))    ! see time(2) and localtime(3C)
 TM_POINTER = LOCALTIME(NOW)  ! pass by address, not value
 ! memcpy returns the value of its first argument
 ! -- we may ignore it
 CALL MEMCPY(%VAL(TM_POINTER), TM_SEC, SIZEOF_TM_STRUCT)
```

C Structure Packing Rules (H)

When simulating C structures in Fortran, careful attention must be paid to the actual physical alignment of the structure fields. The following rules apply.

- Fields are packed as tightly as possible as long as they do not cross a boundary of their "natural" type. This means that shorts cannot cross two byte boundaries, long words cannot cross four byte boundaries, and doubles cannot cross eight byte boundaries. NOTE: Concurrent Fortran does not restrict doubles in common blocks to eight-byte boundaries if the **`-Qalign_double=4`** option is used.
- An aggregate type (e.g., structures or union) is aligned according to the most restrictive boundary of all its members. The total aggregate size is also rounded up to a size which is a multiple of the size of the most restrictive type of all members.

Refer to the *C Reference Manual* for further details.

Primitive System Types (H)

Simulating system types that are equivalent to primitive data types (i.e., `typedef int key_t`) is the responsibility of the user. These system types are documented in the system include files.

Accessing errno and System Error Messages (H)

Many system services set the C global status variable, `errno`, if an error occurs. The Fortran library provides three routines for getting the value of `errno` and the contents of

system error messages: `perror`, `gerror`, and `ierrno`. The **`perror(3F)`** routine writes an appropriate message of the last detected system error to logical unit 0. The **`gerror(3F)`** routine returns the system error message appropriate to the last detected error as a character string. The **`ierrno(3F)`** routine returns the error number of the last detected system error, that is, the current value of `errno`.

Refer to the **`perror(3F)`** man page which includes information on all three error routines.

9
Fortran Library

9
Fortran Library

Functions and Routines

The Fortran library contains the Fortran intrinsic functions, as well as several functions that are in addition to the Fortran 77 standard.

This chapter gives an alphabetical summary of these functions. Descriptions include syntax and argument descriptions. More information about each of the intrinsic functions can be found in the system manual pages for the functions.

Intrinsic Functions

Intrinsic functions are supplied with the Fortran compiler. An intrinsic function is used in an expression and evaluated at its point of reference, and its value is made available to the expression at the point of reference when the function completes execution. The Fortran library contains all the intrinsic functions required by the Fortran 77 standard as well as additional functions, which are noted in this chapter with an asterisk (*).

SYNTAX

name (*a* [, *a*...])

DESCRIPTION

name Specifies the specific name or generic name of an intrinsic function.

a Actual argument being passed to the intrinsic function. One or more arguments must be specified, depending on the particular function. All actual arguments in the function reference must be of the same data type, and the order and number of actual arguments must agree with the order and number of arguments defined for the function.

A function argument can be a subscripted array name, a variable name, a constant or the symbolic name of a constant, other functions of the correct type, or any valid expression except a character expression that concatenates an operand of length asterisk, unless the operand is the symbolic name of a constant.

If an actual argument is another intrinsic function reference, the inner function reference must result in a value with a valid data type for the outer intrinsic function. For example, a

logical function reference (e.g., LGE, LGT, LLE, or LLT) cannot be used as an actual argument for a numeric intrinsic function.

All arguments must be defined when the function reference is executed. Arguments for which the result is out of the numeric range of the compiler or for which the result is not mathematically defined (e.g., division by zero) may not be arguments for an intrinsic function.

Syntax of the argument list for each intrinsic function is given in Table 9-1.

Generic and Specific Names

Intrinsic functions that perform the same task but which use different data types have a specific name for each operation of a particular data type.

Examples:

SIN	Real version of the sin function
DSIN	Double precision version of the sin function
CSIN	Complex version of the sin function
ZSIN	Double complex version of the sin function.
CDSIN	Double complex version of the sin function (alternate name).

The generic name of a function can be used in place of all specific names for the function. The generic name permits arguments of any data type allowed by the compiler to be used in the function reference. Based on the data type of the arguments, the compiler references the appropriate function by its specific name. Use of the generic name for a function is helpful if the data type of the function needs to be changed later.

Some intrinsic functions have limited generic names. These names may be used in place of some, but not all, of the specific names of a function. For instance, IABS can be given an argument of INTEGER *2 or INTEGER type. The generic name accesses all the specific names of the function, including the ones which may be accessed by the limited generic name. By default, integer constants passed to generic intrinsic functions are treated as INTEGER *4 entities unless there are other INTEGER *2 parameters to the intrinsic function, in which case they are treated as INTEGER *2 constants.

An intrinsic function name is no longer the name of an intrinsic function in a program unit if the name is used as a statement function name, as a variable name or array name in an explicit type statement, or as a dummy argument name; however, the name is an intrinsic function name in other program units. If the name is used as a variable or array name in a common block, the name cannot be referenced in any program unit of the source program as an intrinsic function.

Use of a generic name in an INTRINSIC or EXTERNAL statement does not affect its generic properties. Use of an intrinsic function name as the name of a main procedure, subroutine, subroutine entry point, block data subprogram, or common block does not affect its use as an intrinsic function name.

The following rules apply to the use of specific names:

- A specific name can be used as a function reference if all arguments are of the correct data type.
- The data type of an intrinsic function cannot be changed by specifying its specific name in an explicit type statement.
- A specific name cannot be used as an actual argument unless its name appears in an `INTRINSIC` statement. The names of intrinsic functions for performing type conversion, lexical ordering, or selection of smallest or largest values may not be used as actual arguments.
- Specific and generic names are local to a program unit; however, if the name is used in an `EXTERNAL` statement, the name is no longer available for use as an intrinsic function name in that program unit.
- Non-standard specific names have been added to facilitate additional degrees of precision for complex functions.

An `IMPLICIT` statement cannot be used to change the data type of a generic or specific function name.

Summary of Concurrent Fortran Intrinsic Functions

Table 9-1 lists the intrinsic functions specified in the Fortran 77 standard, as well as some specific functions added to accommodate non-standard data types. The generic name (if any) is listed in the first column along with the reference to the man page which contains complete information about the function. The limited generic name (if any) is in the second column, and the specific name is in the third column. If a specific name does not exist, then it is indicated by two dashes (i.e., "- -"). The data type of the arguments and the data type of the result are in the fourth and fifth columns, respectively, with `DOUBLE` abbreviated to DBLE. Finally, the sixth column contains the number of arguments the intrinsic function allows. If a note to the intrinsic function is necessary, it is included in this column in square brackets (i.e., "[1]").

Functions or capabilities that are not in the Fortran 77 standard are noted with asterisks.

Table 9-1. Concurrent Fortran Intrinsic Functions

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
ABS	IABS	IABS *JIABS *IIABS	INTEGER INTEGER INTEGER *2	INTEGER INTEGER INTEGER *2	1
(**abs(3F)**)		ABS DABS CABS *CDABS *ZABS	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC REAL DBLE PREC DBLE PREC	[2]

Table 9-1. Concurrent Fortran Intrinsic Functions (Cont.)

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
ACOS (**acos(3F)**)		ACOS DACOS	REAL DBLE PREC	REAL DBLE PREC	1
(**aimag(3F)**)	AIMAG	AIMAG *DIMAG	COMPLEX DBLE CMPX	REAL DBLE PREC	1
AINT (**aint(3F)**)		AINT DINT	REAL DBLE PREC	REAL DBLE PREC	1
(**max(3F)**)	AMAX0	*AJMAX0 *AIMAX0	INTEGER INTEGER *2	REAL REAL	>1
(**min(3F)**)	AMIN0	*AJMIN0 *AIMIN0	INTEGER INTEGER *2	REAL REAL	>1
ANINT (**round(3F)**)		ANINT DNINT	REAL DBLE PREC	REAL DBLE PREC	1
ASIN (**asin(3F)**)		ASIN DASIN	REAL DBLE PREC	REAL DBLE PREC	1
ATAN (**atan(3F)**)		ATAN DATAN	REAL DBLE PREC	REAL DBLE PREC	1
ATAN2 (**atan2(3F)**)		ATAN2 DATAN2	REAL DBLE PREC	REAL DBLE PREC	2
(**mil(3F)**)	*BTEST	*BJTEST *BITEST	INTEGER INTEGER *2	LOGICAL LOGICAL *2	2 [2]
(**ftype(3F)**)	CHAR	-- * -- * --	INTEGER INTEGER *2 INTEGER *1	CHARACTER CHARACTER CHARACTER	1 [3]
CMPLX (**ftype(3F)**)		* -- -- -- -- -- * --	INTEGER *2 INTEGER REAL DBLE PREC COMPLEX DBLE CMPX	COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX	1 or 2 [1]
(**conjg(3F)**)	CONJG	CONJG *DCONJG	COMPLEX DBLE CMPX	COMPLEX DBLE CMPX	1
COS (**cos(3F)**)		COS DCOS CCOS *CDCOS *ZCOS	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	1
COSH (**cosh(3F)**)		COSH DCOSH	REAL DBLE PREC	REAL DBLE PREC	1

Table 9-1. Concurrent Fortran Intrinsic Functions (Cont.)

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
DBLE	*DFLOAT	*DFLOTJ *DFLOTI	INTEGER INTEGER *2	DBLE PREC DBLE PREC	1
(**ftype(3F)**)		-- -- -- *DREAL	REAL DBLE PREC COMPLEX DBLE CMPX	DBLE PREC DBLE PREC DBLE PREC DBLE PREC	
DCMPLX		* -- * -- * -- * -- * -- * --	INTEGER *2 INTEGER REAL DBLE PREC COMPLEX DBLE CMPX	DBLE CMPX DBLE CMPX DBLE CMPX DBLE CMPX DBLE CMPX DBLE CMPX	1 or 2 [1]
(**ftype(3F)**)					
DIM	IDIM	*JIDIM *IIDIM	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**dim(3F)**)		DIM DDIM	REAL DBLE PREC	REAL DBLE PREC	
(**dprod(3F)**)		DPROD	REAL	DBLE PREC	2
EXP		EXP DEXP CEXP *CDEXP *ZEXP	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	1
(**exp(3F)**)					
(**mil(3F)**)	*IAND	*JIAND *IIAND	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**mil(3F)**)	*IBCLR	*JIBCLR *IIBCLR	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**mil(3F)**)	*IBITS	*JIBITS *IIBITS	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**mil(3F)**)	*IBSET	*JIBSET *IIBSET	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**ftype(3F)**)	ICHAR	-- * --	CHARACTER CHARACTER	INTEGER INTEGER *2	1 [2][3]
(**mil(3F)**)	*IEOR	*JIEOR *IIEOR	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**index(3F)**)	INDEX	INDEX * --	CHARACTER CHARACTER	INTEGER INTEGER *2	2 [2]

Table 9-1. Concurrent Fortran Intrinsic Functions (Cont.)

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
INT		-- * --	INTEGER INTEGER *2	INTEGER INTEGER	1
	IFIX	*JIFIX *JINT *IIFIX *IINT	REAL REAL REAL REAL	INTEGER INTEGER INTEGER *2 INTEGER *2	
	IDINT	*JIDINT *IIDINT	DBLE PREC DBLE PREC	INTEGER INTEGER *2	
(**ftype(3F)**)		-- *-- *-- *--	COMPLEX COMPLEX DBLE CMPX DBLE CMPX	INTEGER INTEGER *2 INTEGER INTEGER *2	[2]
(**mil(3F)**)	*IOR	*JIOR *IIOR	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**mil(3F)**)	*ISHFT	*JISHFT *IISHFT	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**mil(3F)**)	*ISHFTC	*JISHFTC *IISHFTC	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
(**len(3F)**)	LEN	-- * --	CHARACTER CHARACTER	INTEGER INTEGER *2	1 [2]
(**strcmp(3F)**)	LGE	-- *--	CHARACTER CHARACTER	LOGICAL LOGICAL *2	2 [2]
(**strcmp(3F)**)	LGT	-- *--	CHARACTER CHARACTER	LOGICAL INTEGER *2	2 [2]
(**strcmp(3F)**)	LLE	-- *--	CHARACTER CHARACTER	LOGICAL INTEGER *2	2 [2]
(**strcmp(3F)**)	LLT	-- *--	CHARACTER CHARACTER	LOGICAL INTEGER *2	2 [2]
LOG (**log(3F)**)		ALOG DLOG CLOG *ZLOG *CDLOG	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	1
LOG10 (**log10(3F)**)		ALOG10 DLOG10 *CLOG10	REAL DBLE PREC COMPLEX	REAL DBLE PREC COMPLEX	1
MAX	MAX0	*JMAX0 *IMAX0	INTEGER INTEGER *2	INTEGER INTEGER *2	>1
(**max(3F)**)		AMAX1 DMAX1	REAL DBLE PREC	REAL DBLE PREC	[2]

Table 9-1. Concurrent Fortran Intrinsic Functions (Cont.)

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
MAX1 (**max(3F)**)		*JMAX1 *IMAX1	REAL REAL	INTEGER INTEGER *2	>1 [2]
MIN	MIN0	*JMIN0 *IMIN0	INTEGER INTEGER *2	INTEGER INTEGER *2	>1
(**min(3F)**)		AMIN1 DMIN1	REAL DBLE PREC	REAL DBLE PREC	[2]
MIN1 (**min(3F)**)		*JMIN1 *IMIN1	REAL REAL	INTEGER INTEGER *2	>1 [2]
MOD (**mod(3F)**)		*IMOD *JMOD MOD AMOD DMOD	INTEGER *2 INTEGER INTEGER REAL DBLE PREC	INTEGER *2 INTEGER INTEGER REAL DBLE PREC	2 [2]
(**nan$(3F)**)		*NAN$ *DNAN$ *CNAN$ *ZNAN$ *CDNAN$	-- -- -- -- --	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	0 [4]
NINT (**round(3F)**)	 IDNINT	NINT *JNINT *ININT IDNINT *JIDNNT *IIDNNT	REAL REAL REAL DBLE PREC DBLE PREC DBLE PREC	INTEGER INTEGER INTEGER *2 INTEGER INTEGER INTEGER *2	1 [2]
(**mil(3F)**) (**bool(3F)**)	*NOT	*JNOT *INOT	INTEGER INTEGER *2	INTEGER INTEGER *2	2 [2]
REAL (**ftype(3F)**)	FLOAT 	*FLOATJ *FLOATI -- SNGL -- *--	INTEGER INTEGER *2 REAL DBLE PREC COMPLEX DBLE CMPX	REAL REAL REAL REAL REAL REAL	1
SIGN (**sign(3F)**)	ISIGN 	*JISIGN *IISIGN SIGN DSIGN	INTEGER INTEGER REAL DBLE PREC	INTEGER INTEGER *2 REAL DBLE PREC	2 [2]

Table 9-1. Concurrent Fortran Intrinsic Functions (Cont.)

Generic Name (man Page)	Limited Generic Name	Specific Name	Data Type of Arguments	Data Type of Results	# of Args
SIN (**sin(3F)**)		SIN DSIN CSIN *ZSIN *CDSIN	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	1
SINH (**sinh(3F)**)		SINH DSINH	REAL DBLE PREC	REAL DBLE PREC	1
SQRT (**sqrt(3F)**)		SQRT DSQRT CSQRT *ZSQRT *CDSQRT	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	REAL DBLE PREC COMPLEX DBLE CMPX DBLE CMPX	1
TAN (**tan(3F)**)		TAN DTAN	REAL DBLE PREC	REAL DBLE PREC	1
TANH (**tanh(3F)**)		TANH DTANH	REAL DBLE PREC	REAL DBLE PREC	1

* Functions or capabilities that are not in the Fortran 77 standard.

NOTE:

[1] `CMPLX` or `DCMPLX` may have one or two arguments. If there is one argument, it may be of type `INTEGER`, `INTEGER*2`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX`. If there are two arguments, they must both be of the same type and may be of type `INTEGER`, `REAL`, or `DOUBLE PRECISION`.

[2] If the **`-i2`** option is specified, then each generic or limited generic intrinsic function which has a default result type of `INTEGER` or `LOGICAL` will instead have a result type of `INTEGER *2` or `LOGICAL *2`, respectively.

[3] `CHAR` produces a character constant of length one at compile time when given an integer constant argument. `ICHAR` produces an integer constant at compile time when given a character constant argument. These may be used in constant expressions, e.g., `PARAMETER` values.

[4] These are constant intrinsics that evaluate to a constant at compile time.

%INT1, %INT2, and %INT4 Integer Size Intrinsics (H)

Concurrent Fortran provides the `%INT1`, `%INT2`, and `%INT4` intrinsics to convert from one integer size to another.

`%INT1 (`*arg*`)` Converts *arg* to `INTEGER*1` which is useful for passing constants to routines that expect `INTEGER*1` arguments.

`%INT2 (`*arg*`)` Converts *arg* to `INTEGER*2` which is useful for passing constants to routines that expect `INTEGER*2` arguments.

`%INT4 (`*arg*`)` Converts *arg* to `INTEGER*4`. This is useful for specifying `INTEGER*4` constants in program units compiled with the **`-i2`** option, which otherwise forces all integer constants to be two bytes.

Example:

```
INTEGER I,J
I = 32769
J = %INT4(32769)
PRINT *,I,J
END
```

Output, when compiled with **`-i2`**, is:

```
-32767 32769
```

%LOG1, %LOG2, and %LOG4 Logical Size Intrinsics (H)

Concurrent Fortran provides the `%LOG1`, `%LOG2` and `%LOG4` intrinsics to convert from one logical size to another.

`%LOG1 (`*arg*`)` Converts *arg* to `LOGICAL*1` which is useful for passing constants to routines that expect `LOGICAL*1` arguments.

`%LOG2 (`*arg*`)` Converts *arg* to `LOGICAL*2` which is useful for passing constants to routines that expect `LOGICAL*2` arguments.

`%LOG4 (`*arg*`)` Converts *arg* to `LOGICAL*4` which is useful for passing constants to routines that expect `LOGICAL*4` arguments.

Example:

```
! .TRUE. and .FALSE. default to LOGICAL*4
CALL LOGSUB (.TRUE., %LOG2(.TRUE.))
END

SUBROUTINE LOGSUB (VAR1, VAR2)
LOGICAL*2 VAR1, VAR2
PRINT *, VAR1, VAR2
RETURN
END
```

Output is:

```
F  T
```

POSIX® P1003.9 Library Functions (H)

The Fortran library includes a full implementation of *POSIX P1003.9 Fortran 77 Language Bindings*, the set of Fortran-accessible routines that provide P1003.1-mandated system services. The **-lposix9** flag should be passed to the linker to access the library. See **posix9(3F)** for more details.

Additional Library Functions (H)

The Fortran library also includes functions which allow access to system functions as well as more powerful bitwise functions. None of these functions are part of standard Fortran 77. Where the man page name is not intuitive from the function name, it has been supplied. The functions the compiler recognizes as special intrinsics are marked with an asterisk *.

Function	Int.	Use
abort	*	terminate Fortran program
access		determine accessibility of a file
adjustl	*	remove leading blanks from character value
adjustr	*	remove trailing blanks from character value
alarm		execute a subroutine after a specified time
and	*	bitwise boolean intrinsic (**bool(3F)**)
besj0		bessel function (**bessel(3F)**)
besj1		bessel function (**bessel(3F)**)
besjn		bessel function (**bessel(3F)**)

Function	Int.	Use
`besy0`		bessel function (**`bessel(3F)`**)
`besy1`		bessel function (**`bessel(3F)`**)
`besyn`		bessel function (**`bessel(3F)`**)
`bufferin`		VOS-like asynchronous I/O (**`bufferio(3F)`**)
`bufferout`		VOS-like asynchronous I/O (**`bufferio(3F)`**)
`cancel`		cancel pending asynchronous I/O (**`bufferio(3F)`**)
`ceiling`	*	largest integer not less than real value
`chdir`		change default directory
`chmod`		change mode of a file
`csignal`		specify Fortran action upon receipt of a system signal
`ctime`		return system time (**`time(3F)`**)
`ctof77str`	*	convert C string to Fortran string
`dbesj0`		bessel function (**`bessel(3F)`**)
`dbesj1`		bessel function (**`bessel(3F)`**)
`dbesjn`		bessel function (**`bessel(3F)`**)
`dbesy0`		bessel function (**`bessel(3F)`**)
`dbesy1`		bessel function (**`bessel(3F)`**)
`dbesyn`		bessel function (**`bessel(3F)`**)
`dffrac`		fractional accuracy of floating point numbers (**`flmin(3F)`**)
`dflmin`		minimum positive floating point value (**`flmin(3F)`**)
`dflmax`		maximum positive floating point value (**`flmin(3F)`**)
`drand`		random number generator (**`rand(3F)`**)
`dtime`		return elapsed execution time (**`etime(3F)`**)
`erf`		error function
`erfc`		complementary error function (**`erf(3F)`**)
`etime`		return elapsed execution time
`even`	*	test for even integer
`exit`	*	terminate process with status
`exponent`	*	exponent portion of Fortran real variable
`f77tocstr`	*	convert Fortran string to C string
`f77tocstr_trim`	*	convert Fortran string to C string (**`f77tocstr(3F)`**)
`fdate`		return date and time in an ASCII string.
`ffrac`		fractional accuracy of floating point numbers (**`flmin(3F)`**)

Function	Int.	Use
`fget`		get a character from a logical unit (**`getc(3F)`**)
`flmin`		minimum positive floating point value
`flmax`		maximum positive floating point value (**`flmin(3F)`**)
`floor`	*	largest integer not greater than real value
`flush`	*	flush output to a logical unit
`fork`		create a copy of this process
`fpecnt`		trap and repair floating point faults (**`trpfpe(3F)`**)
`fputc`		write a character to a Fortran logical unit (**`putc(3F)`**)
`fraction`	*	fractional portion of Fortran real variable
`fseek`		reposition a file on a logical unit
`fstat`		get file status (**`stat(3F)`**)
`ftell`		inquire position of file on a logical unit (**`fseek(3F)`**)
`gerror`		get system error message (**`perror(3F)`**)
`getarg`		return Fortran command-line argument
`getc`		get a character from a logical unit
`getcwd`		get path name of current working directory
`getenv`		return environment variable
`getgid`		get group ID of the caller (**`getuid(3F)`**)
`getlog`		get user's login name
`getpid`		get process ID
`getuid`		get user ID of the caller
`gmtime`		return system time (**`time(3F)`**)
`hostnm`		get name of current host
`iargc`		number of Fortran command-line arguments
`idate`		return date in numerical form
`ierrno`		get system error message (**`perror(3F)`**)
`index`		return location of substring
`inmax`		maximum positive integer value (**`flmin(3F)`**)
`ioinit`		change f77 I/O initialization
`irand`		random number generator (**`rand(3F)`**)
`isatty`		test unit for being a terminal device (**`ttynam(3F)`**)
`itime`		return time in numerical form (**`idate(3F)`**)
`kill`		send a signal to a process
`link`		make a link to an existing file

Function	Int.	Use
`loc`		return the address of an object
`long`		integer object conversion
`lshift`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`lshifta`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`lshiftl`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`lstat`		get file status (**`stat(3F)`**)
`ltime`		return system time (**`time(3F)`**)
`mclock`		return Fortran time accounting
`mvbits`		move bits (**`mil(3F)`**)
`not`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`odd`	*	test for odd integer
`or`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`perror`		get system error message
`putc`		write a character to a Fortran logical unit
`qsort`		quick sort
`rand`		random number generator
`rebuffer`		resize a unit's I/O buffer
`rename`		rename a file
`rshift`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`rshifta`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`rshiftl`	*	bitwise boolean intrinsic (**`bool(3F)`**)
`short`		integer object conversion (**`long(3F)`**)
`signal`		specify Fortran action upon receipt of a system signal
`sleep`		suspend execution for an interval
`srand`		random number generator (**`rand(3F)`**)
`stat`		get file status
`status`		VOS-like asynchronous I/O (**`bufferio(3F)`**)
`symlnk`		make a symbolic link to a file (**`link(3F)`**,**`symlnk(3F)`**)
`system`		issue a shell command from Fortran
`tclose`		tape I/O (**`topen(3F)`**)
`time`		return system time
`topen`		tape I/O
`traper`		trap arithmetic errors
`tread`		tape I/O (**`topen(3F)`**)

Function	Int.	Use
`trewin`		tape I/O (**`topen(3F)`**)
`trpfpe`		trap and repair floating point faults
`tskipf`		tape I/O (**`topen(3F)`**)
`tstate`		tape I/O (**`topen(3F)`**)
`ttynam`		find name of a terminal port
`twrite`		tape I/O (**`topen(3F)`**)
`unlink`		remove a directory entry
`verify`	*	verify character string contents
`wait`		wait for a process a terminate
`xor`	*	bitwise boolean intrinsic (**`bool(3F)`**)

10
Compilation and Execution (H)

10
Compilation and Execution (H)

Compilation

The Concurrent Fortran compiler performs a combination of preprocessing, compiling, assembling, and linking depending on the parameters and options specified. One or more source files, object files, and system libraries may be specified when the compiler is invoked or can be defined implicitly when the compiler is invoked.

There are three Concurrent Fortran products, and each is invoked its own way. The user must have the directory **`/usr/ccs/bin`** in his `PATH` environment variable.

Native PowerPC

The Native PowerPC Fortran compiler runs on a PowerPC based machine running PowerMAX OS and generates code for the same machine.

The compiler is invoked from the shell as follows:

`hf77` **`[`***options arguments files***`]`**

`f77` **`[`***options arguments files***`]`**

where *options* is a list of option specifiers, *arguments* is a list of arguments passed to the link editor, and *files* is a list of source files whose names are suffixed with **`.f`**, **`.F`**, **`.fpp`**, **`.dp`**, **`.r`**, **`.c`**, **`.s`**, **`.o`**, **`.a`** or **`.so`**.

Object files are generated in the ELF format. Debug information is generated in the DWARF Version 2 format.

The **`hf77(1)`** man page provides additional information concerning development environments and options.

Cross Intel to PowerPC

The Cross Intel to PowerPC Fortran compiler runs on an Intel based machine running RedHat or RedHawk Linux, but generates code for a PowerPC based machine running PowerMAX OS.

The compiler is invoked from the shell as follows:

xf77 [*options arguments files***]**

where *options* is a list of option specifiers, *arguments* is a list of arguments passed to the link editor, and *files* is a list of source files whose names are suffixed with **.f**, **.F**, **.fpp**, **.dp**, **.r**, **.c**, **.s**, **.o**, **.a** or **.so**.

Object files are generated in the ELF format. Debug information is generated in the DWARF Version 2 format.

The **xf77(1)** man page provides additional information concerning development environments and options.

Native Intel

The Native Intel Fortran compiler runs on an Intel based machine running RedHat or RedHawk Linux, but generates code for a Pentium-4 based machine running RedHat or RedHawk Linux.

The compiler is invoked from the shell as follows:

cf77 [*options arguments files***]**

where *options* is a list of option specifiers, *arguments* is a list of arguments passed to the link editor, and *files* is a list of source files whose names are suffixed with **.f**, **.F**, **.fpp**, **.dp**, **.r**, **.c**, **.s**, **.o**, **.a** or **.so**.

Object files are generated in the ELF format. Debug information is generated in the DWARF Version 2 format.

The **cf77(1)** man page provides additional information concerning development environments and options.

Multiple Versions

Multiple releases 6.1 and later of the Concurrent Fortran compiler may be installed at the same time. Multiple releases and system defaults are maintained via the c.install and c.release tools, which are also used with the Concurrent C/C++ compilers.

c.install

Install, remove, or modify a release installation

The syntax of the **c.install** command is:

c.install -rel *release* **[***options***]**

The following options are available with the **c.install** command:

Option	Meaning	Function
-arch *arch*	default arch	Set the default PowerMAXOS architecture (nh, moto, synergy, etc.) for PowerPC-targeting compilers.
-cpu *cpu*	default cpu	Set the default Intel processor to optimize code for for Intel-targeting compilers.
-d	default	Mark the selected release installation as the system-wide default
-env *env*	environment	Specify an environment pathname
-gcc_version *version*	gcc version	Set the default gcc version for Intel-targeting compilers to get libraries from.
-f	force	Permit the removal of the last release on the system without confirmation
-H	help	Display syntax and options for this function
-i *path*	install	Install the release located at *path* into the release database (the name is determined from the **-rel** option)
-linux_release *release*	linux release	Set the default Linux release for Intel-targeting compilers to get libraries from.
-m *path*	move	Move the selected release installation to *path*
-osversion *osversion*	default os version	Set the default version of PowerMAX OS being targeted by PowerPC-targeting compilers.
-p	pre-5.1	Mark the selected release isntallation as the default for **cc**, **hc**, **cc++**, and **c++**.
-r	remove	Remove the specified release installation from the release database
-rel *release*	release	Specify a Concurrent C/C++ release (REQUIRED)
-target *cpu*	default cpu	Set the default *cpu* being targeted by PowerPC-targeting compilers. Defaults to the native cpu under PowerMAX OS. A default must be set under PLDE.
-v	verbose	Report changes as they are made

NOTE

Only the System Administrator (or a super user) can invoke **c.install.**

The **-i**, **-m**, and **-r** options may never be used together.

The **c.install** utility is the tool that allows users to register installations with the system's installation database. It may be used to install, move, and remove installations.

When the **-i** option is given, then the structure located at the specified path name is registered with the database as a valid installation. The name of the installation is registered as the release given by the **-rel** option. Therefore, the **-rel** option is required when using the **-i** option to install an installation.

For example, the following command:

```
$ c.install -rel newf77 -d -i /somedir/dir
```

assumes that **/somedir/dir** contains a valid directory structure and "installs" this version of the compilers in the database as **newf77**.

When the **-d** option is used, then **c.install** registers the installation with the database, and also marks the installation as the system-wide default installation (as in the above example).

c.release

Display release installation information

The syntax of the **c.release** command is:

c.release [*options***]**

The following represents the **c.release** options:

Option	Meaning	Function
-arch *arch*	default arch	Set the user's default architecture target (nh, mot, synergy, etc.) for PowerPC-targeting compilers
-cpu cpu	default cpu	Set the user's default cpu for Intel-targeting compilers.
-e	env	Display the path of the selected environment
-env *env*	environment	Specify an environment pathname
-gcc_version *version*	default gcc versiiion	Set the user's default gcc version for Intel-targeting compilers.
-H	help	Display syntax and options for this function
-linux_release *release*	linux release	Set the user's default linux release for Intel-targeting compilers.
-n	name	Display the name of the selected release
-osversion *osversion*	default os ver.	Set the user's default osversion target for PowerPC-targeting compilers.
-p	path	Display the path to the selected release
-q	query	Display the selected environment and release
-r	remove	Remove the default release currently set for the invoking user
-rel *release*	release	Specify a Concurrent C/C++ release

Option	Meaning	Function
-S	system default release	Specify system default release (ignoring user default, PDE_RELEASE environment variable, etc.)
-target *target*	default target	Set user's default target microprocessor for PowerPC-targeting compilers
-U	user default release	Specify the user default release (ignoring system default, PDE RELEASE environment variable, etc.)
-u	user	Set the default release for the invoking user

If invoked without options, **c.release** lists all available release installations on the current host. For example,

```
$ c.release
```

provides output similar to the following:

```
The following compiler releases are available on this machine:

   Name              Lang   Path
   ----              ----   ----
   5.4               CF     /usr/opt/plde-c++-5.4
 * 6.1               CFI    /usr/opt/ccur-compilers-6.1

   Lang: C=PowerPC C++/C     F=PowerPC F77     I=Intel F77

The following PowerPC cross target OS releases are available on this
machine:

   Version           Architecture(s)
   -------           ---------------
 * 4.3               moto, * nh
   5.1               moto, * nh, synergy
   6.1               synergy

   The default target microprocessor is ppc604e

The following Intel gcc libraries are available on this machine:

   Linux Release            Gcc Version(s)
   -------------            -----------
 * i386-redhat-linux      * 3.2
   i386-redhat-linux7       2.96

(*) Designates the defaults
```

Screen 10-1. c.release output

The **-q** option displays the release for the specified environment (or the local environment if no environment is specified). For example,

```
$  c.release -q
```

in a Concurrent C/C++ environment named **test** provides the following output:

```
environment path: /csteam/vir/home/jgj/test
release name: 5.3
release path: /usr/opt/plde-c++-5.3
```

Screen 10-2. c.release -q output

c.release may be invoked with any combination of **-rel** and/or **-env** options. All remaining options are mutually exclusive, and may not be combined in a single invocation of **c.release**.

Compiler Input Files

Types of files are recognized by the suffix that is appended to the file name. The following suffixes identify particular kinds of files:

.f	Fortran source file	**.c**	C source file
.F **.fpp**	Fortran source file run through **cpp(1)** before compilation	**.s**	assembler source file
.dp	Datapool definition source file	**.o**	object file
.r	RATFOR source file	**.a**	archive file
		.so	shared-library file

Some of the released shared libraries contain a mixture of shared-object and static code. This is necessary for proper use of global data. System file names ending with **.so.1** are the shared-object portions of these libraries. Do not specify them separately on the command line; specify the **.so** library name.

RATFOR source files are passed to the **ratfor(1)** preprocessor. C source files are compiled by the appropriatre C compiler. The C compiler also accepts assembler source files and transfers them to the assembler. Object and archive files are handed over to the link editor. The C compiler is passed the **-c**, **-g**, **-D**, **-I**, **-U**, **-v** and **-S** options, if they are specified.

If source files are not specified, the compiler does not perform any action.

Compiler Options

The Concurrent Fortran compiler accepts many options which may be specified separately, each preceded by a "−", or together, with the first option being preceded by a "−". Each of the options is described on the **`hf77(1)`**, **`xf77(1)`**, or **`cf77(1)`** man page as appropriate.

Compiler Arguments

Any additional arguments which do not fall into the previous two sections are taken to be either link-editor option arguments or object programs (typically produced by an earlier run), or libraries of routines. These arguments, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the default name **`a.out`**.

Conditional Compilation

During compilation, a block of source lines may be compiled or skipped depending on flags set with the **`-FLAG`** compiler option. Conditional compilation may be used to produce different versions of a program with each version having different capabilities and characteristics.

The **`-FLAG`** *list* option specifies flag numbers for conditional compilation directives (VOS-style). The *list* may contain zero or more flag numbers (1-23), separated by commas. Do not leave a space between the **`-FLAG`** and the first flag number. Even if flags are not specified, it is still necessary to specify the **`-FLAG`** option to successfully compile any source which contains conditional compilation directives. Note that the listing produced by the compiler (via **`-L`**) shows the conditional compilation directives as comments, and skipped lines as blank lines.

When using conditional compilation, debug or diagnostic statements may be made a permanent part of the program. Fortran statements are placed in the range of one or more conditional compilation blocks. Each of these blocks is controlled by one or more conditional compilation flags. Several conditional compilation flags can be placed in the program to provide different levels of debugging.

A block of source statements is skipped based on a condition established with either of two skip statements, `:SKFS` (or `:skfs`) and `:SKFZ` (or `:skfz`). These statements have the following form:

```
:SKFS f
...
:ESKP
```

or

```
:SKFZ f
...
:ESKP
```

where each *f* is a flag number. The flag number is an unsigned integer constant from 1 through 23. The `:ESKP` (or `:eskp`) command closes a skip block and must be preceded by a matching `:SKFS` or `:SKFZ` statement. A colon in column 1 must precede the statement keyword as shown, or the statement will be treated as part of the Fortran language. Only one flag number may be specified.

Flag numbers establish the skip condition. If a skip condition is true, the range of statements between the skip statement and the next `:ESKP` statement are skipped. For `:SKFS`, the condition is true only if the specified flag in the statement is on (i.e., if the same flag number is specified in the **`-FLAGS`** compile time parameter when the compiler was invoked). For `:SKFZ` the condition is true only if the flag is off (i.e., the flag number was not specified in the **`-FLAGS`** compile time parameter). If a condition is false the statements in the skip block are processed.

Skip blocks may be nested up to 100 levels deep, and a block may be empty. If skip blocks are nested, the first `:ESKP` statement encountered closes the last opened skip block. Each opened skip block must be properly closed before compilation ends. If the skip condition is true for a skip block, any other blocks in the range are skipped regardless of the skip condition.

Examples:

```
:SKFS   1
:SKFZ   20
:SKFZ   2
```

An error in a `:SKFS`, `:SKFZ`, and `:ESKP` statement results in a warning message, and the compiler ignores the statement.

Example:

```
      DIMENSION ID (6)
      ...
      DO 1 I=1,6
      CALL GETCHR (INCHAR, MODE)
:SKFZ 2
      WRITE (*, 100) I, INCHAR, MODE
100   FORMAT (I6, A6, I6)
:ESKP
      IF (MODE .LT.  0) GO TO 2
      ID(I) = INCHAR
1     CONTINUE
      CALL GETCHAR (INCHAR, MODE)
      IF (MODE .GE.  0) CALL ERROR
2     CONTINUE
:SKFZ 1
      WRITE (*, 101) ID, INCHAR
101   FORMAT (6A1, A6)
:ESKP
      END
```

The preceding example obtains a Fortran identifier in the one dimensional array `ID` and its delimiter `INCHAR`. Subroutine `GETCHR` returns the next character from the input stream and its `MODE`. The two conditional compilation blocks produce two levels of debugging information. The first block is controlled by flag number 2. If this flag is on at compile time, the first block is compiled thus causing `I`, `INCHAR`, and `MODE` to be printed during each iteration of the `DO` loop. If flag number two was not set at compile time, this output is not produced during execution. The second block is controlled by flag number 1. If this flag was set at compile time, the second block is compiled. This causes the identifier `ID` and the delimiter `INCHAR` to be printed. If flag number 1 was not set at compile time, this output will not be produced during execution.

Environment Variables

This section describes shell environment variables that are significant when compiling or executing Fortran programs. See also the man page.

f77_dump_flag

The environment variable, `f77_dump_flag` controls core dumps for programs compiled with Concurrent Fortran. A normal Fortran program does not dump core if an internal error occurs unless the first character of `f77_dump_flag` is a "`y`". However, if you have compiled and linked your program using the **`-g`** debug option and the first character of `f77_dump_flag` is not an "`n`", the program dumps core by default if an internal error occurs.

fortunit

If no file name is supplied when opening the unit and there is no DEFAULTFILE= specifier, the environment variable fortunit if defined contains the file name to be used. If the environment variable is not set, the file name defaults to **fort.unit**.

F77INCLPATH

F77INCLPATH is a shell environment variable that can be set to contain a list of directories, separated by a colon, that the compiler uses to search for Fortran INCLUDE files. It is similar in format to the PATH environment variable.

Example:

```
F77INCLPATH="/usr/local/fort/include:/usr/local/simul/include"
```

Refer to "Include Lines (H)" on page 2-8.

LD_BIND_NOW, LD_LIBRARY_PATH, LD_RUN_PATH

These variables affect the behavior of **ld(1)** and the dynamic linker. See the "Link Editing" Chapter of the *Compilation Systems Volume 1 (Tools)* manual.

STATIC_LINK

Using STATIC_LINK is an alternative to specifying **-Zlink=static**. The presence of this variable in the environment is all that is needed to turn on static linking; the value of the variable is not interpreted. The following sequence in the Bourne or Korn shells sets this variable.

```
$ STATIC_LINK=yes
$ export STATIC_LINK
```

To explicitly remove STATIC_LINK from the environment in the Bourne or Korn shells:

```
$ unset STATIC_LINK
```

TARGET_ARCH

Using TARGET_ARCH is an alternative to specifying **-Qtarget=**. Any of the valid target values may be used. Additional TARGET_ARCH values may apply; see the man page.

Option	TARGET_ARCH Value
-Qtarget=ppc604	**PowerPC 604™**
-Qtarget=ppc	**PowerPC**

Linking Mixed-Language Programs

Normally, invocation of the **hf77(1)** compiler driver, causes it to automatically invoke the **ld(1)** link editor with appropriate options and libraries; **ld(1)** in turn creates an executable program. Creating a program from a mixture of C and Fortran modules sometimes requires altering this process.

The following text discusses the default **ld(1)** invocation.

```
/bin/ld -t [-u _MAIN__] /usr/ccs/lib/crt0.o \
    [/usr/ccs/lib/vax.o] [/usr/ccs/lib/unsint1.o] \
    object.o [...] -L/usr/ccs/lib[-lg] -lhU77 -lhF77 \
    -lhI77 -lm -lc -o a.out
```

/bin/ld The full path name of the **ld(1)** link editor.

-t The option that turns off warnings about common blocks being accidentally defined with different sizes in different routines.

-u _MAIN__ The option that enters the main routine as an undefined symbol in the symbol table; this forces the main routine to be loaded. This option is necessary only if the main routine is a Fortran module in a user library.

/usr/ccs/lib/crt0.o
The object file that calls main. The Fortran libraries contain a routine called main that does some setup for Fortran I/O and signal handling. The version of main in the Fortran libraries then call _MAIN__; this is the name of the main routine generated by the Fortran compiler.

/usr/ccs/lib/vax.o
The object file that causes the Fortran libraries, **hU77**, **hF77**, and **hI77**, to interpret and return LOGICAL values consistent with VAX implementations. This object file is necessary only when compiling Fortran code with the **-V** or **-VAX** options.

/usr/ccs/lib/unsint1.o
The object file that causes the Fortran libraries to interpret

	`INTEGER*1` variables as unsigned. This object file is necessary only when compiling Fortran code with the **`-uns_int1`** option.
`object.o` [...]	The names of user-specified object files and libraries.
`-L/usr/ccs/lib`	Instructs **`ld(1)`** to look for libraries in the directory **`/usr/ccs/lib`**.
`-lg`	The debugging library. This library is necessary only when compiling with the **`-g`** option.
`-lhU77 -lhF77 -lhI77`	The Fortran libraries. So that all module names are resolved, it is sometimes necessary to repeat these library names. One instance when this may be necessary is when the main routine is a C module.
`-lm`	The math library. Intrinsics in the Fortran libraries contain references to this library.
`-lc`	The C library. The Fortran libraries require the inclusion of the C library.
`-o a.out`	The name of the executable, by default **`a.out`**.

A
Array Storage

Arrays are stored linearly in main memory. Table A-1 shows how arrays are stored internally. Note that the leftmost subscript varies most rapidly.

Elements of array `A(3,3,2)` are stored as follows**:**

Table A-1. Array Storage

Memory Location	Array Element
1st	`A(1,1,1)`
2nd	`A(2,1,1)`
3rd	`A(3,1,1)`
4th	`A(1,2,1)`
5th	`A(2,2,1)`
6th	`A(3,2,1)`
7th	`A(1,3,1)`
8th	`A(2,3,1)`
9th	`A(3,3,1)`
10th	`A(1,1,2)`
11th	`A(2,1,2)`
12th	`A(3,1,2)`
13th	`A(1,2,2)`
14th	`A(2,2,2)`
15th	`A(3,2,2)`
16th	`A(1,3,2)`
17th	`A(2,3,2)`
18th	`A(3,3,2)`

B
Non-Standard Extensions to Fortran 77 (H)

Non–standard features of the **hf77** compiler are described in this appendix.

The **hf77** compiler supports many non–standard extensions in the VAX/VMS Fortran compiler, as well as some **f77** extensions and VOS **sauf77** extensions. Some extensions have not been included because they require system–level support which is not provided. Other extensions which have been implemented may behave differently from those on a VAX because of system differences. As always, caution should be exercised before using these extensions.

Ampersand (`&`) is accepted as a continuation symbol when found in column 1. Also the compiler accepts a variable length input format.

1. More than 19 continuation lines are supported. There is no effective limit to the number of continuation lines that may be provided.
2. Debugging statements may be included as part of the source file for a Fortran program by using `D` in column 1. If the **`-D`** option is selected these statements become part of the program, otherwise these lines are interpreted as comments.
3. A tab character in one of the first six positions of a line signals the end of the statement number and continuation part of the line. The remaining characters form the body of the line. If a tab appears elsewhere on a line, the compiler treats the tab as a blank space.
4. End of line comments are supported. An exclamation point (`!`) terminates the source line and anything that follows it is interpreted as a comment.
5. The `NAME` statement is supported by the compiler. It is equivalent to the `PROGRAM` statement. See “PROGRAM and NAME Statements” on page 2-10 for syntax of this statement.
6. The `INCLUDE` statement is supported by this compiler and allows the user to include another source file as part of the source for the current program unit.
7. Symbolic names can be up to 1023 characters long and they may include underscores and dollar signs as well as lower case letters.
8. There is support for additional data types. These data types are described in greater detail in “Data Types” on page 2-12.

```
BYTE
COMPLEX*16
DOUBLE COMPLEX
INTEGER*1
INTEGER*2
LOGICAL*1
LOGICAL*2
REAL*8
```

9. An extension to the `IMPLICIT` statement allows the user to reset implicit types by using the `IMPLICIT NONE` or `IMPLICIT UNDEFINED` specification statement. See "IMPLICIT Statement" on page 4-27 for information about this feature.

10. This implementation allows `DATA` statements to appear anywhere within the source of a program unit. More information on `DATA` statements can be found in "DATA Statement" on page 4-12.

11. Data initialization clauses are accepted in the declaration statements. See Chapter 4 for information about the syntax declaration statements using initialization.

12. The `DATAPOOL` global data mechanism is supported. Refer to "DATAPOOL Statement (H)" on page 4-16 for more information.

13. The compiler allows the `AUTOMATIC` storage class for variables to be allocated on the stack. The `STATIC` keyword is also accepted. See "AUTOMATIC Statement (H)" on page 4-7 and "STATIC Statement (H)" on page 4-37 for details.

14. The `CEXTERNAL` keyword and a number of C/Fortran string conversion intrinsics are provided. See "CEXTERNAL Statement (H)" on page 4-8 and "CEXTERNAL Declaration (H)" on page 8-31.

15. The `POINTER` statement declares a block of variables to be accessed relative to a base address. For details, see "POINTER Statement (H)" on page 4-34.

16. The compiler permits single subscripts in `EQUIVALENCE` statements and assumes that all missing subscripts are equal to one. The compiler prints a warning message for each incomplete subscript.

17. Non–integer subscript expressions are accepted by the compiler.

18. Real–valued parameters defined in `PARAMETER` statements may be used to dimension arrays.

19. The `NAN$` constant intrinsic is available for flagging uninitialized Real variables with an IEEE–specified signal. For more information, see "Real Data" on page 2-21.

20. Null character strings are supported by the compiler.

21. Binary, octal, and hexadecimal constants may be specified with a syntax similar to the Hollerith constant syntax, as well as with the usual single–quote and double–quotes notation. The new syntax allows users to use constants of unspecified length. Refer to "Binary Data (H)" on page 2-19, "Octal Data (H)" on page 2-18, and "Hexadecimal Data (H)" on page 2-17 for more information.

22. Hollerith data can be stored as the value of numeric or logical variables and array elements.

23. New operators, `.XOR.`, `.ROTAT.`, and `.SHIFT.` have been added. See ".SHIFT. and .ROTAT. Integer Operators (H)" on page 3-12 and "Logical Expressions" on page 3-17 for details.

24. The logical operators now may be used with integer operands to perform bitwise logical computations.

25. Logical and integer operators and operands can be mixed. This allows for a more flexible use of these data types.

26. Multiple assignment statements are supported. Refer to "Multiple Assignment Statements (H)" on page 3-31 for more information.

27. Some additional control constructs are supported. These constructs are described in more detail in Chapter 5.

`DO`	`FOR`
`DO WHILE`	`LOOP`
`DO UNTIL`	`WHILE`
`END DO`	`SELECT CASE`

28. Namelist directed input/output is supported by this compiler. This allows symbolic I/O variable selection at run time, making a program more flexible. This is described in "NAMELIST Statement (H)" on page 4-30 and Chapter 6.

29. List–directed internal `READ` and `WRITE` statements are permitted by the compiler. See "List-Directed I/O Statements" on page 6-7 for details.

30. `ACCEPT` and `TYPE` statements are supported. `ACCEPT` is functionally similar to `READ`, and `TYPE` is functionally similar to `WRITE` and `PRINT`. See Chapter 6 for information about `READ` and `WRITE` statements.

31. The *u*`'`*r* direct access format in `READ` and `WRITE` statements is also supported by this compiler.

32. Additional `OPEN` and `INQUIRE` keywords are accepted by the compiler. Some of these keyword perform no function because they have no meaning in terms of the file structure on this system. Keywords in the following list that have an asterisk beside them are accepted but ignored; the compiler issues a warning when they are used.

	`ASSOCIATEVARIABLE`		`NAME`
*	`BLOCKSIZE`	*	`NOSPANBLOCKS`
*	`BUFFERCOUNT`	*	`ORGANIZATION`
	`CARRIAGECONTROL`		`READONLY`
	`DEFAULTFILE`	*	`RECORDSIZE`
	`DISP`		`RECORDTYPE`
	`DISPOSE`	*	`SHARED`
*	`EXTENDSIZE`		`TYPE`
*	`INITIALSIZE`	*	`USEROPEN`
	`MAXREC`		

33. The `O` and `Z` format descriptors for octal and hexadecimal output are supported. The `Q` and `$` descriptors are supported to provide greater control over formatted input and output.

34. INTERNAL subroutines and functions are supported. See "INTERNAL Subprograms (H)" on page 8-20 for details.

35. Three special argument list intrinsic functions are supported by the compiler: `%VAL`, `%REF`, `%LOC`. `%VAL` passes any argument by value rather than reference; `%REF` undoes any `%VAL` action; and `%LOC` generates the address of a variable, resulting in the address of the variable being passed. See "%VAL, %LOC, and %REF Argument List Intrinsics (H)" on page 8-5 for a description of these special functions.

36. The `%INT1`, `%INT2`, and `%INT4` intrinsics, which provide a simple method of converting between integer variables and constants of different sizes, are supported. The `%LOG1`, `%LOG2`, and `%LOG4` intrinsics, which provide a simple method of converting between logical variables and constants of different sizes, are also supported. They are all especially useful in argument lists. See "%INT1, %INT2, and %INT4 Integer Size Intrinsics (H)" on page 9-9 and "%LOG1, %LOG2, and %LOG4 Logical Size Intrinsics (H)" on page 9-9 for more information.

37. There are several additional intrinsic functions which are available in the compiler. See Chapter 9 for a complete listing of all supported intrinsic functions.

38. Conditional compilation flags may be used to produce different versions of a program with each version having different capabilities and characteristics. Even if no flags are going to be specified, it is still necessary to specify the **`-FLAG`** option to successfully compile any source which contains conditional compilation directives. Example:

```
      INTEGER I, SUM
      SUM =20
      DO 1 I=1,10
      SUM = SUM + I
:SKFS 2
      WRITE (*, 100) I, SUM
100   FORMAT (I2, I3)
:ESKP
1     CONTINUE
      WRITE (*, 101) SUM
101   FORMAT (I3)
      END
```

The block is controlled by flag number 2. If this flag is on, then the skip condition is true. The range of statements between the skip statement and the next `:ESKP` statement are skipped. If this flag is off, then all statements are compiled and executed.

C
Incompatibilities with Fortran 66

Incompatibilities of Fortran 77 with Fortran 66 are identified in this appendix.

1. The method of specifying alternate return points is significantly different in Fortran 77. In particular, asterisks in the argument list identify alternate return dummy arguments, and an integer expression on the `RETURN` statement identifies which argument is to be used as the return point. In Fortran 66, the corresponding dummy arguments where integer variables, and whose name appeared on the `RETURN` statement.

   Note that the Fortran 77 return is similar to a computed `GO TO`, whereas the Fortran 66 version is similar to an assigned `GO TO`.

2. A line with all blank characters in columns 1 through 72 is a comment line and is not interpreted as the initial line of a statement.

3. In Fortran 77, a subscript value for a multi-dimensioned array cannot exceed its corresponding upper bound declaration. For example, `A(11,1)` is not permitted for the array `A(10,5)`.

4. A warning is issued when a symbolic name is explicitly typed more than once in the same program unit.

5. Data cannot be read directly into an `H` descriptor field defined in an input `FORMAT` statement.

6. A simple input or output list cannot be enclosed in parentheses; parentheses may be present only as part of an implied-`DO`. This restriction avoids any ambiguity if complex constants appear in an I/O list.

7. An entity that is associated with an entity in an input list becomes defined when the input list entity becomes defined and not after the input statement completes execution.

8. Fortran 77 always writes a plus (+) or minus (−) sign prior to the exponent field for an `E` or `D` output field.

9. The class of functions in Fortran 77 called intrinsic includes the basic external function class of Fortran 66. An intrinsic function name used as an actual argument must appear in an `INTRINSIC` statement rather than an `EXTERNAL` statement.

10. The data type of an intrinsic function cannot be changed by using the name in an explicit type statement.

11. Intrinsic function names have been added to Fortran 77 that could conflict with user subprogram names. See Table 9-1 for a list of the intrinsic functions.

12. The range of the arguments and the result of intrinsic functions could be different in Fortran 77 than they were in Fortran 66. See Chapter 9 for information on the intrinsic functions.

13. The `FIND` statement is accepted but has no effect in Concurrent Fortran 77.

14. The `RECUR` statement is not supported in Concurrent Fortran 77.

15. Fortran 66 allows a negative value for the width (number of positions) for format specification `X`. Fortran 77 and ANSI standards prohibit negative values.

16. The letter specification in an `IMPLICIT` statement may not be omitted in Fortran 77.

17. A Fortran 77 external function does not set the condition code register to reflect the value returned by the function.

18. The syntax used to specify dummy arguments for an `ENTRY` point is significantly different. Fortran 66 requires specification of all dummy arguments within the first statement of the program unit. Fortran 77 requires dummy arguments to be specified on the `ENTRY` statement.

19. Fortran 77 does not support the `DEFINE FILE` statement.

Glossary

This glossary defines terms used in the documentation. Terms in *italics* are defined here.

access method

A *file*'s organization which controls the order in which *records* of the file are retrieved or written. It is dependent on the properties of the storage medium. Fortran permits both *sequential access* and *direct access* files.

actual argument

A *constant*, *expression* or *symbolic name* appearing in the argument list of a *function* reference or *subroutine* call. Actual arguments are the entities that are specified in parentheses after the function or subroutine name, when the function or subroutine is invoked in a program unit. Actual arguments match the order, *data type*, and number of *dummy arguments* defined in the initial statement or entry point of the *subprogram*.

argument

See *actual argument* and *dummy argument*.

arithmetic assignment statement

An *assignment statement* that assigns a *numeric*, *logical*, *character*, or *Hollerith* value to a numeric *variable* or *array element name*.

arithmetic expression

An *expression* that produces numeric values that are used in an arithmetic context, such as in an *arithmetic assignment statement*. Evaluation of an arithmetic expression results in a single numeric value.

array

A sequence of *variables* that have the same *symbolic name* and *data type*.

array element name

A reference to a single *array* element consisting of the array name and a subscript for each dimension of the array.

array assignment statement

An assignment statement that assigns a value to every visible element of an array.

assignment statement

An *executable statement* that assigns a value to a *variable* or an *array* element. The four kinds of assignment statements are *arithmetic*, *character*, *logical*, and statement label assignments (i.e. the `ASSIGN` statement.).

binary operator

An *operator* that appears between two *operands*.

bitwise operation

A *logical* operation that usually generates faster code than a *short-circuit operation*. This operation does not require testing-and-branching which are time-consuming on modern RISC architectures.

blank common

A common block with no name specified after the keyword `COMMON`.

built-in function

See *intrinsic function*.

CHARACTER

The *data type* of a non-numeric, non-logical value consisting of a fixed-length string of ASCII characters.

character assignment statement

An *assignment statement* that assigns a *character string* value to a *variable* name or *array* element name.

character constant expression

A *constant expression* in which each primary is a character constant, the *symbolic name* of a character constant, or a character constant expression enclosed within parentheses. The concatenation *operator* (`//`) is allowed for *character string concatenation*. The *intrinsic function*, `CHAR`, when given an *integer* constant argument, generates a character constant of length one at compile time; it may be used to form a character constant expression. Character constant expressions are the only character expressions that may be used to set character `PARAMETER` values.

character expression

An *expression* that results in a value that is a *character string*.

character string

A data value of fixed length that includes letters, numbers, or special characters.

character string concatenation

An *operation* in which two or more *character string operands* are combined into a single string by appending the right operand to the left operand.

character substring

A contiguous portion of a *character string* value.

collating sequence

The arrangement of characters in an order such that when two characters are compared numerically, one character is either less than, equal to, or greater than the other.

COMPLEX

The *data type* of a pair of optionally signed *real* numbers. The first is the real portion of a complex number, and the second is the imaginary portion of a complex number.

compound arithmetic expression

An *arithmetic expression* that consists of two or more numeric *operands*, connected by arithmetic *operators*, appearing in an arithmetic context.

compound character expression

A *character expression* that consists of two or more *operands* connected by the character concatenation operator (`//`), appearing in a character context.

constant

A syntactical element with a fixed value that is not subject to change. A constant can be a signed or unsigned number, a *logical* value, a literal *character string*, or a *Hollerith* string.

constant expression

A *simple* or *compound arithmetic expression* that contains only *constants*. An *expression* used in a specification statement (e.g., as an *array* declarator in a `COMMON`, `DATAPOOL`, `AUTOMATIC`, `STATIC`, `DIMENSION`, or explicit type statement) must be an *integer* constant expression; i.e., *variable* names, array element names, or *function* references are not permitted, and the resulting *data type* of the expression must be an integer.

continuation line

A line after the initial line that holds part of a statement.

current record

The *record* at which a *file* is currently positioned for reading or writing. If the file is positioned at the *initial* or *terminal point*, there is no current record.

data type

The kind of data a *symbolic name* represents, as well as the storage limits and, for numeric quantities, the precision. The maximum and minimum size for data of a particular type and the accuracy limits on the data depend on the storage unit sizes defined for the data type. Data types include *character*, *integer*, *logical*, *real*, and *complex*.

direct access

The file *access method* in which uniquely numbered, fixed-length *records* may be written or read in any order.

DOUBLE COMPLEX

The *data type* of a pair of double precision numbers. The first is the real portion of a *double precision* complex number, and the second is the imaginary portion of a double precision complex number.

DOUBLE PRECISION

The *data type* of an optionally signed *real* number that is stored with a greater degree of accuracy than a single precision real number. Double precision numbers are stored in eight-byte form (with a 55-bit mantissa).

dummy argument

A *constant*, *expression* or *symbolic name* appearing in the argument list of a *subprogram* definition or entry statement. Dummy arguments represent the correct order, data type, and total number of *actual arguments* that are passed to the subprogram when the subprogram is invoked. All dummy arguments must be *variable* names or unsubscripted *array* names (i.e., no constants, expressions, or subscripted array names are permitted). The length and *data type* of each dummy argument are defined explicitly in an explicit type statement in the subprogram or implicitly by the first character of the name.

executable statement

A statement that specifies actions to be performed and is identified by Fortran *key-words*.

expression

An *operand* and possibly some *operators* that represent a value. The Fortran language permits *arithmetic*, *character*, *relational*, and *logical expressions*.

external file

A *file* read from or written to an external device (e.g., a line printer, magnetic tape drive, magnetic disk drive, etc.). An external file can be empty.

external function

A *function* supplied by the user.

file

A collection of *records*. It is external or internal.

flow-of-control context

The context in which a value is interpreted within a decision or control situation--for example, the *logical expression* of `IF` and `WHILE` statements.

format specification

An entity that defines the size of input and output fields, what type of data is being read or written (*numeric*, *character*, *Hollerith*, or *logical*) and how the data is to be edited in *formatted I/O* statements.

formatted I/O

A form of I/O that edits *records* on both input and output and requires a *format specification* in the I/O statement.

free-format I/O

See *list-directed I/O*.

function

A *subprogram* with a defined *data type*. It is invoked when its name, followed by any arguments in parentheses, is referenced or used in an *expression* where the value of the function is needed. It includes *intrinsic functions*, *statement functions*, and *external functions*.

Hollerith constant

A non-empty string of characters. Hollerith constants may appear only in a `DATA` statement, a `FORMAT` statement, and during assignment to numeric variables.

host subprogram

A *subprogram* that contains an *internal subprogram*.

initial point

The position just before the first *record* of a *file*.

INTEGER

The *data type* of an optionally signed whole number containing no fractional portion and no decimal point. Integers are stored in one-byte, two-byte, or four-byte form.

internal file

A character *variable*, character *array*, character *array element name*, or *character substring* used to transfer data from one location in memory to another location in memory and is used to convert data from one form to another (e.g., from numeric to character form).

internal subprogram

A *subprogram* defined within and visible only within the body of a *host subprogram*. It must not contain other internal subprograms. It may have arguments and may make *uplevel references* to variables that are visible within the host subprogram without additional declarations.

intrinsic function

A *function* supplied with the Fortran compiler.

keyword

A syntactical element that identifies a Fortran statement (e.g., `DO`, `FORMAT`, `IF`, `STOP`, etc.) or is a separator in a Fortran statement (e.g., `THEN`, etc.). A keyword identifying a Fortran statement is usually the first word of the statement. Whether a particular sequence of characters represents a keyword or a *symbolic name* is implied by context. Fortran keywords have no abbreviated forms.

list-directed I/O

A form of *sequential access* I/O that permits data on one or more *records* to be read or written until all items in an input or output list are satisfied. List-directed I/O statements do not require a `FORMAT` statement. Data editing is performed by the compiler and cannot be changed by the user.

LOGICAL

The *data type* of a value representing the logical or boolean concept true or false. Logicals are stored in one-byte, two-byte, or four-byte form.

logical assignment statement

An *assignment statement* that assigns a *logical* value of `.TRUE.` or `.FALSE.` to a *variable* name or *array element name*.

logical expression

An *expression* that expresses a *logical* computation that produces a single result of type logical with a value of true or false. A logical expression contains a single logical *operand*, or two or more logical *operands* connected by logical *operators*.

mixed mode arithmetic expression

An *arithmetic expression* that contains a mixture of *operands* with different numeric *data types*. In a mixed mode expression, as each subexpression within the expression is evaluated, the resulting mode is that of the operand with the highest ranking data type.

multiple assignment statement

An assignment statement that assigns a value to more than one *variable*, *array element name*, or *array.*

namelist-directed I/O

A form of sequential I/O that reads or writes a series of data *records* from a specified *unit*. Each series begins with a header block and ends with a terminating block. Each record may contain a list of *variable*-name/value specifications, causing the variable to assume the specified value. Variables that may be updated in this manner are defined in a `NAMELIST` statement.

next record

The *record* just after the *current record*. If the last record is the current record or if the *file* is positioned at the *terminal point*, there is no next record. If a file is positioned at its *initial point*, the first record is the next record.

non-executable statement

A statement that is a directive to the compiler. It describes the characteristics and arrangement of input data, sets the initial values for *variables* and array elements, indicates input and output editing information, defines and classifies *program units*, or specifies entry points in *subprograms*.

operand

A *variable*, *array element name*, or *constant* that is acted upon by an *operator.*

operator

A syntactical element that acts upon *operands* that are *symbolic names* and *constants*. A mnemonic or special character used in Fortran *expressions* to perform arithmetic computations, to concatenate character strings, or to perform relational and logical comparisons.

parameter

1) A constant with a *symbolic name* specified with the `PARAMETER` statement. 2) See *argument*.

preceding record

The *record* just before the *current record*. If the first record is the current record or if the *file* is positioned at its *initial point*, there is no preceding record. If the file is positioned at its *terminal point*, the last record of the file is the preceding record.

program unit

A main program or a *subprogram*.

REAL

The *data type* of an optionally signed real number containing an *integer* part or a fractional part, or both. Numbers of type real are stored in four-byte or eight-byte form.

record

A sequence of one or more data values transferred by an I/O statement. The length of a record is the number of bytes (i.e., characters) in the record.

reference

A use of a *function* in an *expression* where the value of the function is needed.

relational expression

An *expression* that compares the resultant values of two *arithmetic expressions* or of two *character expressions*. A mixture of character and arithmetic *operands* is not permitted. The arithmetic or character expression is a *simple* or *compound expression*.

scope

The extent of visibility. *Symbolic names* with global scope represent one entity in all *program units* throughout the entire *source program*. Symbolic names with local scope represent one entity in a single program unit, but the same name can represent another entity in another program unit.

sequential access

The file *access method* in which variably lengthed *records* are written one after the other and are read in the order in which they were written. The records are either all *formatted* or all *unformatted*.

short-circuit operation

An `.AND.` or `.OR.` operation that avoids evaluating both *operands* when evaluation of the first operand determines the final result. Short-circuiting requires testing-and-branching; the overhead of short-circuiting may be prohibitive on modern RISC architectures if the cost of evaluating each operand is small.

simple arithmetic expression

An *arithmetic expression* that consists of one *operand*. The operand can be a numeric *constant* or the *symbolic name* of a constant, a numeric *variable* name, a numeric *array element name*, or a numeric *function* reference.

simple character expression

A *character expression* that consists of a single *operand*: one character *constant* or the *symbolic name* of a character constant, one character *variable*, one character *array element name*, or one character *function* reference.

source program

One or more *program units* containing Fortran statements and optional comments.

specification statement

A statement that defines the *data types* of *symbolic names*, declares the storage requirements of *variables* and *arrays*, defines initial values for variables and array elements, specifies the dimensions of arrays, defines program entities that are known globally throughout the *source program* among all program units, and gives symbolic names to arithmetic, character, and logical constants. Specification statements are *non-executable* and have no effect during the execution of the *source program*.

specifier

An I/O control list value that must be specified in a particular position in the list of control information or denotes a specification in keyword form *keyword=value.* See also *format specification*.

statement function

A one-statement *function* that is written by the user and that pertains only to the *program unit* in which it is defined.

statement label

A syntactical element that identifies the initial line of a statement. It is one to five decimal digits appearing anywhere in columns 1 through 5 of the initial line of a fixed format statement. Zero is not a valid statement label. Statement labels provide a point of reference so that another statement within the *program unit* can refer to the labeled statement.

storage association

When two or more storage sequences share one or more memory locations. Storage sequences are totally associated if they share the same memory locations. Storage sequences are partially associated if they share one or more, but not all, memory locations.

storage sequence

A collection of contiguous memory locations. A `COMMON` declaration, and the individual elements of an array, form storage sequences.

subprogram

A *program unit*, independent of the main program, that is either written by the user or supplied with the Fortran compiler. Subprograms are classified as *functions*, *subroutines*, or block data.

subroutine

A *subprogram* that is invoked with a `CALL` statement and returns zero or more values for use by the calling *program unit*.

substring

A single character in a *character string*, a subset of contiguous characters in a character string, or the entire character string in duplicate.

symbolic name

A syntactical element that identifies a user-defined entity, such as a *variable*, named constant, *subprogram* name, or `COMMON` block name. It consists of 1 to 1023 letters, digits, dollar signs, or underscores, the first of which must be a letter.

terminal point

The position just after the last *record* of a *file*.

truth value

The value of a *logical expression*. Depending on the implementation, one or more values may represent a true truth value or a false truth value.

unary operator

An *operator* that appears before its one *operand*.

unformatted I/O

A form of I/O that reads or writes *records* that consist of binary data; each record is a string of binary digits. No data translation or editing is performed; thus, a *format specification* is not specified.

unit

A number used in I/O statements that is associated with a *file* name.

uplevel reference

A *reference* to a *host subprogram*'s variable from one of its *internal subprograms*.

user-defined function

See *external function*.

variable

A *symbolic name* with a specific *data type*. The name of a variable refers to a location in memory where a data value is stored or is to be stored. Referencing a variable means that the variable is used in a context where its value is needed.

value-producing context

The right-hand side of an *assignment statement* or an *operand* of an arithmetic *operator.*

Index

Symbols

A

D

E

K

L

M

N

O

P

R

S

T

U

V

W

Z

Spine for 1.5” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up