HP 3000 Computer Systems

HP FORTRAN 77/iX Reference

HEWLETT PACKARD

HP Part No. 31501-90010
Printed in U.S.A. June 1992

E0692
Fourth Edition

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	**Date**	**Software Version**
First Edition	October 1988	31501A.02.00
Second Edition	October 1989	31501A.03.05
Third Edition	December 1990	31501A.04.11
Fourth Edition	June 1992	31501A.04.31

Preface

This is the reference manual for the HP FORTRAN 77 programming language as it is implemented on the MPE/iX operating system. This manual assumes that the reader has been trained in the FORTRAN language and knows FORTRAN programming techniques.

For MPE/iX only, this manual replaces the following two manuals:

- *HP FORTRAN 77 Reference Manual* (5957-4685)
- *HP FORTRAN 77/XL Reference Manual Supplement* (31501-90001)

The information previously contained in the reference manual and supplement is now contained in this manual.

Chapter Summary

This manual is organized into the following chapters:

chapter 1	Introduces the vocabulary and structure of HP FORTRAN 77. It includes an example source file.
chapter 2	Describes fundamental parts of the HP FORTRAN 77 language. It identifies the character set, defines keywords and symbolic names, and describes data types.
chapter 3	Describes each statement in the HP FORTRAN 77 language.
chapter 4	Describes the HP FORTRAN 77 input/output statements in detail. It defines all format descriptors and includes examples of their use.
chapter 5	Describes file formats and related I/O topics.
chapter 6	Describes some fundamentals of using HP FORTRAN 77 under this operating system, such as how to invoke the compiler and linker.
chapter 7	Provides descriptions of the compiler directives available in HP FORTRAN 77 under this operating system.
chapter 8	Describes the interface between HP FORTRAN 77 and other languages, as well as with the operating system.
chapter 9	Describes facilities in HP FORTRAN 77 under this operating system that are useful for run-time error management.

chapter 10	Describes how HP FORTRAN 77 data types are formatted in memory.
appendix A	Lists and describes compile-time error messages, compiler warnings, ANSI warnings, and run-time errors.
appendix B	Lists the HP FORTRAN 77 intrinsic functions.
appendix C	Compares HP FORTRAN 77 with the ANSI 77 standard, FORTRAN 66/V, and FORTRAN 7X.
appendix D	Presents HP's implementation of the ASCII character set.
appendix E	Lists a program using indexed sequential access (ISAM).

Additional Documentation

The following manuals are referenced in this manual:

- *HP FORTRAN 77/iX Programmer's Guide* (31501-90011)
- *HP FORTRAN 77/iX Migration Guide* (31501-90004)
- *HP Link Editor/iX Reference Manual* (32650-90030)
- *HP Pascal/iX Reference Manual* (31502-90001)
- *HP Pascal/iX Programmer's Guide* (31502-90002)
- *Trap Handling Programmer's Guide* (32650-90026)
- *Compiler Library/XL Reference Manual* (32650-90029)
- *MPE/iX Intrinsics Reference Manual* (32650-90028)
- *Native Language Programmer's Guide* (32650-90022)
- *HP Symbolic Debugger/iX User's Guide* (31508-90003)

The *HP FORTRAN 77/iX Programmer's Guide* contains detailed discussions of selected HP FORTRAN 77 topics.

The *HP FORTRAN 77/iX Migration Guide* contains information on how to run FORTRAN 66/V and HP FORTRAN 77/V programs on the MPE/iX operating system and how to convert them to HP FORTRAN 77/iX programs.

Conventions

`UPPERCASE`	In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example: `COMMAND` can be entered as any of the following: `command` `Command` `COMMAND` It cannot, however, be entered as: `comm` `com_mand` `comamnd`
italics	In a syntax statement or an example, a word in italics represents an optional parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file: `COMMAND` *filename*
punctuation	In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered: `(`*filename*`):(`*filename*`)`
underlining	Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt: `Do you want to continue? >>  yes`

`{ }`	In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`: $\texttt{COMMAND} \left\{ \begin{array}{l} \texttt{ON} \\ \texttt{OFF} \end{array} \right\}$
`[ ]`	In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted: `COMMAND` *filename* `[OPTION]` When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated. $\texttt{COMMAND}\ \textit{filename} \left[\begin{array}{l} \texttt{OPTION} \\ \textit{parameter} \end{array} \right]$

`[ ... ]`	In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:
	
	`[,`*parameter*`] [...]`
	
	In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:
	
	`[`*parameter*`] [,...]`

Conventions (continued)

| ... |	In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select `A, AB, BA`, or `B`. The elements cannot be repeated. $\left\{\begin{matrix}\mathtt{A}\\\mathtt{B}\end{matrix}\right\}$ | ... |
...	In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.
Δ	In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank: (*parameter*)Δ(*parameter*)
$\boxed{}$	The symbol $\boxed{}$ indicates a key on the keyboard. For example, $\boxed{\text{Return}}$ represents the carriage return key or $\boxed{\text{Shift}}$ represents the shift key.
$\boxed{\text{CTRL}}$*character*	$\boxed{\text{CTRL}}$*character* indicates a control character. For example, $\boxed{\text{CTRL}}$ `Y` means that you press the control key and the Y key simultaneously.
base prefixes	The prefixes %, #, and $ specify the numerical base of the value that follows: %*num* specifies an octal number. #*num* specifies a decimal number. $*num* specifies a hexadecimal number. If no base is specified, decimal is assumed.

Contents

Figures

Tables

1

Introduction to HP FORTRAN 77

The FORTRAN language was the first high level computer language to receive wide acceptance for application programming in the scientific community. First implemented in 1957, FORTRAN evolved through many changes and extensions, until in 1966 the American National Standards Institute (ANSI) published a “Standard FORTRAN” (X3.9-1966). This standard provided the basic structure of most FORTRAN compilers for many years.

Many compilers extended the standard. To further define the FORTRAN standard to include many of the extensions, ANSI updated the standard in 1977. The document describing this new standard, *American National Standard Programming Language FORTRAN, ANSI X3.9-1978* was published in 1978. Because most of the work on the language was completed in 1977, this standard FORTRAN is often called FORTRAN 77. In this manual, the ANSI standard is referred to as “the ANSI 77 standard.”

HP FORTRAN 77 is based on the ANSI 77 standard. It has many extensions to provide a more structured approach to program development and more flexibility in computing for scientific applications. Wherever such an extension is described, it is specifically referred to as “an extension to the ANSI 77 standard.” As part of its extensions, FORTRAN 77 fully implements those extensions described in the Department of Defense publication *Military Standard FORTRAN, DOD Supplement to American National Standard X3.9-1978*, MIL-STD-1753. Wherever such an extension is described, it is specifically referred to as “MIL-STD-1753 standard extension to the ANSI 77 standard.”

Note

In the rest of this manual, the term “FORTRAN” specifically refers to “HP FORTRAN 77”.

The FORTRAN 77 Compiler

The FORTRAN 77 compiler constructs object language programs from source language files written according to the rules of the FORTRAN language described in this manual. The FORTRAN 77 compiler is executable under various operating systems. The code generated by the compiler, standard binary output files, can be loaded and executed under the specific operating system. Exact details for specifying these files are found in the reference manuals for the operating system being used and in Chapter 10 of this manual.

FORTRAN 77 is a *multipass* compiler. A *pass* is a processing cycle of the source program. When the compiler is invoked, it produces a relocatable binary object program according to the options specified in its run string. Source and object listings can be produced if specified in the compiler. Refer to Chapter 5 for more details on the FORTRAN 77 compiler run string.

FORTRAN Vocabulary

A FORTRAN source file is composed of one or more program units. Each of the program units is constructed from characters grouped into lines and statements.

Sample FORTRAN Source File

Figure 1-1 shows a sample FORTRAN source file, consisting of one main program unit (`exone`) and one subprogram unit (`nfunc`). The line numbers are shown for reference only and do not appear in the source file. The definitions of the FORTRAN source file terms that follow refer to the sample program in Figure 1-1.

```
$LIST ON
      PROGRAM exone
C     This program shows program structure.
C     The purpose of the program is to compute
C     the sum of the first n integers using
C     a function subprogram unit.
C
      INTEGER*4 sum,nfunc   ! Specification statement.
*
      WRITE(6,'('' Enter value-->'')')    ! Prompt user.
      READ *,n             ! Enter integer limit to sum.
*     Compute sum in subprogram nfunc.
      sum=nfunc(n)         ! Invoke subprogram.
      WRITE(6,33) n,sum
   33 FORMAT(" Sum of the first ",I6,
     1 " integers = ",I10)  ! Continuation line.
      STOP
      END
*
*     Function subprogram unit follows.
      INTEGER*4 FUNCTION nfunc(k)
D     PRINT *, 'nfunc called, k = ', k
      nfunc = 0
      DO i = 1,k           ! Loop to compute sum.
          nfunc = nfunc+i
      END DO
      RETURN           ! Return value in function name.
      END
```

Figure 1-1. Sample FORTRAN Source File

FORTRAN Terms

Executable Program	An executable program is one that can be used as a self-contained computing procedure. An executable program consists of one main program and its subprograms, if any. (Figure 1-1 shows an executable program in its entirety.)
Program Unit	A program unit is a group of statements organized as a main program, a subprogram, or a block data subprogram. (In Figure 1-1, `exone` and `nfunc` are program units.)
Main Program	A main program is a set of statements and comments beginning with a PROGRAM statement or any other statement except a FUNCTION, SUBROUTINE, or BLOCK DATA statement, and ending with an END statement. (In Figure 1-1, lines 1 through 18 are a main program.)
Subprogram	A FORTRAN subprogram is a set of statements and comments headed by a FUNCTION, SUBROUTINE, or BLOCK DATA statement. When headed by a FUNCTION statement, it is called a function subprogram (In Figure 1-1, see lines 21 through 28); when headed by a SUBROUTINE statement, it is called a subroutine subprogram; and when headed by a BLOCK DATA statement, it is called a block data subprogram. Subprograms can also be written in other languages, such as Pascal or C.
Line	A line is a string of up to 72 characters. All characters must be from the HP ASCII character set, described in Appendix D. The character positions in a line are called columns, and are consecutively numbered 1, 2, 3, ... , 72 from left to right. (In Figure 1-1, 1 through 28 are lines.)
Initial Line	An initial line is not a comment line or a continuation line, and contains the digit `0` or a blank in column 6. Columns 1 through 5 can contain a statement label or blanks. (In Figure 1-1, lines 2, 8, 10, 11, 13, 14, 15, 17, 18, and 21 through 28 are initial lines.)
Continuation Line	A continuation line is a subsequent line of a multiple line statement. A continuation line contains any characters other than the digit `0` or a blank in column 6, and does not contain the character `C`, `*`, or `$` in column

1. Any characters can appear in columns 2 through 5. A tab character in column 1 through 6 and immediately followed by a digit from `1` to `9` is also a continuation indicator to the compiler; there must be blanks or nothing before the tab character. A line that is longer than 72 characters must use a continuation character and be continued on the next line.

A continuation line can follow only an initial statement line or another continuation line (unless separated from an initial line or continuation line by a comment line). By default, a statement can have up to 19 continuation lines. If the CONTINUATIONS compiler directive is specified, a statement can have up to 99 contnuation lines. (In Figure 1-1, line 16 is a continuation line.)

Statement

A statement is an initial line optionally followed by continuation lines. The statement is written in columns 7 through 72. The order of the characters in the statement is columns 7 through 72 of the first line, columns 7 through 72 of the first continuation line, and so on. (In Figure 1-1, lines 2, 8, 10, 11, 13 through 18, and 21 through 28 are statements.)

Directive Line

A directive line contains a `$` in column 1, and the text of the directive to the compiler in columns 2 through 72. (Refer to Chapter 8 for a list of the valid compiler directives.) A directive line can be continued. (In Figure 1-1, line 1 is a directive line.)

Comment Line

A comment line is marked by a `C`, an `!`, or an `*` in column 1. (In Figure 1-1, lines 3 through 7, 9, 12, 19, and 20 are comment lines.)

An exclamation point (`!`) in columns 7 through 72 signifies an end-of-line comment. This is an extension to the ANSI 77 standard. (In Figure 1-1, lines 8, 10, 11, 13, 16, 23, and 27 contain end-of-line comments.)

Debug Line

A debug line is marked by a `D` in column 1. It acts as either a statement or a comment line, depending on the current setting of the

DEBUG compiler directive. (In Figure 1-1, line 22 is a debug line).

Source File Structure

FORTRAN is column sensitive:

- Compiler directive lines (those starting with keywords preceded by a dollar sign) begin in column 1.
- All other FORTRAN statements can begin in columns 7 through 72. This permits indenting to improve program appearance.
- Statement labels appear in columns 1 through 5.
- Column 6 must be blank or contain a digit `0` for all lines except continuation, comment, and directive lines.
- A `C`, `!`, or `*` in column 1 denotes a comment line.
- A `D` in column 1 denotes a debug line.
- Tabs in columns 1 to 6 advance to column 7, while tabs in columns 7 to 72 are treated as spaces.

Figure 3-1 shows the required order of FORTRAN statements within program units.

2

Language Elements

A FORTRAN program is a sequence of statements that, when executed in a specified order, process data to produce desired results. Because each program has different data needs, FORTRAN provides 11 data types for constants, variables, functions, and expressions. FORTRAN also provides three additional constant formats, which are extensions to the ANSI 77 standard. All are described in "Data Types" later in this chapter. Keywords, special characters, special symbols, symbolic names, and data make up the statements of a FORTRAN program. This chapter describes the elements of statements.

The FORTRAN 77 Character Set

Each language element is written using the letters `A` to `Z`, the digits `0` to `9`, and the following special characters:

Character	Character Name	Character	Character Name
	Blank	,	Comma
=	Equals	:	Colon
+	Plus	'	Apostrophe (single quote)
-	Minus		
*	Asterisk		
/	Slash	!	Exclamation point [1]
(	Left parenthesis	"	Quotation mark [1] (double quote)
)	Right parenthesis	%	Percent sign [1]
.	Decimal point	&	Ampersand [1]
$	Dollar sign	_	Underscore [1] (break)

Note:

1. Extension to the ANSI 77 standard.

A tab character (Control-I) in columns 1 to 6 causes blank characters to be inserted up through column 6. For example, a tab character in column 2 inserts blanks in columns 2 to 6. Elsewhere, except when embedded in a literal string, a tab is interpreted as a blank character. If a tab character in column 1 to 6 is followed immediately by a digit from 1 to 9, with blanks or nothing before the tab character, the digit is treated as a continuation line indicator.

As an extension to the ANSI 77 standard, the 26 lowercase letters (`a` to `z`) are allowed. The compiler considers them identical to their uppercase equivalents, except in character or Hollerith constants.

(Note that this differs from the C language, in which lowercase letters are distinct from uppercase letters in identifiers.) Lowercase letters can improve program readability.

In addition, any printable ASCII character can be used in character and Hollerith constants, and in comments.

Blanks can be used anywhere within a statement. They are ignored except in character and Hollerith constants and in compiler directives.

Special Symbols

The special symbols are groups of characters that define specific operators and values. The special symbols are:

Symbol	Symbol Name	Symbol	Symbol Name
`**`	Exponentiation	`.TRUE.`	Logical true
`//`	Concatenation	`.FALSE.`	Logical false
`.EQ.`	Equal	`.NOT.`	Logical negation
`.NE.`	Not equal	`.AND.`	Logical AND
`.LT.`	Less than	`.OR.`	Logical inclusive OR
`.LE.`	Less than or equal	`.EQV.`	Logical equivalence
`.GT.`	Greater than	`.NEQV.`	Logical nonequivalence (same as `.XOR.`)
`.GE.`	Greater than or equal	`.XOR.`	Logical exclusive OR[1] (same as `.NEQV.`)

Note:

1. Extension to the ANSI 77 standard.

Keywords

Keywords are predefined FORTRAN entities that identify a statement or compiler directive. The statement keywords of FORTRAN are listed below. The compiler directive keywords are given in Chapter 7.

ACCEPT [1]	DOUBLE PRECISION	IF	RECORD [1]
ASSIGN	ELSE	IMPLICIT	RETURN
BACKSPACE	ELSE IF	INCLUDE [2]	REWIND
BLOCK DATA	ENCODE [1]	INQUIRE	REWRITE
BYTE [1]	END	INTEGER	SAVE
CALL	END DO [2]	INTRINSIC	STOP
CHARACTER	END MAP [1]	LOGICAL	STRUCTURE [1]
CLOSE	END STRUCTURE [1]	MAP [1]	SUBROUTINE
COMMON	END UNION [1]	NAMELIST [1]	THEN
COMPLEX	ENDIF	NONE [2]	TYPE [1]
CONTINUE	ENDFILE	OPEN	UNION [1]
DATA	ENTRY	PARAMETER	UNLOCK
DECODE [1]	EQUIVALENCE	PAUSE	VIRTUAL [1]
DELETE	EXTERNAL	PRINT	VOLATILE [1]
DIMENSION	FORMAT	PROGRAM	WHILE [2]
DO	FUNCTION	READ	WRITE
DOUBLE COMPLEX [1]	GOTO	REAL	

Notes:

1. Extension to the ANSI 77 standard.
2. MIL-STD-1753 standard extension to the ANSI 77 standard.

Comments

FORTRAN uses two types of comments: comment lines and embedded comments. A comment line is denoted in a source file by a `C`, `*`, or `!` in column 1, or by a blank line. A comment line is not a statement and does not affect the program in any way. Comment lines can be placed anywhere in a source file, including between lines of a continued statement.

An exclamation point (`!`) following a statement on the same line indicates the beginning of an embedded comment, unless the exclamation point is contained in a character or Hollerith constant. The compiler ignores the exclamation point and any text following; that is, it treats them as blanks. This use of the exclamation point is an extension to the ANSI 77 standard. Exclamation points are not allowed in directive lines.

Symbolic Names

Symbolic names define the names of any of the following:

- Main program
- Subroutine or function
- Block data subprogram
- Common block
- Named constant
- Simple variable
- Array
- Record, structure, and record field
- Namelist group-name

Symbolic names can be user-defined or predefined by FORTRAN. Each symbolic name consists of a sequence of characters, the first of which must be a letter. The rest can be letters, digits, the underscore character (_), or the dollar sign ($). The underscore character and the dollar sign in symbolic names are extensions to the ANSI 77 standard. Letters can be uppercase or, as an extension to the ANSI 77 standard, lowercase. The name can be any length, but only the first 32 characters are significant. This is also an extension to the ANSI 77 standard, because the standard permits only six characters.

Examples

```
FORTRAN_COMPILER_INITIALIZATION_SUBROUTINE
char_string
NumBer_of_ERRors
VAR$_1
REAL_VALUE
sum_of_real_values
error_flag
EXTERNAL_routine$
```

Notice that, because only the first 32 characters are significant, the compiler considers the following to be the same name:

```
                               Character 32
                                 |
FORTRAN_COMPILER_INITIALIZATION_SUBROUTINE
FORTRAN_COMPILER_INITIALIZATION_SUBPROGRAM
```

Because uppercase and lowercase letters are not distinguished in symbolic names, the following are equivalent:

```
result3
RESULT3
ResulT3
```

Note

Case is significant only when a letter is used in a character or Hollerith constant.

The name that identifies a variable, named constant, or function also identifies its default data type. A first letter of `I`, `J`, `K`, `L`, `M`, or `N` implies type INTEGER, either INTEGER*4 (default) or INTEGER*2, depending on the setting of the compiler directives LONG and SHORT. See "Data Types" for more detail. Any other letter implies type REAL. This default implied typing can be redefined with an IMPLICIT statement. It can be overridden with an explicit type statement.

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

Symbolic names can be identical to keywords because the interpretation of a sequence of characters is implied by its context. Similarly, the symbolic name of a named constant or variable can be the same as the symbolic name of a common block, without conflict.

The following are valid statements in FORTRAN:

Examples	Notes
`READ = IF + DO * REAL`	`READ`, `IF`, `DO`, and `REAL` are recognized as variables. They can also be used elsewhere as keywords in statements.
`IF (IF .EQ. GOTO) GOTO 99`	The `IF` and `GOTO` within the logical expression are recognized as variables. The `IF` and `GOTO` outside the expression are recognized as statement keywords.
`DO 10 j = 1.5`	The symbol `DO 10 j` is recognized as a variable, even though it contains blanks, mixed case, and the characters `DO`.

Although FORTRAN permits the above examples, using them is poor programming practice because they lessen program readability.

External Names

External names are a special type of user-defined name used by the linker. In FORTRAN, external names are generated for subroutines, functions, entry points, and common blocks. Unless the ALIAS or EXTERNAL_ALIAS directive is used (refer to Chapter 7), the external name is the same as the name used in the source code. A FORTRAN external name should never conflict with the name of a system routine or intrinsic.

FORTRAN Intrinsic Functions

FORTRAN intrinsic functions are symbolic names that are predefined by FORTRAN. Refer to Appendix B for a list of the FORTRAN intrinsic functions.

If a user-defined symbolic name is the same as a predefined symbolic name, any use of that name within the same program unit refers to the user-defined name. That is, the intrinsic function of that name is not recognized within the program unit. (Also refer to "EXTERNAL Statement (Nonexecutable)" in Chapter 3.)

Data Types

Each constant, variable, function, or expression is of one type only. The type defines:

- The set of values that an entity of that type can assume.
- The amount of storage that variables of that type require.
- The operations that can be performed on an entity of that type.

Warning messages are issued for duplicate data type declarations.

HP FORTRAN 77 has 11 data types, falling into five general categories, as shown in Table 2-1.

Table 2-1. Data Type Keywords

General Name	Data Type Keyword	Equivalent Keyword
Integer	BYTE [2]	LOGICAL*1 [2]
	INTEGER*2 [2]	INTEGER [1,3] (option)
	INTEGER*4 [2]	INTEGER [1,3] (default)
Real	REAL*4 [2]	REAL [1]
	REAL*8 [2]	DOUBLE PRECISION [1]
	REAL*16 [2]	(none)
Complex	COMPLEX*8 [2]	COMPLEX [1]
	COMPLEX*16 [2]	DOUBLE COMPLEX [2]
Logical	LOGICAL*2 [2]	LOGICAL [1,3] (option)
	LOGICAL*4 [2]	LOGICAL [1,3] (default)
Character	CHARACTER [1]	(none)

Notes:
1. ANSI 77 standard.
2. Extension to the ANSI 77 standard.
3. The equivalence depends on the setting of the compiler directives LONG and SHORT.
4. BYTE is a one byte integer which may be used in a logical context. It is sometimes called LOGICAL*1.

A keyword shown in column 3 of Table 2-1 has the same meaning as the keyword to the left in column 2 (subject to the compiler directives noted in the table). Consequently, in the rest of this manual, the data types will be referred to specifically by the keywords in column 2, and generally by the names in column 1.

For example, the word "Integer" will always mean any data defined with any of the keywords in the cells on the same row, and the keyword REAL*8 will always include data defined as DOUBLE PRECISION.

Note

By default, the type keywords INTEGER and LOGICAL are equivalent to INTEGER*4 and LOGICAL*4, respectively. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make INTEGER and LOGICAL equivalent to INTEGER*2 and LOGICAL*2, respectively. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 6 for further details.

The storage size and the range of values for each data type are shown in Table 2-2. Storage is measured in 8-bit bytes. Storage format is described in Chapter 10.

Table 2-2. Data Type Specifications

Data Type	Range of Values	Storage
BYTE (LOGICAL*1)	signed decimal −128 to +127 (a 1-byte integer); `.TRUE.` or `.FALSE.`; one 8-bit ASCII character	1 byte
INTEGER*2	−32768 to +32767	2 bytes
INTEGER*4	−2147483648 to +2147483647	4 bytes
REAL*4	0.0 and $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402823 \times 10^{+38}$	4 bytes
REAL*8	0.0 and $\pm 2.225073858507202 \times 10^{-308}$ to $\pm 1.797693134862315 \times 10^{+308}$	8 bytes
REAL*16	0.0 and $\pm 3.362103143112093506262677817321753 \times 10^{-4932}$ to $\pm 1.189731495357231765085759326628007 \times 10^{+4932}$	16 bytes
COMPLEX*8	Real and imaginary parts each have REAL*4 range.	8 bytes
COMPLEX*16	Real and imaginary parts each have REAL*8 range.	16 bytes
LOGICAL*2	`.TRUE.` or `.FALSE.`	2 bytes
LOGICAL*4	`.TRUE.` or `.FALSE.`	4 bytes
CHARACTER	One or more 8-bit ASCII characters	1 byte per character

As extensions to the ANSI 77 standard, HP FORTRAN 77 also provides three special constant data types, Hollerith, octal, and hexadecimal, shown in Table 2-3 with their ranges and storage sizes. These differ from other data types in that they cannot be associated with variables, functions, or expressions.

Table 2-3. Constant Data Types

Data Type	Range of Values	Storage
Hollerith	One or more 8-bit ASCII characters [1]	1 byte per character
Octal	Dependent on context [2]	16 bytes
Hexadecimal	Dependent on context [3]	16 bytes

Notes:
1. See "Typeless Constants" and "Hollerith Constants" in this chapter.
2. See "Typeless Constants" and "Octal Constants" in this chapter.
3. See "Typeless Constants" and "Hexadecimal Constants" in this chapter.

Each data type is described in this chapter, along with a description of the constants of each type. A constant is a data element that represents one specific value, such as `-3`, `.TRUE.`, `'character constant'`, or `47.21E-8`.

The PARAMETER statement allows you to give symbolic names to constants. The operations that can be performed on each data type are described in "Expressions" in this chapter. Each FORTRAN type statement is described in detail in Chapter 3. Refer to Chapter 10 for details on the data format in memory of each type.

BYTE (LOGICAL*1) Data Type

The BYTE (LOGICAL*1) data type can represent:

- A signed 8-bit integer in the range −128 to +127.
- The logical values true and false.
- An 8-bit ASCII character.

Variables and constants of type BYTE are stored in one byte. Refer to Chapter 10 for details. LOGICAL*1 and BYTE are extensions to the ANSI 77 standard. They are equivalent.

You can specify a BYTE variable explicitly by declaring it:

- In a LOGICAL*1 or BYTE type statement.
- In a LOGICAL, LOGICAL*2, or LOGICAL*4 type statement with a `*1` length override.

You can specify a BYTE variable implicitly without declaring it by beginning it with a letter that implies LOGICAL*1 or BYTE. Such initial letters can be set with an IMPLICIT statement. There is no default.

BYTE Constants

A BYTE constant can be:

- An integer constant, in the range −128 to +127.
- A logical constant, `.TRUE.` or `.FALSE.`, representing true or false, respectively. The periods are required, as shown.
- A character constant of one character.

Note

The underlying definition of a BYTE/LOGICAL*1 data type is as a one-byte integer. Its use as a logical and character datum is an addition to this definition. In most cases, these uses are unrestricted; however, when it is a list item in list-directed READ and WRITE statements it can only be used as a one-byte integer. See "List-Directed Input/Output" in Chapter 4 for further details.

INTEGER*2 Data Type

The INTEGER*2 data type represents the set of signed whole numbers in the range −32768 to +32767. Variables and constants of type INTEGER*2 are stored in two bytes. Refer to Chapter 10 for details. INTEGER*2 is an extension to the ANSI 77 standard.

You can specify an INTEGER*2 variable explicitly by declaring it:

- In an INTEGER*2 type statement.
- In an INTEGER type statement when INTEGER is equivalent to INTEGER*2.
- In an INTEGER or INTEGER*4 type statement with a `*2` length override.

You can specify an INTEGER*2 variable implicitly without declaring it by beginning it with a letter that implies INTEGER*2, or that implies INTEGER when INTEGER is equivalent to INTEGER*2. By default, the initial letters `I`, `J`, `K`, `L`, `M`, and `N` imply INTEGER. These defaults can be changed with an IMPLICIT statement.

Note

By default, the type keyword INTEGER is equivalent to INTEGER*4. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make INTEGER equivalent to INTEGER*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 6 for further details.

INTEGER*2 Constant

An integer constant consists of an optional plus (`+`) or minus (`-`) sign followed by one or more decimal digits (`0` to `9`). An INTEGER*2 constant has the whole-number range −32768 to +32767.

The default type for integer constants depends on the default type for the INTEGER keyword. If the default type for INTEGER is INTEGER*2 (that is, the SHORT compiler directive is in effect), then integer constants in the range −32768 to +32767 default to INTEGER*2. Otherwise (if LONG is in effect or the value is too large), integer constants default to INTEGER*4.

You may specify an INTEGER*2 constant explicitly by appending the letter `I` to the number. This is an extension to the ANSI 77 standard.

An integer constant value outside the INTEGER*4 range generates a compile-time error. If `I` is appended to the constant, a value outside the INTEGER*2 range also generates a compile-time error. When assigned to or read into a variable at run-time, a number outside the range of the variable causes an overflow condition. The handling of overflow conditions is system dependent. Refer to Chapter 9 for more details.

Examples

The following are valid INTEGER*2 constants.

```
-32767      -638      30000I      -4I      0      45
```

INTEGER*4 Data Type

The INTEGER*4 data type represents the set of signed whole numbers in the range −2147483648 to +2147483647. Variables and constants of type INTEGER*4 are stored in four bytes. Refer to Chapter 10 for details. INTEGER*4 is an extension to the ANSI 77 standard.

You can specify an INTEGER*4 variable explicitly by declaring it:

- In an INTEGER*4 type statement.
- In an INTEGER type statement when INTEGER is equivalent to INTEGER*4.
- In an INTEGER or INTEGER*2 type statement with a `*4` length override.

You can specify an INTEGER*4 variable implicitly without declaring it by beginning it with a letter that implies INTEGER*4, or that implies INTEGER when INTEGER is equivalent to INTEGER*4. By default, the initial letters `I`, `J`, `K`, `L`, `M`, and `N` imply INTEGER. These defaults can be changed with an IMPLICIT statement.

Note

By default, the type keyword INTEGER is equivalent to INTEGER*4. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make INTEGER equivalent to INTEGER*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 6 for further details.

INTEGER*4 Constant

An integer constant consists of an optional plus (`+`) or minus (`-`) sign followed by one or more decimal digits (`0` to `9`). An INTEGER*4 constant has the whole-number range −2147483648 to +2147483647.

The default type for integer constants depends on the default type for the INTEGER keyword. If the default type for INTEGER is INTEGER*2 (that is, the SHORT compiler directive is in effect), then integer constants in the range −32768 to +32767 default to INTEGER*2. Otherwise (if LONG is in effect or the value is too large), integer constants default to INTEGER*4.

You may specify an INTEGER*4 constant explicitly by appending the letter `J` to the number. This is an extension to the ANSI 77 standard.

An integer constant value outside the INTEGER*4 range generates a compile-time error. When assigned to or read into a variable at run-time, a number outside the range of the variable causes an overflow condition. The handling of overflow conditions is system dependent. Refer to Chapter 9 for more details.

Examples

The following are valid INTEGER*4 constants.

```
   -3          14              -99526     30000J
```

-4J	2147483647	0	32768

REAL*4 Data Type

The REAL*4 data type, sometimes called “single precision”, represents the set of real numbers whose normal range is 0.0 and $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402823 \times 10^{+38}$ and whose precision is approximately seven decimal digits. Variables and constants of type REAL*4 are stored in four bytes in floating point format. Refer to Chapter 10 for details. REAL*4 is an extension to the ANSI 77 standard. It is equivalent to the ANSI standard REAL type.

You can specify a REAL*4 variable explicitly by declaring it:

- In a REAL*4 or a REAL type statement.
- In a REAL*8, REAL*16, or DOUBLE PRECISION type statement with a `*4` length override.

You can specify a REAL*4 variable implicitly without declaring it by beginning it with a letter that implies REAL*4 or REAL. By default, the initial letters `A` to `H` and `O` to `Z` imply REAL. These defaults can be changed with an IMPLICIT statement.

REAL*4 Constant

A REAL*4 constant must contain a decimal point or an exponent or both. It can have a leading plus (`+`) or minus (`-`) sign. The exponent is specified with the letter `E`.

Syntax

sn`.`*n*

s`.`*n*

sn`.`

sn`.`*n*`E`*s*`e`

s`.`*n*`E`*s*`e`

sn`.E`*s*`e`

sn`E`*s*`e`

Item	Description/Default	Restrictions
s	Optional sign.	None.
n	Whole number or fraction of value.	One or more decimal digits.
e	Exponent.	One or more decimal digits.

The construct `E`*se* represents a power of 10. For example:

`14.E-5` = $14. \times 10^{-5}$ = `.00014`

`5.834E2` = 5.834×10^{2} = `583.4`

Examples

The following are valid REAL*4 constants.

```
-.74E-12     1.99526.     .125             5.997255E8
```

```
-99526.      10.          23.99844E-25     6E0
```

REAL*8 Data Type

The REAL*8 data type, sometimes called “double precision”, represents the set of real numbers whose normal range is 0.0 and $\pm 2.225073858507202 \times 10^{-308}$ to $\pm 1.797693134862315 \times 10^{+308}$ and whose precision is approximately 17 decimal digits. Variables and constants of type REAL*8 are stored in eight bytes in floating point format. Refer to Chapter 10 for details. REAL*8 is an extension to the ANSI 77 standard. It is equivalent to the ANSI standard DOUBLE PRECISION type.

You can specify a REAL*8 variable explicitly by declaring it:

- In a REAL*8 or a DOUBLE PRECISION type statement.
- In a REAL*4 or REAL*16 type statement with a `*8` length override.

You can specify a REAL*8 variable implicitly without declaring it by beginning it with a letter that implies REAL*8 or DOUBLE PRECISION. Such initial letters can be set with an IMPLICIT statement. There is no default.

REAL*8 Constant

A REAL*8 constant can contain a decimal point. It must have an exponent. It can have a leading plus (`+`) or minus (`-`) sign. The exponent is specified with the letter `D`.

Syntax

sn`.`*n*`D`*se*

s`.`*n*`D`*se*

sn`.D`*se*

sn`D`*se*

Item	Description/Default	Restrictions
s	Optional sign	None
n	Whole number or fraction of value.	One or more decimal digits.
e	Exponent.	One or more decimal digits.

The construct `D`*se* represents a power of 10. For example:

`14.D-5` = $14. \times 10^{-5}$ = `.00014`

`5.834D2` = 5.834×10^{2} = `583.4`

Examples

The following are valid REAL*8 constants.

```
-.74D-12      10.D0                  5.99725529D8
1.99526D1     23.9984432697338D-25   6D0
```

REAL*16 Data Type

The REAL*16 data type, sometimes called “quad precision”, represents the set of real numbers whose normal range is 0.0 and $\pm 3.362103143112093506262677817321753 \times 10^{-4932}$ to $\pm 1.189731495357231765085759326628007 \times 10^{+4932}$ and whose precision is approximately 34 decimal digits. Variables and constants of type REAL*16 are stored in 16 bytes in floating point format. Refer to Chapter 10 for details. REAL*16 is an extension to the ANSI 77 standard.

You can specify a REAL*16 variable explicitly by declaring it:

- In a REAL*16 type statement.
- In a REAL*4, REAL*8, or DOUBLE PRECISION type statement with a `*16` length override.

You can specify a REAL*16 variable implicitly without declaring it by beginning it with a letter that implies REAL*16. Such initial letters can be set with an IMPLICIT statement. There is no default.

REAL*16 Constant

A REAL*16 constant can contain a decimal point. It must have an exponent. It can have a leading plus (`+`) or minus (`-`) sign. The exponent is specified with the letter `Q`.

Syntax

sn`.`*n*`Q`*se*
s`.`*n*`Q`*se*
sn`.Q`*se*
sn`Q`*se*

Item	Description/Default	Restrictions
s	Optional sign	None
n	Whole number or fraction of value.	One or more decimal digits.
e	Exponent.	One or more decimal digits.

The construct `Q`*se* represents a power of 10. For example:

`14.Q-5` = $14. \times 10^{-5}$ = `.00014`

`5.834Q2` = 5.834×10^{2} = `583.4`

Examples

The following are valid REAL*16 constants.

```
6Q0           -6.475175119438025110924438958227647Q-4966
1.0Q+15       +1.189731495357231765085759326628007Q+4932
.00001Q-10    3.1415926535897 9Q0
```

COMPLEX*8 Data Type

The COMPLEX*8 data type defines a set of complex numbers whose representation is an ordered pair of REAL*4 values. The first of the pair represents the real part of the value and the second represents the imaginary part. Each part has the same range and precision as a REAL*4 value. Variables and constants of type COMPLEX*8 are stored in eight bytes as two REAL*4 values. Refer to Chapter 10 for details. COMPLEX*8 is an extension to the ANSI 77 standard. It is equivalent to the ANSI standard COMPLEX data type.

You can specify a COMPLEX*8 variable explicitly by declaring it:

- In a COMPLEX*8 or a COMPLEX type statement.
- In a COMPLEX*16 or DOUBLE COMPLEX type statement with a `*8` length override.

You can specify a COMPLEX*8 variable implicitly without declaring it by beginning it with a letter that implies COMPLEX*8 or COMPLEX. Such initial letters can be set with an IMPLICIT statement. There is no default.

COMPLEX*8 Constant

The form of a COMPLEX*8 constant is an ordered pair of numeric constants (which may each be REAL*4, INTEGER*4, or INTEGER*2), separated by a comma, and surrounded by parentheses.

Syntax

(*real_part* , *imag_part*)

Examples

The following are valid COMPLEX*8 constants.

```
(3.0,-2.5E3)      (3.5,5.4)         (45.9382,12)
(0,0)             (-187,-160.5)
```

COMPLEX*16 Data Type

The COMPLEX*16 data type defines a set of complex numbers whose representation is an ordered pair of REAL*8 values. The first of the pair represents the real part of the value and the second represents the imaginary part. Each part has the same range and precision as a REAL*8 value. Variables and constants of type COMPLEX*16 are stored in 16 bytes as two REAL*8 values. Refer to Chapter 10 for details. COMPLEX*16 and DOUBLE COMPLEX are extensions to the ANSI 77 standard. They are equivalent.

You can specify a COMPLEX*16 variable explicitly by declaring it:

- In a COMPLEX*16 or DOUBLE COMPLEX type statement.
- In a COMPLEX*8 or COMPLEX type statement with a `*8` length override.

You can specify a COMPLEX*16 variable implicitly without declaring it by beginning it with a letter that implies COMPLEX*16 or DOUBLE COMPLEX. Such initial letters can be set with an IMPLICIT statement. There is no default.

COMPLEX*16 Constant

The form of a COMPLEX*16 constant is an ordered pair of numeric constants, separated by a comma, and surrounded by parentheses. One of the pair of constants must be REAL*8. The other can be REAL*8, REAL*4, INTEGER*4, or INTEGER*2).

Syntax

(*real_part* , *imag_part*)

Examples

The following are valid COMPLEX*16 constants.

```
(-187,-160.5)          (0,5.99537D5)      (3.5,5.4D0)
(0,0D0)                (3.0,-2.5D3)       (45.9382D0,12)
(-153D-12,4.66257)     (1.56792456774D-24,-9.74375486354D-21)
```

LOGICAL*2 Data Type

The LOGICAL*2 data type represents the logical values true and false. Variables and constants of type LOGICAL*2 are stored in two bytes. Refer to Chapter 10 for details. LOGICAL*2 is an extension to the ANSI 77 standard.

You can specify an LOGICAL*2 variable explicitly by declaring it:

- In a LOGICAL*2 type statement.
- In a LOGICAL type statement when LOGICAL is equivalent to LOGICAL*2.
- In a LOGICAL, LOGICAL*1, or LOGICAL*4 type statement with a `*2` length override.

You can specify a LOGICAL*2 variable implicitly without declaring it by beginning it with a letter that implies LOGICAL*2, or that implies LOGICAL when LOGICAL is equivalent to LOGICAL*2. Such initial letters can be set with an IMPLICIT statement. There is no default.

Note

By default, the type keyword LOGICAL is equivalent to LOGICAL*4. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make LOGICAL equivalent to LOGICAL*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 6 for further details.

LOGICAL*2 Constants

A LOGICAL*2 constant has the following forms and values:

Constant	Value
`.FALSE.`	Logical false
`.TRUE.`	Logical true

The periods are required, as shown.

LOGICAL*4 Data Type

The LOGICAL*4 data type represents the logical values true and false. Variables and constants of type LOGICAL*4 are stored in four bytes. Refer to Chapter 10 for details. LOGICAL*4 is an extension to the ANSI 77 standard.

You can specify an LOGICAL*4 variable explicitly by declaring it:

- In a LOGICAL*4 type statement.
- In a LOGICAL type statement when LOGICAL is equivalent to LOGICAL*4.
- In a LOGICAL, LOGICAL*1, or LOGICAL*2 type statement with a `*4` length override.

You can specify a LOGICAL*4 variable implicitly without declaring it by beginning it with a letter that implies LOGICAL*4, or that implies LOGICAL when LOGICAL is equivalent to LOGICAL*4. Such initial letters can be set with an IMPLICIT statement. There is no default.

Note

By default, the type keyword LOGICAL is equivalent to LOGICAL*4. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make LOGICAL equivalent to LOGICAL*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 6 for further details.

LOGICAL*4 Constants

A LOGICAL*4 constant has the following forms and values:

Constant	Value
`.FALSE.`	Logical false
`.TRUE.`	Logical true

The periods are required, as shown.

CHARACTER Data Type

The CHARACTER data type represents a string of characters. The string can consist of any characters from the 8-bit ASCII character set, described in Appendix D. Variables and constants of type CHARACTER are stored in one byte per character.

You can specify an CHARACTER variable explicitly by declaring it in a CHARACTER type statement.

You can specify a CHARACTER variable implicitly without declaring it by beginning it with a letter that implies CHARACTER. Such initial letters can be set with an IMPLICIT statement. There is no default.

Each character in a string has a character position that is numbered consecutively: 1, 2, 3, and so forth. The number indicates the sequential position of a character in the string, from left to right.

CHARACTER Constant

The form of a character constant is an apostophe (') or quotation mark ("), optionally followed by a string of characters, and terminated with a pairing apostrope or quotation mark. The use of the quotation mark is an extension to the ANSI 77 standard.

Syntax

> ' [*character*] [. . .] '
>
> " [*character*] [. . .] "

The length of a character constant is the number of characters between the delimiting characters (which are not counted).

If an apostrophe is included in a string delimited by apostrophes, or a quotation mark is included in a string delimited by quotation marks, it must be written twice with no intervening blanks to distinguish it from the delimiting characters. Such pairs count as one character.

As an extension to the ANSI 77 standard, null strings are permitted in the same context where other strings are allowed. Null strings and non-null strings are equivalent because they follow the rules of character constants or typeless constants depending on the context.

As an extension to the ANSI 77 standard, character literals can represent numeric constants. See the following section, "Typeless Constants", for details.

You can include nonprintable characters in a string, but it is better to specify these with the CHAR intrinsic function and concatenate them to a string. See Appendix B for details. The blank character is valid and significant in a CHARACTER value. Lowercase characters are not identical to their uppercase equivalents in CHARACTER values.

Examples

```
'Input the next item'              "Item #1 =>"
'EXPECTING A "1" OR A "2"'         "EXPECTING A ""1"" OR A ""2"""
'EXPECTING A ''1'' OR A ''2'''     "EXPECTING A '1' OR A '2'"
'That''s life!'                    "That's life!"
''                                 ""
```

Typeless Constants

Hollerith, octal, and hexadecimal constants (see following sections) are considered typeless constants. Character constants that are used in numeric expressions are handled like Hollerith constants. Typeless constants are extensions to the ANSI 77 standard. A *typeless constant* is a constant that does not undergo the type checking that would normally prevent you from using it in expressions.

The following four assignments to INTEGER*4 variable `i` result in identical values for `i`.

`i = 'ffff'`	*character constant*
`i = 1717986918`	*decimal integer constant*
`i = '66666666'X`	*hexadecimal constant*
`i = 4Hffff`	*Hollerith constant*

The numeric value of the character constant `'ffff'` is quite different from the numeric value of the hexadecimal constant `'ffff'X`.

`'ffff' = "ffff" = 1717986918`	*the* `f`*s are letters having ASCII byte values*
`'ffff'X = 'FFFF'X = 65535`	*the* `f`*s are hexadecimal digits having half-byte values*

Typical uses of typeless constants include:

- Performing integer arithmetic with binary values
- Manipulating expressions oriented to bit masks
- Pattern handling
- Performing simple arithmetic on ASCII values

All typeless constants are internally converted to a 32-digit (16-byte) hexadecimal value and eventually converted to one of the FORTRAN 77 standard types. The conversion rule for typeless constants is as follows: If a typeless constant appears in an expression with an operand that has an assigned type, it takes the type of the other operand. If this rule cannot be applied, the typeless constant is converted to INTEGER*4.

The following examples illustrate the conversion rule for typeless constants:

Examples	Notes

`REAL r`
`r = 'CAFE'X + 1`
	The hexadecimal value of `'CAFE'` is taken as INTEGER*4 because the other operand, 1, is an INTEGER*4. `'CAFE'` becomes the integer value 51966, which is added to the value 1. The resulting integer value 51967 is assigned to `r`.

`INTEGER n`
`n = 'CAFE'X + 0I`
	The hexadecimal value of `'CAFE'` is taken as INTEGER*2 because the other operand, `0I`, is INTEGER*2. Therefore the value −13570 is assigned to `n`.

`REAL r`
`r = 'CAFE'X + 1.0`
	The hexadecimal value of `'CAFE'` is taken as the real value 7.28198E−41 because the other operand, 1.0, is real. The value 7.28198E−41 is so small that, when it is added to 1.0 using floating-point arithmetic, the result is 1.0, which is the value assigned to `r`.

Note

As illustrated in the last example, you must use caution when mixing typeless constants and floating-point types.

When typeless constants are passed as parameters to subprograms, the default INTEGER*4 type is assumed unless the constant is embedded in an expression. Therefore, the statement:

```
CALL SUBROUTINE('CAFE'X)
```

passes the INTEGER*4 value 51966 to the subroutine. However, the statement:

```
CALL SUBROUTINE('CAFE'X+0.0)
```

passes the real value 7.28198E−41 and the statement:

```
CALL SUBROUTINE(('CAFE'X+0)+0.0)
```

passes the real value 51966.0.

Typeless constants can be used wherever constant expressions are allowed. They can be mixed with other constants regardless of their final value. Therefore, expressions such as the following are possible:

```
I = z'a' + "bc" * (-123b/('x'-1)) + '0'x - 9j
```

This assignment yields the value 1 (of type INTEGER*4).

When character and Hollerith constants are found in arithmetic expressions such as the one above, they are padded on the right with blanks (hexadecimal 20). Therefore, the following are all equivalent:

```
I = '0'
I = '0ΔΔΔ'
I = 1H0
I = 4h0ΔΔΔ
```

Hollerith Constants

Hollerith constants are an extension to the ANSI standard. They are available for compatibility with older programs and with some system routines. They can appear in arithmetic expressions representing ASCII values. A Hollerith constant consists of a positive integer constant specifying the number of characters (including blanks), followed by the letter `H` and the character string, which can include trailing blanks.

Syntax

*n*H*c*[*c*] [. . .]

Examples

```
2H$$
8Ha string
12HReport Title
6H&proga
3H12a
7Hqu'oted
```

Hollerith constants can be used in most places where an integer constant is allowed, such as in assignment statements, DATA statements, PARAMETER statements, and equality comparisons (for example, `.EQ.` or `.NE.`). Hollerith constants can be assigned and compared to arithmetic variables and expressions, but not to logical expressions. When necessary, a Hollerith constant is truncated on the right or blank-filled on the right so that its length is equal to that of the other operand. The resulting type of the Hollerith constant is that of the argument on the other side of the operator. Hollerith to Hollerith operations are not allowed.

Data type is not assumed when Hollerith constants are used as arguments. Note, however, that Hollerith constants cannot be passed as character constants. Hollerith and character constants are not interchangeable.

When Hollerith constants are used as arguments, no blank filling occurs. Therefore, the lengths of Hollerith constants must correspond correctly with formal arguments.

Examples	Notes
`r = 2Hab`	These two statements are equivalent.
`r = 4HabΔΔ`	

`COMPLEX c`
`IF (c .NE. 19Hwhen the wind blows)`
	These two statement pairs are equivalent.

`COMPLEX c`
`IF (c .NE. 8Hwhen the) ...`	

`i2 = 2hxy`	The resulting value of `i2` is 30841 (type INTEGER*2).
`i4 = 2hxy`	The resulting value of `i4` is 2021204000 (type INTEGER*4).

Blank filling and truncation to resolve length differences can be accomplished on Hollerith constants only, and not on arithmetic variables created from Hollerith data. On arithmetic variables, the appropriate arithmetic conversions are performed. For example, if `i2` is equal to `2Hxy`, and `i4` is equal to `i2`, then `i4` is not equal to `4Hxy`.

Note

Hollerith literals can represent numeric constants. See "Typeless Constants" earlier in this chapter for details.

Octal Constants

Octal constants are an extension to the ANSI 77 standard. They are a special format of octal values that are stored internally as hexadecimal values of up to 32 hexadecimal digits (16 bytes). Eventually they are converted to a standard type.

Octal constants are left-padded with zeros. For example,

```
O'7777'
```

is stored internally as the hexadecimal value:

```
00000000000000000000000000000FFF
```

(that is, `FFF` preceded by 29 zeros).

Three formats are allowed for octal constants:

Syntax

*sn*B
O'*n*'
'*n*'O

Item	Description/Default	Restrictions
s	Optional sign.	None.
n	Unsigned octal number.	Contains only the octal digits `0` to `7`.

The O'*n*' form is a MIL-STD-1753 standard extension to the ANSI 77 standard.

Octal constants can be used in most places where an integer constant is allowed. See "Typeless Constants" earlier in this chapter for details.

Note

For good programming style, you should avoid using octal constants in floating-point expressions.

Examples

```
400B             O'2137'             '2137'O
100000B          O'37777777777'      37777777777B
```

Octal constants are not assigned a type. The data type to which they are converted is determined by the context in which they are found, as explained in the next two sections.

Octal Constants in Assignments

When associated with another operand in an assignment statement, an octal constant takes the type of the other operand. If no type can be taken from the other operand, INTEGER*4 is assumed.

If not associated with another operand in an assignment statement, an octal constant takes the type of the entity on the left side of the equal sign. This is illustrated in the first two assignments in the following examples.

Examples	**Notes**

`INTEGER*2 I2`
`INTEGER*4 I4`
`.`
`.`
`.`
`I2 = O'54131'`
	The resulting value of `I2` is 22617 (type INTEGER*2). This is the numeric equivalent of the Hollerith constant `'XX'`.

`I4 = o'54132'`	The resulting value of `I4` is 22618 (type INTEGER*4). This is the numeric equivalent of the Hollerith constant `'XY'`.
`I2 = I4 - O'1'`	The resulting value of `I4` is 22617 (INTEGER*2). This is the numeric equivalent of `'XX'` . Note that this operation is not possible using Hollerith constants (see the next two assignments).
`I2 = 2HXX`	The resulting value of `I2` is 22617 (type INTEGER*2).
`I4 = 2HXY`	The resulting value of `I4` is 1482235936 (type INTEGER*4). Two blanks have been appended.
`I2 = I4 - O'1'`	The resulting value of `I2` is 8223 (type INTEGER*2). The result has been truncated.
`I4 = 1 + 4H0000 + O'1'`	The resulting value of `I4` is 808464434 (type INTEGER*4). This is the numeric equivalent of `'0002'` .

Octal Constants as Actual Parameters

When used as actual parameters, octal constants are converted to INTEGER*4. To pass an octal constant as a different data type, use the constant in an expression of the desired type.

Examples	**Notes**
`CALL example(O'7777')`	Passes the value 4095 as data type INTEGER*4 (the default).
`CALL example(O'7777' + I2)`	Passes the value 4095 as data type INTEGER*2.

Note

When the data type that receives an octal constant does not have sufficient space to hold all the significant bits, a warning is issued and the constant is left-truncated to fit as many bits as possible into the variable. For example, if the following assignment is made:

```
I2 = o'76543210'
```

`I2` (INTEGER*2) is assigned the value `o'143210'`, which is the maximum number of bits of that value that can fit into the INTEGER*2 variable.

Hexadecimal Constants

Hexadecimal (base 16) constants are an extension to the ANSI 77 standard. They are stored internally as hexadecimal values of up to 32 hexadecimal digits (16 bytes), and eventually they are converted to a standard type. The `Z'`*n*`'` form is a MIL-STD-1753 extension to the ANSI 77 standard.

Hexadecimal constants are left-padded with zeros. For example,

```
Z'FFFF'
```

is stored internally as:

```
0000000000000000000000000000FFFF
```

(that is, `FFFF` preceded by 28 zeros).

The following formats are allowed for hexadecimal constants:

Syntax

`Z'`*n*`'`

`'`*n*`'X`

Item	Description/Default	Restrictions
n	Unsigned hexadecimal number.	Contains the hexadecimal digits `0` to `9` and `A` to `F`.

Either form can be used in most places where an integer constant is allowed. See "Typeless Constants" earlier in this chapter for details.

Examples

```
Z'f921'
z'CAFE'
'F9A1'X
'cafe'x
```

Hexadecimal constants are not assigned a type. They are converted to a standard FORTRAN type according to the context in which they are found. This is explained in the next two sections.

Hexadecimal Constants in Assignments

When associated with another operand in an assignment statement, a hexadecimal constant takes the type of the other operand. If no type can be taken from the other operand, INTEGER*4 is assumed.

If not associated with another operand in an assignment statement, a hexadecimal constant takes the type of the entity on the left side of the equal sign. This is illustrated in the first two assignments in the following example.

Examples	Notes

`INTEGER*2 I2`
`INTEGER*4 I4`
`DOUBLE PRECISION R8`
.
.
.
`I2 = Z'FFFF'`
	The resulting value of `I2` is −1 (type INTEGER*2).

`I4 = Z'FFFF'`	The resulting value of `I4` is 65535 (type INTEGER*4).
`R8 = z'3ff0000000000000'`	The resulting value of `R8` is 1.0 (type s'' 'REAL*8'').
`I4 = Z'3FF0000000000000' - R8`	The resulting value of `I4` is 0 (type INTEGER*4).
`I4 = Z'1' - R8`	The resulting value of `I4` is −1 (type INTEGER*4).
`I4 = z'1' + 4H0000 + o'1'`	The resulting value of `I4` is 808464434 (type INTEGER*4). This is the numeric equivalent of `'0002'`.

Note

As a good programming practice, you should avoid using hexadecimal constants in floating-point expressions.

Hexadecimal Constants as Actual Parameters

When used as actual parameters, hexadecimal constants are converted to INTEGER*4. To pass a hexadecimal constant as a different data type, the constant should be used in an expression of the desired type.

Examples	Notes
`CALL example(Z'FFF')`	Passes the value 4095 as data type INTEGER*4 (the default).
`CALL example(Z'FFF' + 0I2)`	Passes the value 4095 as data type INTEGER*2.

Note

When the data type that receives a hexadecimal constant does not have sufficient space to hold all the significant bits, a warning is issued and the constant is left-truncated to fit as many bits as possible into the variable. For example, if the following assignment is made:

```
I2 = z'abcdef'
```

`I2` (type INTEGER*2) is assigned the value `z'cdef'`.

Variables

A variable name is a symbolic name that represents a data element whose value can be changed during program execution by the use of assignment statements, READ statements, and so forth.

A variable can represent a single value of one simple type, such as character, complex, integer, logical, or real; a collection of values of the same type, as in an array; or a collection of values of different types, as in a record.

Refer to "Symbolic Names" earlier in this chapter for a description of valid variable names.

Simple Variables

A simple variable is used to process a single data item. It identifies a storage area that can contain only one value at a time. Subscripted variables are treated in this manual as simple variables unless stated otherwise.

Examples

```
total
voltage
Final_Score
i
sum_of_values
ERROR_FLAG1
array3_element(i,j)
FORMAT
```

Arrays

An array is a collection of several values of the same type. An array name is a symbolic name that represents all values or elements of an array. To designate exactly one element of the array, follow the array name with one or more subscripts.

A group of values arranged in a single row is a one-dimensional array. The elements of such an array are identified by a single subscript. If two subscripts are used to identify an element of an array, then that array is two-dimensional, and so forth. An array can have an unlimited number of dimensions. The number of dimensions allowed in an array is system dependent.

Array Declarators

Array declarators are used in DIMENSION, COMMON, VIRTUAL, and type declaration statements to define the number of dimensions, the number of elements per dimension (called bounds), and the data to be stored in the elements.

Syntax

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

Item	Description/Default	Restrictions
name	Symbolic name of the array.	None.
d	Dimension declarator.	There must be one dimension declarator for each dimension of the array.

Examples	Notes
`DIMENSION xyz(4,2,4)`	Three-dimensional REAL array of `xyz` with 32 elements.
`COMMON iabc(3,4)`	Two-dimensional INTEGER*4 array of `iabc` with 12 elements.
`INTEGER*2 I2(4)`	One-dimensional INTEGER*2 array of `I2` with 4 elements.

The syntax of a dimension declarator is:

Syntax

[*m* :] *n*

Item	Description/Default	Restrictions
m	Lower dimension bound.	None.
n	Upper dimension bound.	The upper bound must be greater than or equal to the lower bound.

If only the upper dimension bound is specified, the value of the lower dimension bound is one. The value of either dimension bound can be positive, negative, or zero; however, the value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound.

The lower and upper dimension bounds are arithmetic expressions containing constants, symbolic names of constants, or variables. The expressions defining the upper and lower bounds must not contain a function or array element reference. The upper dimension bound of the last dimension in the array declarator of a formal argument can be an asterisk, signifying that the last dimension is assumed (undefined).

Note

Using an asterisk in a dimension declarator is limited to declarators of formal arguments of subprograms.

The array bounds indicate the number of dimensions of the array and the maximum number of elements in each dimension. The number of elements in each dimension is defined by *n* `-` *m* `+ 1`, where *n* is the upper bound and *m* is the lower bound.

Examples	Notes
`name(4,-5:5,6)`	Specifies a three-dimensional array. The first dimension can have four elements, the second 11, and the third six.
`decision_table (2,3,2,2,3,4,2)`	Specifies a seven-dimensional array.
`m(0:0)`	Specifies a one-dimensional array of one element: `m(0)`.
`list(10)`	Specifies a one-dimensional array of 10 elements: `list(1)` to `list(10)`.

A complete array declarator for a particular array can be used once only in a program unit, although the array name can appear in several specification statements. For example, if the array declarator is used in a DIMENSION statement, the array name can only be used in a COMMON or type statement. If the complete array declarator is used in a COMMON or type statement, the array must not be mentioned in a DIMENSION statement.

Normally, array bounds are specified with integer constants. If the bounds are specified with integer variables, the integer variables must be formal arguments to the subprogram. However, the array itself can either be a formal argument or a nonstatic local variable (that is, one that does not appear in a SAVE or DATA statement). See "Adjustable Arrays" and "Dynamic Arrays" in the following sections for further information.

Adjustable Arrays

Normally, array bounds are specified by integer constants and are determined by the values of these constants. In an adjustable array, one or more of the array bounds are specified by an expression involving integer variables instead of integer constants.

Adjustable arrays can be used in subprograms to allow the array bound to be defined as a value passed from the caller of the subprogram. The array bounds are therefore formal arguments, and storage is allocated for the array by the caller of the subprogram in which the array is found. The next example illustrates an adjustable array.

Examples	Notes
`PROGRAM main`	Storage is allocated for `array` by the program
`INTEGER array(10)`	`main`. The subprogram `routine` uses the variable
`i = 10`	`i` only for bounds checking and subscript
`CALL routine(array,i)`	calculation.
`END`	
`SUBROUTINE routine(ar,i)`	
`INTEGER ar(i)`	
`ar(1) = i`	
`END`	

Dynamic Arrays

An array that is a nonstatic local variable is called a *dynamic array*. For a dynamic array, storage is allocated by the current subprogram dynamically on the stack. This dynamic array feature is an HP extension to the ANSI 77 standard. The next example illustrates the use of a dynamic array.

Examples	Notes
`PROGRAM main`	`main` passes only the integer `i` to `routine`.
`i = 10`	`dyn_array` is a nonstatic local array. The
`CALL routine(i)`	subprogram allocates storage for 10 4-byte
`END`	integers on the stack for `dyn_array`.
`SUBROUTINE routine(i)`	
`INTEGER dyn_array(i)`	
`dyn_array(i) = i`	
`END`	

Subscripts

Subscripts designate a specific element of an array. An array element reference (subscripted variable) must contain the array name followed by as many subscripts as there are dimensions in the array. The subscripts are separated by commas and enclosed in parentheses. Each subscript value must fall between the declared lower and upper bounds for that dimension.

For example, a subscripted variable for a one-dimensional array of three elements declared by `a(3)` or `a(1:3)` could have the form `a(1)`, `a(2)`, or `a(3)` to represent the elements of the array `a`. If a subscript is outside its declared lower and upper bounds, the results are unpredictable; the compiler does not generate an error message (unless the RANGE option is specified).

Examples	Notes
`arr(1,2)`	Represents the element `1,2` of the array `arr`. If `arr` was declared by `arr(10,20)`, `arr` would describe a two-dimensional table and `arr(1,2)` would describe the element in the second column of the first row.
`chess_board(i,j,k)`	Subscripts `i`, `j`, and `k` are variables that represent different elements of array `chess_board`.
`arr(i+4,j-2)`	Subscripts `i+4` and `j-2` are expressions that represent specific elements of array `arr` when evaluated.

Array Element Storage

The total number of elements in an array is calculated by multiplying the number of elements in each dimension. For example, the array declarator `i(3,4,-3:5)` indicates that array `i` contains 108 elements:

```
3*4*(5-(-3)+1) = (3*4*9) = 108
```

The amount of memory needed to store an array is determined by the number of elements in the array and the type of data that the array contains.

LOGICAL*1 arrays store each element in one byte.

INTEGER*2 and LOGICAL*2 arrays store each element in two bytes.

INTEGER*4, REAL*4, and LOGICAL*4 arrays store each element in four bytes.

REAL*8 and COMPLEX*8 arrays store each element in eight bytes.

REAL*16 and COMPLEX*16 arrays store each element in 16 bytes.

CHARACTER arrays store each character in one byte.

A one-dimensional array is stored as a linear list. Arrays of higher dimensions are stored in “column major order”, with the first subscriptvarying most rapidly, the second the next most rapidly, and so forth, with the last varying least rapidly.

Example

Array declarator:	`arr(2,0:1,-5:-4)`

Array storage:	`arr(1,0,-5)`
	`arr(2,0,-5)`
	`arr(1,1,-5)`
	`arr(2,1,-5)`
	`arr(1,0,-4)`
	`arr(2,0,-4)`

```
arr(1,1,-4)
arr(2,1,-4)
```

Arrays as Parameters

When arrays are passed as parameters, the size of the actual argument array must not exceed the size of the formal argument array. Because array bounds across separate compilation units are not checked at run-time, no warning is issued if the actual array size exceeds the formal array size. Altering these unreserved locations could yield unpredictable results.

Character Substrings

A character substring is a contiguous portion of a character variable.

Syntax

name ([*first*] : [*last*])

array (*s* [, *s*][. . .]) ([*first*] : [*last*])

Item	Description/Default	Restrictions
name	Character variable name.	None.
array(*s*[,*s*][...])	Character array element.	None.
first	Integer expression that specifies the leftmost position of the substring.	Default value is one.
last	Integer expression that specifies the rightmost position of the substring.	Default value is the length of the string.

The value of *first* and *last* must be such that:

`1` `<=` *first* `<=` *last* `<=` *len*

where *len* is the length of the character variable, named constant, or array element. The length of a substring is *last* `-` *first* `+ 1`.

Examples	Notes
`name(2:4)`	If the value of `name` is `'SUSANNA'`, then `name(2:4)` specifies `'USA'`.
`address(:4)`	If the value of `address` is `'1452 NORTH'`, then `address(:4)` specifies `'1452'`.
`city(6,2) (5:8)`	If the value of `city(6,2)` is `'SAN JOSE'`, then `city(6,2) (5:8)` specifies `'JOSE'`.
`title` or `title(:)`	These specify the complete character variable.

Records

A record is a collection of one or more data items called fields. The fields of a record can be of any type including records or structures. Like arrays, records can contain more than one data element. Unlike arrays, records can contain data elements of different types. An array element is identified with a unique index, a record element is accessed with a unique name.

Structure Declarations

Structure declarations are used to define the form of a record. The STRUCTURE statement specifies the name of the structure. The name is used by the RECORD statement to identify the structure that is used as the form for a record.

A structure establishes the size, shape, type, and names of the different fields. A structure declaration does not allocate storage. A record declaration establishes a memory reference using the declared structure. There can be more than one record for a given structure declaration.

A structure is declared with a statement block, starting with the STRUCTURE statement and ending with the END STRUCTURE statement. The following statements can be used within the block to define the structure:

- Data type declarations that define the fields within the structure.
- Substructures that are nested structure declarations.
- Mapped common area (union declarations).

Examples

The following example declares the structure `birth` which contains four fields: `day`, `month`, `birth_yr`, and `curr_yr`. Notice that `curr_yr` is initialized as 1989. Records defined with the structure `birth` will have the field `curr_yr` initialized to 1989.

```
STRUCTURE /birth/
LOGICAL*1 day, month
INTEGER*2 birth_yr
INTEGER*2 curr_yr /1989/
END STRUCTURE
```

In the following example, the structure `student` is declared. It contains the fields `name`, `sex`, `school_yr`, and `b_date`. Notice that `name` and `b_date` are also structures, or substructures within the structure `student`. The structure `name` is unnamed because it is not declared with slashes as `student` is declared. However, `name` does contain the fields `last`, `first`, and `middle`. The record `b_date` has the form of structure `birth`, the structure declared in the example above.

```
STRUCTURE /student/
STRUCTURE /name/
CHARACTER*20 last, first, middle
```

```
END STRUCTURE
CHARACTER*1 sex
LOGICAL*1 school_yr
RECORD /birth/ b_date
END STRUCTURE
```

Record Declarations

Records are comparable to variables and arrays. The name of a structure is used to define the data type of the record. The RECORD statement is similar to a type declaration, since it can define record scalars and arrays.

Example

The following example is based on the structure `student` (which has the structure `birth` used within it), shown in the previous section.

```
RECORD /student/ class(30)
```

The above RECORD statement creates an array of 30 records that have the structure `student`. In all 30 records, the field values are undefined except `b_date.curr_yr` which is initialized to 1989 in the `birth` structure declaration.

Record References

A field in a record is referenced by combining the record name and the field name with a period (.). If a record or a field is an array, its name can be subscripted.

Syntax

recspec [. *fieldspec*] [. . .]

Item	Description/Default	Restrictions
recspec	The name of a record. If it is an array, the name may be subscripted.	If the record is an array and a *fieldspec* is present, the *recspec* must include a subscript.
fieldspec	The name of a field within the record. If the field is an array, the name may be subscripted.	If the field is an array and a following *fieldspec* is present, the current *fieldspec* must include a subscript.

Examples

Using the record array `class`, which was defined in the previous section, the following variables can be specified:

`class(3).name.first(1:10)`	*A character substring*
`class(5).school_yr`	*A LOGICAL*1 element*
`class`	*An array*
`class(30)`	*A record*
`class(15).b_date`	*A record*

`class(15).b_date.birth_yr`	*An INTEGER*2 element*

Expressions

An expression can be a constant, simple or subscripted variable, function reference, substring, scalar record field reference, or a combination of operands, joined by arithmetic, character, logical, or relational operators. There are four types of expressions:

- Arithmetic
- Character
- Logical
- Relational

Arithmetic expressions return a single value of type INTEGER*2, INTEGER*4, REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16. Character expressions return character values. Relational and logical expressions evaluate to either true or false (a logical value).

Arithmetic Expressions

Arithmetic expressions perform arithmetic operations. An arithmetic expression can consist of a single operand or of one or more operands plus arithmetic operators, parentheses, or both. An arithmetic operand can be a numeric constant, the symbolic name of a numeric constant, an array element reference, scalar record field reference, or a function reference. As an extension to the ANSI 77 standard, an arithmetic operand can also be a logical variable or constant, depending upon compiler directives, as described in Chapter 7.

The arithmetic operators are:

Operator	**Meaning**
`+`	Addition; unary plus (positive or plus sign)
`-`	Subtraction; unary minus (negation or minus sign)
`*`	Multiplication
`/`	Division
`**`	Exponentiation

A unary operator affects one operand only. For example, the unary minus (also called a minus sign, or sign of negation) designates the expression following it to be negative.

The following are valid arithmetic expressions:

```
a
-4. + z
3.145
SQRT(r + d)
arr(5,2)*45.5
num(i)
.true. .XOR. foundit
a**2
(c**4)*d
total + sum_of_values
number_of_successes/number_of_tries*100
```

Multiplication must be specified explicitly. FORTRAN has no implicit multiplication that can be indicated by `a(b)` or `ab`; the form `a*b` must be used.

Hierarchy of Arithmetic Operators

The order of evaluation of an arithmetic expression is established by a precedence among the operators. This precedence determines the order in which the operands are to be combined. The precedence of the arithmetic operators is:

Operator	Rank	Meaning
`**`	Highest	Exponentiation
`* /`	⋮	Multiplication and division
`+ -`	Lowest	Addition and subtraction, unary plus and minus

Expressions within parentheses are evaluated first. Exponentiation precedes all arithmetic operations within an expression; multiplication and division occur before addition and subtraction.

For example, the expression:

```
-a**b + c * d + 6
```

is evaluated in the following order:

`a**b` is evaluated to form the operand `op1`.

`c*d` is evaluated to form the operand `op2`.

`-op1 + op2 + 6` is evaluated to form the result.

If an expression contains two or more operators of the same precedence, the following rules are applied:

- Two or more exponentiation operations are evaluated from right to left.
- Multiplication and division or addition and subtraction are evaluated from left to right.

The expression:

```
2**3**a
```

is evaluated in the following order:

`3**a` is evaluated to form `op1`.

`2**op1` is evaluated.

The expression:

```
a/b*c
```

is evaluated in the following order:

`a/b` is evaluated to form `op1`.

`op1*c` is evaluated.

The expression:

```
i/j + c**j**d - h*d
```

is evaluated in the following order:

`j**d` is evaluated to form `op1`.

`c**op1` is evaluated to form `op2`.

`i/j` is evaluated to form `op3`.

`h*d` is evaluated to form `op4`.

`op3 + op2` is evaluated to form `op5`.

`op5 - op4` is evaluated.

Parentheses can control the order of evaluation of an expression. Each pair of parentheses contains a subexpression that is evaluated according to the rules stated above. When parentheses are nested in an expression, the innermost subexpression is evaluated first.

The expression:

```
((a + b)*c)**d
```

is evaluated in the following order:

`a+b` is evaluated to form `op1`.

`op1*c` is evaluated to form `op2`.

`op2**d` is evaluated.

The expression:

```
((b**2 - 4*a*c)**.5)/(2*a)
```

is evaluated in the following order:

The subexpression `b**2 - 4*a*c` is evaluated to form `op1`.

`op1**.5` is evaluated to form `op2`.

`2*a` is evaluated to form `op3`.

`op2/op3` is evaluated.

Note

The actual order of evaluation may be different from that shown, but the result is the same as if the described order were followed (except when referencing functions that have side effects).

Consecutive Operators

As an extension to the ANSI 77 standard, consecutive operators in arithmetic expressions are allowed if the second operator is a unary plus (+) or minus (-).

The expression:

```
A ** - B * C
```

is evaluated in the following order:

`B` is negated to form `op1`.

`A**op1` is evaluated to form `op2`.

`op2*C` is evaluated.

The expression:

```
A + - B * - C
```

is evaluated in the following order:

`C` is negated to form `op1`.

`B*op1` is evaluated to form `op2`.

`op2` is negated to form `op3`.

`A+op3` is evaluated.

Expressions with Mixed Operands

Integer, real, and complex operands can be intermixed freely in an arithmetic expression. As an extension to the ANSI 77 standard, logical operands can be intermixed with numeric operands.

Before an arithmetic operation is performed, the lower type is converted to the higher type. The type of the expression is that of the highest type operand in the expression. Operand types rank from highest to lowest in the following order:

Data Type	**Rank**
COMPLEX*16	Highest
COMPLEX*8	
REAL*16	
REAL*8	
REAL*4	
INTEGER*4	
INTEGER*2	
LOGICAL*4	
LOGICAL*2	
LOGICAL*1	Lowest

An exception to the above is that, if one operand is REAL*8 and the other is COMPLEX*8, the result is COMPLEX*16. Another exception is that, if one operand is REAL*16 and the other is COMPLEX*8 or COMPLEX*16, the result is COMPLEX*16.

The conversion precedence for mixed type arithmetic expressions is described in Table 2-4. For example, if `a` and `b` are real variables and `i` and `j` are integer variables, then, in the expression `a*b-i/j`, `a` is multiplied by `b` to form the real value `op1`. Next, `i` is divided by `j` with integer division to form the integer value `op2`. Finally, `op2` is converted to real, and subtracted from `op1`, to produce a real result.

Table 2-4. Conversion of Mixed Type Operands

	L*1	**L*2**	**L*4**	**I*2**	**I*4**	**R*4**	**R*8**	**R*16**	**C*8**	**C*16**
L*1	L*1	L*2	L*4	I*2	I*4	R*4	R*8	R*16	C*8	C*16
L*2	L*2	L*2	L*4	I*2	I*4	R*4	R*8	R*16	C*8	C*16
L*4	L*4	L*4	L*4	I*4	I*4	R*4	R*8	R*16	C*8	C*16
I*2	I*2	I*2	I*4	I*2	I*4	R*4	R*8	R*16	C*8	C*16
I*4	I*4	I*4	I*4	I*4	I*4	R*4	R*8	R*16	C*8	C*16
R*4	R*4	R*4	R*4	R*4	R*4	R*4	R*8	R*16	C*8	C*16
R*8	R*8	R*8	R*8	R*8	R*8	R*8	R*8	R*16	C*16	C*16
R*16	R*16	R*16	R*16	R*16	R*16	R*16	R*16	R*16	C*16	C*16
C*8	C*8	C*8	C*8	C*8	C*8	C*8	C*16	C*16	C*8	C*16
C*16	C*16	C*16	C*16	C*16	C*16	C*16	C*16	C*16	C*16	C*16

Key to symbols in Table 2-4:

Symbol	**Stands for Data Type**
L*1	LOGICAL*1
L*2	LOGICAL*2
L*4	LOGICAL*4
I*2	INTEGER*2
I*4	INTEGER*4
R*4	REAL*4
R*8	REAL*8
R*16	REAL*16
C*8	COMPLEX*8
C*16	COMPLEX*16

When any value is raised to an integer power, the operation is performed by repeated multiplications. When any value is raised to a noninteger power, the operation is performed by logarithms and exponentiation.

Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each operand is an arithmetic constant, the symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. Variables, array elements, record field references, and function references are not allowed, with the following exception: As an extension to the ANSI 77 standard, some intrinsic functions are allowed in the PARAMETER statement.

Character Expressions

A character expression performs character operations. Evaluation of a character expression produces a result of type CHARACTER.

The simplest form of a character expression is a character constant, the symbolic name of a character constant, a character variable reference, a character array element reference, a character substring reference, a scalar record field reference of type CHARACTER, or a character function reference. More complicated character expressions can be formed by using two or more character operands together with the character operator. The character operator, concatenation, is formed by two slashes, `//`.

Syntax

c1 `//` *c2*

Item	Description/Default	Restrictions
c1, *c2*	Character expressions, or character entities as described above.	None.

The result of a concatenation operation is a character string whose value is the value of *c1* concatenated on the right with the value of *c2*. The length of the resulting string is the sum of the lengths of *c1* and *c2*. For example, the value of `'FOOT' // "BALL"` is the string `'FOOTBALL'`.

Parentheses have no effect on the value of a character expression. For example, the expression

```
'ab'//('CD'//'ef')
```

is the same as the expression

```
'ab' // 'CD' //'ef'
```

The result of either expression is `'abCDef'`.

Examples

```
char_string (5:9)
'constant string'
string1//string2//'another string'
home//'/'//filename
```

Character Constant Expressions

A character constant expression is a character expression in which each operand is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Variables, substrings, array elements, record field references, and function references are not allowed, with the following exception: As an extension to the ANSI 77 standard, some intrinsic functions are allowed in the PARAMETER statement.

Relational Expressions

Relational expressions compare the values of two arithmetic expressions or two character expressions. Evaluation of a relational expression produces a result of type logical.

Syntax

op1 relop op2

Item	Description/Default	Restrictions
op1, *op2*	Expressions	Must be either arithmetic or character
relop	Relational operator	None

The relational operators are:

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

Each relational expression is evaluated and assigned the logical value true or false depending on whether the relation between the two operands is satisfied (true) or not (false).

Note

Aggregate record references are not permitted in relational expressions.

Arithmetic Relational Expressions

Arithmetic expressions used as operands in a relational expression are evaluated according to the previously defined rules for arithmetic expressions. If the expressions are of different types, the one with the lower rank is converted to the higher ranking type as specified in Table 2-4. Once the expressions are evaluated and converted to the same type, they are compared. An arithmetic relational expression is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. If the operands do not satisfy the specified relation, the expression is interpreted as the logical value false. The following are valid arithmetic relational expressions:

```
a .GT. 237
a + b - c .LT. num
i + j .GE. z + 1
```

```
o .GT. p
```

Expressions of complex data types can be used as operands with `.EQ.` and `.NE.` relational operators only. The concept of less than or greater than is not defined for complex numbers.

Character Relational Expressions

Character relational expressions compare two operands, each of which is a character expression. The character expressions are evaluated; then the two operands are compared character by character, starting from the left. The initial characters of the two operands are first compared. If the initial character is the same in both operands, the comparison proceeds with the second character of each operand. When unequal characters are encountered, the greater of these two characters in ASCII value is the greater operand. Therefore, the ranking of the operands is determined by the first character position at which the two operands differ. If the operands do not differ at any position, the two operands are equal.

For example, when the two expressions `'PEOPLE'`and `'PEPPER'` are compared, the first expression is considered less than the second. This is determined by the third character `O`, which is less than `P` in the ASCII collating sequence. Refer to Appendix D for the ASCII collating sequence.

If the operands are of unequal length, the comparison is made as if the shorter string was padded with blanks on the right to the length of the longer string.

Examples	Notes
`IMPLICIT CHARACTER*6 (a-n)`	All variables beginning with the letters `a` to `n` are CHARACTER type.
`'the' .LT.'there'`	
`'MAY 23' .GT. 'MAY 21'`	
`name .LE. 'PETERSEN'`	
`char_stri .GE. char_str2`	
`first .EQ. a_string(2:8)`	

Logical Expressions

Logical expressions produce results of type logical with values of true or false. A logical expression can consist of a single operand or one or more operands plus a logical operator. A logical operand can be a logical constant, the symbolic name of a logical constant, a logical variable, a logical array element reference, a scalar record field reference of logical type, or a relational expression. As an extension to the ANSI 77 standard, integer variables or constants can also be used as logical operands, depending upon compiler directives. Refer to chapter 8 for further information.

The logical operators are:

Operator	Meaning
`.NOT.`	Logical negation (unary)
`.AND.`	Logical AND
`.OR.`	Logical inclusive OR
`.EQV.`	Logical equivalence
`.NEQV.`	Logical nonequivalence (same as `.XOR.`)
`.XOR.`	Logical exclusive OR (same as `.NEQV.`)

The unary operator `.NOT.` gives the complement (that is, the opposite) of the logical value of the operand immediately following the `.NOT.` operator.

The `.AND.` operator returns a value of true only if the logical operands on both sides of the `.AND.` operator evaluate to true.

The `.OR.` operator returns a value of true if one or both of the logical operands on either side of the `.OR.` operator are true.

The `.NEQV.` and `.XOR.` operators return a value of true only if one (but not both) of the logical operands on either side of the operator is true. As an extension to the ANSI 77 standard, `.XOR.` can be used in place of `.NEQV.`

The `.EQV.` operator returns a value of true if the logical operands on either side of the `.EQV.` operator are both true or both false.

Table 2-5 is a truth table for the logical operators.

Table 2-5. Truth Table for Logical Operators

a	b	.NOT. a	a .AND. b	a .OR. b	a .NEQV. b a .XOR. b	a .EQV. b
True	True	False	True	True	False	True
True	False	False	False	True	True	False
False	True	True	False	True	True	False
False	False	True	False	False	False	True

The order of evaluation of a logical expression is established by the following precedence of the logical operators:

`.NOT.`	highest
`.AND.`	
`.OR.`	
`.EQV. .NEQV. .XOR.`	lowest

If there is more than one operator of the same precedence, evaluation occurs from left to right.

Examples

The expression:

```
a .OR. b .AND. c
```

is evaluated in the following order:

`b .AND. c` is evaluated to form `lop1`.

`a .OR. lop1` is evaluated.

The expression:

```
z .LT. b .OR. .NOT. k .GT. z
```

is evaluated as follows:

`k .GT. z` is evaluated to form `lop1`.

`.NOT. lop1` is evaluated to form `lop2`.

`z .LT. b` is evaluated to form `lop3`.

`lop3 .OR. lop2` is evaluated.

The expression:

```
z .AND. d .OR. lsum(q,d) .AND. p .AND. i
```

is evaluated in the following order:

`z .AND. d` is evaluated to form `lop1`.

`lsum(q,d)` is evaluated to form `lop2`.

`lop2 .AND. p` is evaluated to form `lop3`.

`lop3 .AND. i` is evaluated to form `lop4`.

`lop1 .OR. lop4` is evaluated.

The expression:

```
a .AND. (b .AND. c)
```

is evaluated in the following order:

`b .AND. c` is evaluated to form `lop1`.

`a .AND. lop1` is evaluated.

As shown in the last example, parentheses can be used to control the order of evaluation of a logical expression. As with arithmetic expressions, the actual order of evaluation may be different from that stated above, but the result is the same as if these rules were followed.

Bit Masking Expressions

As an extension to the ANSI 77 standard, the logical operators can be used with integer operands to perform bit masking operations. You must be aware of the internal binary representations of the data in order to use the masking operators to produce predictable results. (Refer to Chapter 10 for details on data representation in memory.)

A complete truth table is shown in Table 2-6 (a version of Table 2-5 with true = 1 and false = 0). A bit by bit comparison is done of the operands (*i* and *j*), and the corresponding bit of the result is set according to the truth table.

FORTRAN also supplies these bit masking operations and other bit manipulation operations as intrinsic functions, described in Appendix B. These functions comply with the MIL-STD-1753 extension to the ANSI 77 standard.

Table 2-6. Truth Table for Masking Operators

i	j	.NOT. i	i .AND. j	i .OR. j	i .NEQV. j i .XOR. j	i .EQV. j
1	1	0	1	1	0	1
1	0	0	0	1	1	0
0	1	1	0	1	1	0
0	0	1	0	0	0	1

Examples

`.AND.` returns the logical product of two operands:

```
    op1:        0111111111111110 (32766₁₀)
    op2:        0001011001011001 (5721₁₀)

 result:        0001011001011000 (5720₁₀)
```

`.NEQV.` and `.XOR.` return the symmetric difference of two operands:

```
    op1:        0000000011111111 (255₁₀)
    op2:        0001011001011001 (5721₁₀)

 result:        0001011010100110 (5798₁₀)
```

3

FORTRAN Statements

Statements are the fundamental building blocks of FORTRAN program units. This chapter describes the general form of a statement and then discusses the different categories of statements. Following the general discussions are detailed descriptions of FORTRAN statements in alphabetical order. Each description includes the statement syntax, applicable rules, and examples.

FORTRAN Statement Format

A FORTRAN statement has the following general form:

[*label*] *statement*

The *label* identifies a particular statement so that it can be referenced from another portion of the program. A statement label consists of one to five digits placed anywhere in columns 1 through 5. Each label must be unique within a program unit; blanks and leading zeros are ignored by the compiler. Labels are optional and need not appear in numerical order.

Examples	**Notes**
`99999`	Largest label
`0300`	Identical labels
` 300`	
`30 0`	
`1`	Smallest label

The statement itself is written in columns 7 through 72. If a statement is too long for one line, it can be continued on the next line. This is indicated by placing a character other than a zero or a blank in column 6. Columns 1 through 5 of a continuation line are ignored, except that column 1 cannot contain the character `$`, `C`, `!`, or `*`. By default, each statement can have up to 19 continuation lines. If the CONTINUATIONS compiler directive is specified, each statement can have up to 99 continuation lines.

Statement Classification

A FORTRAN statement can be either executable or nonexecutable. Executable statements specify the actions that the program is to take. Nonexecutable statements contain information such as the characteristics of operands, type of data, and format specifications for input/output. Each FORTRAN statement is categorized in Table 3-1.

Table 3-1. Executable and Nonexecutable Statements

Executable Statements	Nonexecutable Statements
ACCEPT	BLOCK DATA
ASSIGN	BYTE
Assignment	CHARACTER
BACKSPACE	COMMON
CALL	COMPLEX (*8, *16)
CLOSE	DATA
CONTINUE	DIMENSION
DECODE	DOUBLE COMPLEX
DELETE	DOUBLE PRECISION
Block DO	END MAP
Labeled DO	END STRUCTURE
DO-WHILE	END UNION
ELSE IF	ENTRY
ELSE	EQUIVALENCE
ENCODE	EXTERNAL
END	FORMAT
END DO	FUNCTION
ENDFILE	IMPLICIT
ENDIF	INCLUDE
GOTO	INTEGER (*2, *4)
Arithmetic IF	INTRINSIC
Block IF	LOGICAL (*1, *2, *4)
Logical IF	MAP
INQUIRE	NAMELIST
OPEN	PARAMETER
PAUSE	PROGRAM
PRINT	REAL (*4, *8, *16)
READ	RECORD
RETURN	SAVE
REWIND	Statement Function
REWRITE	STRUCTURE
STOP	SUBROUTINE
TYPE	SYSTEM INTRINSIC
UNLOCK	UNION
WRITE	VIRTUAL
	VOLATILE

Executable and nonexecutable statements can be further grouped into seven functional categories, displayed in Table 3-2. The categories are:

- Program unit statements.

- Specification statements.
- Value assignment statements.
- Initialization statements.
- Control statements.
- Input/output statements.
- Program halt or suspension statements.

Table 3-2. Classification of Statements

Program Unit Statements	**Description**
BLOCK DATA	Identifies a program unit as a block data subprogram.
END	Identifies the end of a program unit.
ENTRY	Provides an alternative entry into a function or subroutine.
FUNCTION	Identifies a program unit as a function subprogram.
PROGRAM	Identifies a program unit as a main program.
Statement Function	Defines a one-statement function.
SUBROUTINE	Identifies a program unit as a subroutine subprogram.
Specification Statements	**Description**
COMMON	Reserves a block of memory that can be used by more than one program unit.
DIMENSION	Defines the dimensions and bounds of an array.
END MAP	Defines the end of a MAP statement group.
END STRUCTURE	Defines the end of a STRUCTURE statement group.
END UNION	Defines the end of a UNION statement group.
EQUIVALENCE	Associates variables so that they share the same place in memory.
EXTERNAL	Identifies subprogram names used as actual arguments or as nonintrinsics.
IMPLICIT	Specifies the type associated with the first letter of a symbolic name.
INTRINSIC	Identifies intrinsic function names used as actual arguments.
MAP	Identifies a group of statements that define the form of fields within a union.
NAMELIST	Defines a list of variables or array names and associates that list with a group-name.

(Continued on the next page)

Table 3-2. Classification of Statements (continued)

Specification Statements	**Description**
PARAMETER	Defines named constants.
RECORD	Defines records declared in a previous structure declaration.
SAVE	Retains the value of an entity after execution of a RETURN or END statement in a subprogram.
STRUCTURE	Begins a group of statements that defines the form of a record.
Type Specification	Assigns an explicit type to a variable.
UNION	Associates fields within a structure so that they occupy the same physical location in memory.
VIRTUAL	Defines the dimensions and bounds of an array; like DIMENSION.
VOLATILE	Identifies variables, arrays, or common blocks that will not be selected for global analysis or optimization by the compiler.
Value Assignment Statements	**Description**
ASSIGN	Assigns a statement label to an integer variable.
Assignment	Assigns values to variables at execution time.
Initialization Statements	**Description**
DATA	Assigns initial values to variables before execution.
Type Specification	Can optionally initialize variables before execution.
Control Statements	**Description**
CALL	Transfers control to an external procedure.
CONTINUE	Causes execution to continue; has no effect of its own.
Block DO, Labeled DO	Executes a group of statements a specific number of times.
DO-WHILE	Executes a group of statements while a condition is true.
END DO	Terminates a DO or DO-WHILE block.
ENDIF	Terminates an IF-THEN block.
ELSE	Marks the beginning of a block of statements to be executed if the logical expression in its corresponding IF-THEN statement evaluates to false.
ELSE IF	Same as an ELSE statement that has an IF-THEN statement as the first statement of its ELSE block.
GOTO	Transfers control to a specified statement.

(Continued on the next page)

Table 3-2. Classification of Statements (continued)

Control Statements	**Description**
Arithmetic IF	Transfers control based on a condition.
Block IF	Executes optional groups of statements based on one or more conditions.
Logical IF	Conditionally executes a statement based on a logical value.
RETURN	Transfers control from a subprogram back to the calling program.
Input/Output Statements	**Description**
ACCEPT	Transfers input data from the standard input unit to an internal storage area.
BACKSPACE	Positions a file at the previous record.
CLOSE	Terminates access to a file.
DECODE	Transfers data from internal storage to variables.
DELETE	Deletes an indexed sequential access (ISAM) record.
ENCODE	Transfers data from variables to internal storage.
ENDFILE	Writes an end-of-file.
FORMAT	Describes how input/output information is arranged.
INQUIRE	Supplies information about files.
OPEN	Allows access to a file.
PRINT	Transfers data out.
READ	Transfers data in.
REWIND	Positions a file at beginning-of-file.
REWRITE	Updates an indexed sequential access (ISAM) record.
TYPE	Transfers data out.
UNLOCK	Unlocks an indexed sequential access (ISAM) record.
WRITE	Transfers data out.
Program Halt Statements	**Description**
PAUSE	Causes a program suspension.
STOP	Terminates program execution.

Order of Statements

Statements are restricted as to where they can appear in a program unit. Within a program unit, the following rules apply:

- PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA statements can appear only as the first statement in a program unit.
- All specification statements must precede all statement function statements and executable statements.
- IMPLICIT statements must precede all other specification statements except PARAMETER statements.
- All statement function statements must precede all executable statements.
- FORMAT and ENTRY statements can appear anywhere.
- The last line of a program unit must be an END statement.

The required order of statements is shown in Figure 3-1.

Vertical lines delineate varieties of statements that can be interspersed. For example, DATA statements can be interspersed with statement function statements and executable statements.

Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

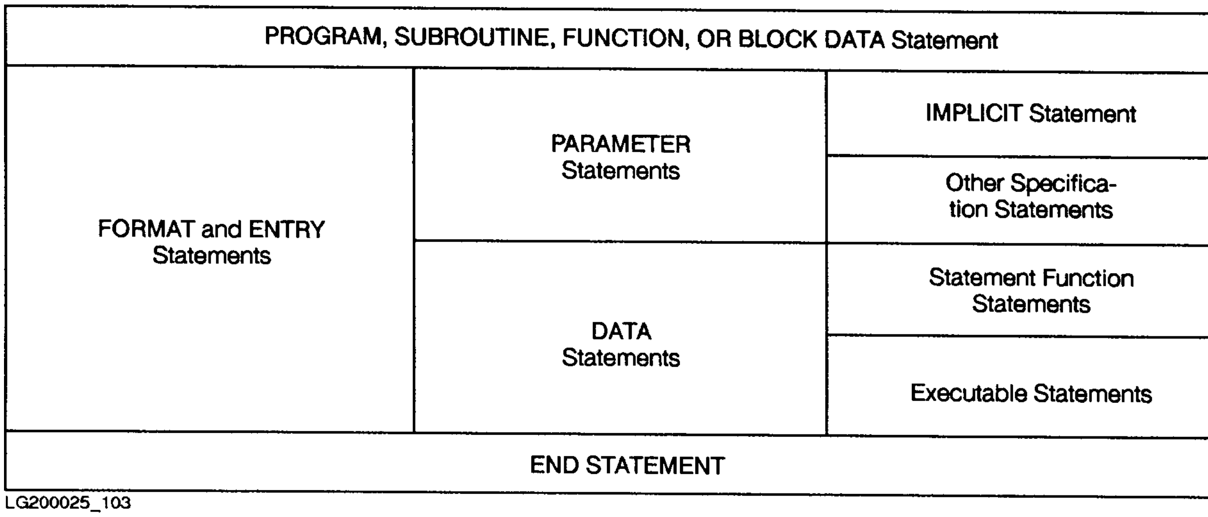
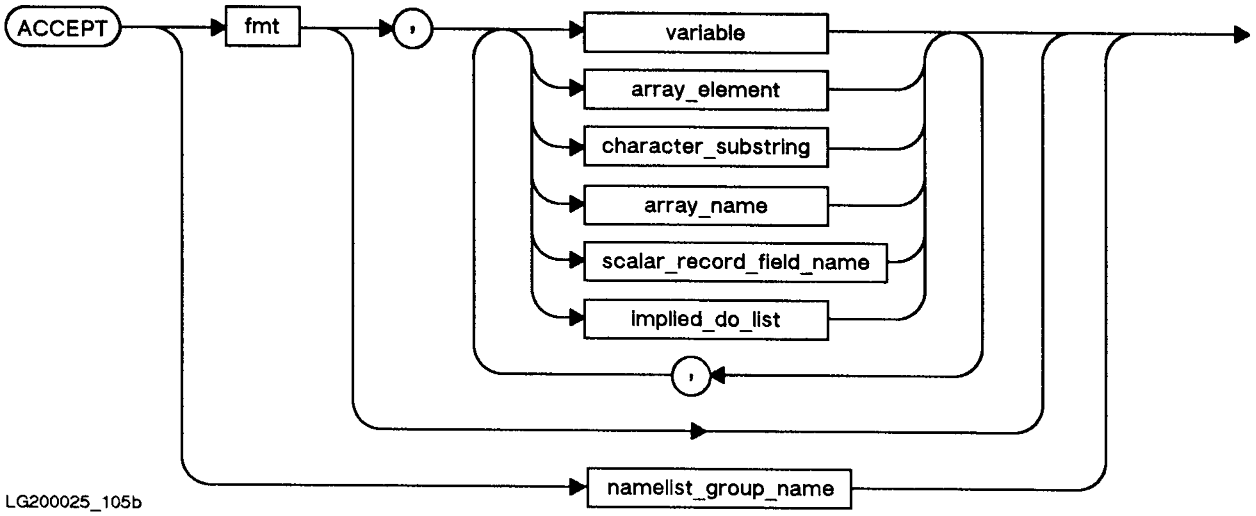

PROGRAM, SUBROUTINE, FUNCTION, OR BLOCK DATA Statement
FORMAT and ENTRY Statements
PARAMETER Statements
IMPLICIT Statement
Other Specification Statements
DATA Statements
Statement Function Statements
Executable Statements
END STATEMENT
LG200025_103

Figure 3-1. Required Order of Statements

ACCEPT Statement (Executable)

The ACCEPT statement transfers input data from the standard input unit to an internal storage area. Input data is transferred under sequential mode access. The ACCEPT statement cannot be connected to user-specified logical units.



Item	Description/Default	Restrictions
fmt	Format designator.	See "Semantics".
namelist_group_name	Symbolic name specifying a list of variables or arrays previously declared in a NAMELIST statement.	None.
variable *array_element* *character_substring* *array_name* *scalar_record_field_name*	Variable location where data is to be stored.	None.
implied_do_list	A list of variables in an implied DO loop. See "DO Statement (Executable)" for details.	None.

Semantics

The format designator, *fmt*, must be one of the following:

- The statement label of a FORMAT statement.
- An INTEGER*4 variable to which the statement label of a FORMAT statement has been assigned through an ASSIGN statement.
- A character or noncharacter array name that contains the representation of a format specification (use of a noncharacter array is an extension to the ANSI 77 standard).
- A character expression that evaluates to the representation of a format specification.
- An asterisk, which specifies list-directed input. See "List-Directed Input/Output" in Chapter 4.

The ACCEPT statement is an extension to the ANSI 77 standard.

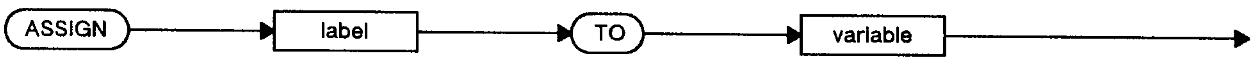
Examples	**Notes**

`ACCEPT 100,number,string`
`100 FORMAT(I3,A10)`
	`number` and `string` are input according to the FORMAT statement `100`.

`ACCEPT *,var1,var2,var3(1)`	`var1`, `var2`, and `var2` are input according to list-directed formating.
`ACCEPT NAME1`	The ACCEPT statement transfers input data to the entities associated with the `NAME1` namelist group.

ASSIGN Statement (Executable)

The ASSIGN statement assigns a statement label to an INTEGER*4 variable.



LG200025_005

Item	Description/Default	Restrictions
label	Statement label.	Must be the label of an executable statement or a FORMAT statement.
variable	INTEGER*4 simple variable.	None.

Semantics

The variable defined as a label by the ASSIGN statement can subsequently be used in an assigned GOTO statement or as the format specifier in an input/output statement.

A variable must be defined with a statement label value in order to be referenced in an assigned GOTO statement or be used as a format identifier in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way; that is, it should not be redefined with an assignment statement or used as a variable in an expression. Also, the variable cannot be passed to a subroutine or function and used within that program unit.

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

Examples	Notes
`ASSIGN 10 TO label1`	Assigns the statement label `10` to the variable `label1`.

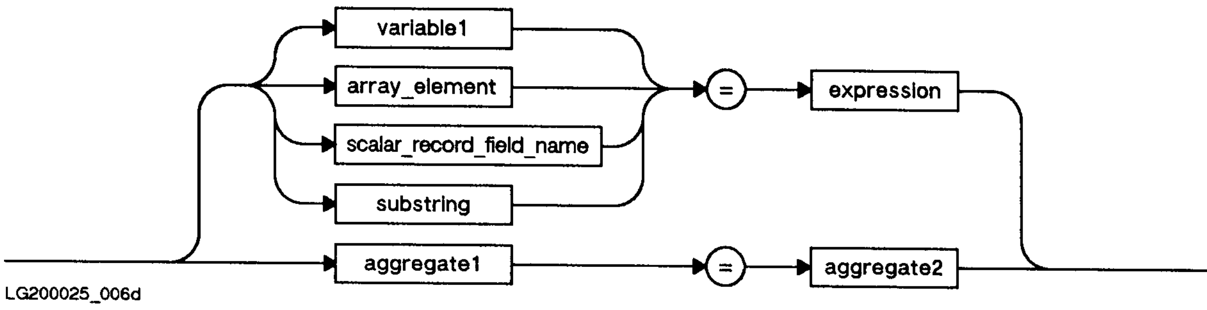
`ASSIGN 20 TO last1`
⋮
`GOTO last1`
	Assigns the statement label `20` to the variable `last1`. The label is that of an executable statement.

`ASSIGN 100 TO form1`
⋮
`100 FORMAT (F6.1,2X,I5/F6.1)`
⋮
`READ (5,form1) sum, ki, ave1`
	Assigns the statement label `100` to the variable `form1`. The label is that of a FORMAT statement.

Assignment Statement (Executable)

The assignment statement evaluates an expression and assigns the resulting value to a data item. There are four kinds of assignment statements:

- Arithmetic
- Logical
- Character
- Aggregate



Item	Description/Default	Restrictions
variable1 *array_element* *scalar_record_field_name* *substring*	Variable or array element.	If arithmetic assignment, type must be integer, real, or complex. If logical assignment, type must be logical. If character assignment, type must be character.
expression	The expression to which the variable or array element is being assigned.	If the variable or array element is arithmetic, expression must be an arithmetic expression; if logical, expression must of type logical; if character, expression must be of type character.
aggregate1 *aggregate2*	A record with one or more fields.	Both must be declared with the same structure.

Arithmetic Assignment Statement (Executable)

Semantics

If the type of the variable on the left of the equal sign differs from that of the expression, type conversion takes place. The expression is evaluated and the result is converted to the type of the variable on the left. The converted result then replaces the current value of the variable. Conversion rules for the assignment statement are shown in Table 3-3, followed by examples in Table 3-4. See also Table 2-4.

Table 3-3.
Type Conversion for Arithmetic Assignment Statements of the Form: Variable = Expression

Rule	Variable Type	Expression Type	Rules
a.	INTEGER**k* (or LOGICAL**k*) (see Note 3)	INTEGER**n*	If $k \geq n$, assign INTEGER**k*. If $k < n$, assign least significant byte or half word to variable. See Note 1.
b.	INTEGER**k* (or LOGICAL**k*) (see Note 3)	REAL**n*	Truncation.
c.	INTEGER**k* (or LOGICAL**k*) (see Note 3)	COMPLEX*8	Real part is REAL*4. Apply rule `b` to real part. Imaginary part is not used.
d.	INTEGER**k* (or LOGICAL**k*) (see Note 3)	COMPLEX*16	Real part is REAL*8. Apply rule `b` to real part. Imaginary part is not used.
e.	REAL**k*	INTEGER**n*	Float and assign REAL**k*. See Note 2.
f.	REAL**k*	REAL**n*	Round and assign REAL**k*.
g.	REAL**k*	COMPLEX*8	Real part is REAL*4. Apply rule `f` to real part. Imaginary part is not used.
h.	REAL**k*	COMPLEX*16	Real part is REAL*8. Apply rule `f` to real part. Imaginary part is not used.
i.	COMPLEX*8	INTEGER or REAL	Convert to REAL*4 by rule `e` or `f` and assign to real part. Imaginary part = 0.
j.	COMPLEX*16	INTEGER or REAL	Convert to REAL*8 by rule `e` or `f` and assign to real part. Imaginary part = 0.
k.	COMPLEX**k*	COMPLEX**n*	Apply rule `f` to real and imaginary parts.

Notes for Table 3-3

1. If the value of the expression is between −32768 and +32767, the result of the conversion from INTEGER*4 to INTEGER*2 is correct; otherwise, the result is incorrect.

2. When converting from INTEGER*4 to REAL*4, the precision can be lost because INTEGER*4 holds 31 significant bits, while the number of significant bits for REAL*4 is system dependent. Refer to Chapter 10 for more details on data representation.
3. As an extension to the ANSI 77 standard, logical variables appearing in an arithmetic context may be treated as integer variables, depending upon compiler directives.

In Table 3-4, *k* and *n* represent examples of possible combinations of byte sizes for the particular data type. For example, if the variable is INTEGER**k* and the expression is REAL**n*, *k* can be 2 or 4, while *n* can be 4, 8, or 16. This represents six possible combinations of byte sizes.

Table 3-4.
Examples of Type Conversions for Arithmetic Assignment Statements of the Form: Variable = Expression

Rule	Variable Type	Variable Value	Expression Value	Expression Type
a.	INTEGER*4 or LOGICAL*4	542	542	INTEGER*2
a.	INTEGER*2 or LOGICAL*2	Undefined (see Note 1).	86420	INTEGER*4
b.	INTEGER*2 or INTEGER*2	3	3.842	REAL*4
b.	INTEGER*2 or LOGICAL*2	373	373.7Q0	REAL*16
c.	INTEGER*2 or LOGICAL*2	502	(5.0297E2,1.27E−5)	COMPLEX*8
d.	INTEGER*4 or LOGICAL*4	−48170	(−4.817D4,1.0096D7)	COMPLEX*16
e.	REAL*4	59.	59	INTEGER*2
f.	REAL*8	10.D+09	10.E+09	REAL*4
f.	REAL*4	1.7014E+38	1.7014118344D+38	REAL*8
h.	REAL*4	8.425	(8.425,−6.02E−2)	COMPLEX*8
h.	REAL*8	2.2964D−8	(2.2964D−8,6.2881D−4)	COMPLEX*16
h.	REAL*16	3.57Q297	(3.57D297,−1.0D32)	COMPLEX*16
i.	COMPLEX*8	(50.0,0)	50	INTEGER*2
i.	COMPLEX*8	(25.0,0)	25	REAL*4
j.	COMPLEX*16	(14.23D−17,0.)	14.23E−17	REAL*4
j.	COMPLEX*16	(1.0D28,0.0D0)	1.00000000000000000000635Q28	REAL*16
k.	COMPLEX*8	(−4.817E4,1.0096E7)	(−4.817D4,1.0096D7)	COMPLEX*16
k.	COMPLEX*16	(8.425D0,_6.02D−2)	(8.425,−6.02E−2)	COMPLEX*8

Examples	Notes
`total = subtotal + tally`	Defines the value of `total` as the value of `subtotal + tally`.
`sum = sum + 1`	Replaces the value of `sum` with the value of `sum + 1`.
`rate(10) = new_rate * 5`	Defines the 10th element of the array `rate` as the value of `new_rate` multiplied by 5.

Logical Assignment Statement (Executable)

Semantics

Both the variable and the expression must logical types in a logical assignment statement.

Examples	Notes

`LOGICAL log1`
`i = 10`
`log1 = i .EQ. 10`
	`log1` is assigned the value `.TRUE.` because `i` equals 10.

`LOGICAL log_res, flag_set`
`num = 100`
`flag_set = .TRUE.`
`log_res = NUM .GT. 200 .AND. flag_set`
	`log_res` is assigned the value `.FALSE.` because `num` is not greater than 200.

Semantics

If the length of the variable is greater than the length of the expression, the value of the expression is left-justified in the variable, and blanks are placed in the remaining positions. If the length of the variable is less than the length of the expression, the value of the expression is truncated from the right until it is the same length as the variable.

Examples	Notes

`CHARACTER*6 name`
`CHARACTER*4 instrument(6),k`
`name = 'CYBELE'`
⋮
	The variable `name` is assigned the character string "`CYBELE`".

`k = 'horn'`
`instrument(5) = k`
⋮
	The fifth element of the array `instrument` is assigned the character string "`horn`".

`instrument(4) = name(3:5) // 'L'`	The fourth element of the array `instrument` is assigned the character string "`BELL`".

`CHARACTER*21 employee_name`
`employee_name = 'GEORGE WASHINGTON'`
	The variable `employee_name` is assigned the value "`GEORGEΔWASHINGTONΔΔΔΔ`".

`CHARACTER security_code*4`
`security_code = 'ZXYwvu'`
	The variable `security_code` is assigned the value "`ZXYw`".

`CHARACTER address*20`
`address (1:4) = '1645'`
`address(6:14) = 'First St.'`
	The first through fourth characters of the variable `address` are assigned the value "`1645`" and the sixth through fourteenth the value "`First St.`".

Semantics

The field values of the aggregate on the left side of the equal sign are assigned to the corresponding fields of the aggregate on the right side of the equal sign. The two aggregates must be declared with the same structure name.

Examples	Notes

`STRUCTURE/student/`
`CHARACTER*32 name`
`INTEGER*2   age`
`END STRUCTURE`
	A structure named `student` is declared with two fields, `name`, a 32-byte character type, and `age`, a 2-byte integer type.

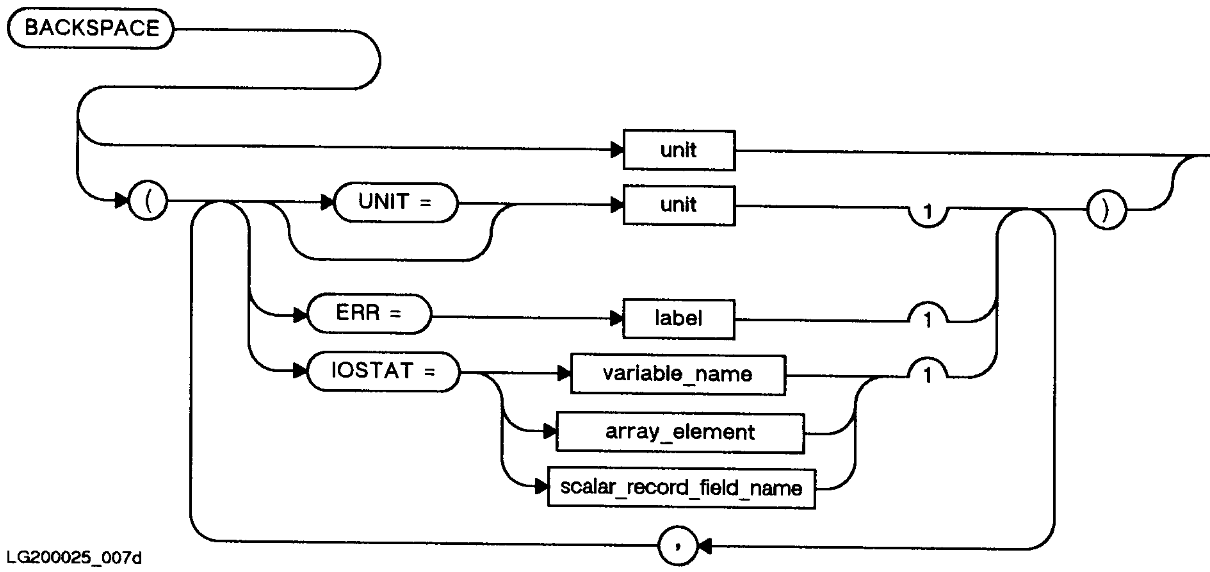
`STRUCTURE/teacher/`
`CHARACTER*32 name`
`INTEGER*2   age`
`END STRUCTURE`
	A structure named `teacher` is declared with two fields whose names and data types are identical to the structure `student`.

`RECORD/student/math_student`
`RECORD/student/english_student`
`RECORD/teacher/math_teacher`
	Two records named `math_student` and `english_student` are declared using the structure named `student`. One record named `math_teacher` is declared using the structure named `teacher`.

`math_student = english_student`	This is a valid aggregate assignment statement, since both variables were declared with the same structure name.
`math_student = math_teacher`	This is an *illegal* aggregate assignment statement, since the two variables were declared with different structure names, even though the two structures are identical in form.

BACKSPACE Statement (Executable)

The BACKSPACE statement positions a sequential file or device at the preceding record.



LG200025_007d

Item	Description/Default	Restrictions
unit	Expression giving the unit number of a connected file.	Must be a nonnegative integer.
variable_name *array_element* *scalar_record_field_name*	Error code return.	Must be an integer type.
label	Statement label.	Must be an executable statement in the same program unit as the BACKSPACE statement.

Semantics

If the prefix `UNIT=` is omitted and the unit specifier is present, *unit* must be the first item in the list.

If the ERR specifier is present and an error occurs during execution of a BACKSPACE statement, control transfers to the statement specified by *label* rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

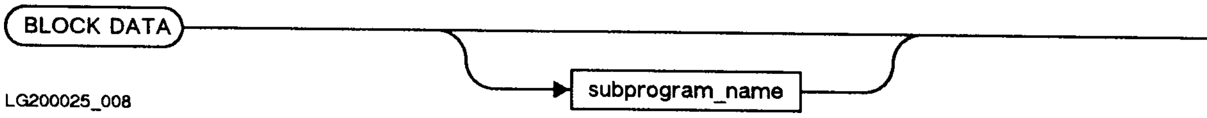
If the file is positioned at its beginning, a BACKSPACE statement has no effect.

As an extension to the ANSI 77 standard, BACKSPACE operations are allowed on files open for direct access. The file is positioned at the preceding record, provided it is not already at the beginning of the file.

Examples	Notes
`BACKSPACE 10`	The sequential file connected to unit 10 is backspaced one record.
`BACKSPACE (UNIT=k+3,IOSTAT=j,ERR=100)`	The file connected to unit `k+3` is backspaced one record. If an error occurs, control transfers to statement 100, and the error code is stored in variable `j`. If no error occurs, `j` is set to 0, and control transfers to the next statement.
`BACKSPACE (UNIT=k+3,IOSTAT=j)`	The file connected to unit `k+3` is backspaced one record. If an error occurs, the error code is stored in the variable `j`. If no error occurs, `j` is set to 0. In both cases, control transfers to the next statement.
`BACKSPACE (UNIT=k+3,ERR=100)`	The file connected to unit `k+3` is backspaced one record. If an error occurs, control transfers to statement 100. If no error occurs, control transfers to the next statement.

BLOCK DATA Statement (Nonexecutable)

The BLOCK DATA statement identifies the beginning of a block data subprogram.



Item	Description/Default	Restrictions
subprogram_name	Subprogram name.	Must not be the same as the name of an external procedure, the main program, a common block, or any other block data subprogram in the same execution program, nor can it be the same as any local name in this block data subprogram.

Semantics

Block data subprograms provide initial values for variables and array elements in labeled common blocks.

Block data subprograms define the size and reserve storage space for each labeled common block and, optionally, initialize the variables, records, and arrays declared in the common block. A block data subprogram begins with a BLOCK DATA statement. It ends with an END statement.

The BLOCK DATA statement must be the first noncomment statement in a block data subprogram. Each named common block referenced in an executable FORTRAN program can be defined in a block data subprogram.

Specification statements, data initialization statements, and blank common statements are allowed in the body of a block data subprogram. Acceptable statements include the COMMON, DATA, DIMENSION, END MAP, END STRUCTURE, END UNION, IMPLICIT, MAP, PARAMETER, RECORD, SAVE, STRUCTURE, and UNION statements, and all explicit type statements (such as INTEGER*4 or REAL*8). EXTERNAL and INTRINSIC statements are not allowed.

The block data subprogram without a name is treated the same as a named block data subprogram. However, there cannot be more than one unnamed block data subprogram in the same program. If a program contains more than one unnamed block data subprogram, the results are unpredictable.

Example	Notes

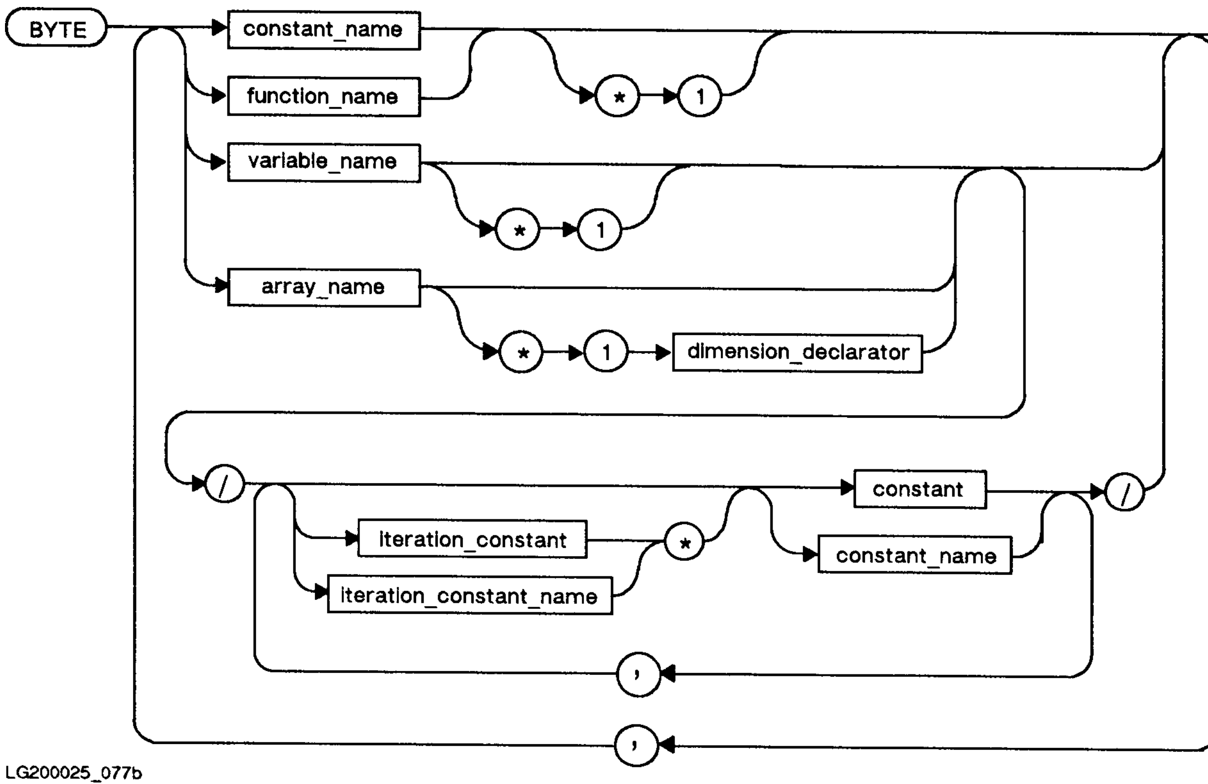
`BLOCK DATA myblock`
`COMMON /xxx/x(5),b(10),c`
`COMMON /set1/iy(10)`
`DATA iy/1,2,4,8,16,32,64,128,256,512/`
`DATA b/10*1.0/`
⋮
`END`
	`myblock` is the optional name of a block data subprogram to reserve storage locations for the named common blocks `xxx` and `set1`. Arrays `iy` and `b` are initialized in the DATA statements shown. The remaining elements in the common block can optionally be initialized or typed in the block data subprogram.

`BLOCK DATA name1`
`COMMON a,b,c`
`DATA a,b,c/10.0,20.0,30.0/`
`END`
	`name1` is the optional name of a block data subprogram to reserve storage locations for the blank common block. The real variables `a`, `b`, and `c` are initialized as shown in the DATA statement.

BYTE Statement (Nonexecutable)

The BYTE type specification statement explicitly assigns the BYTE (LOGICAL*1) data type to symbolic names, and optionally assigns initial values to variables.

See also "LOGICAL Statement (Nonexecutable)".



LG200025_077b

Semantics

The BYTE statement is an extension to the ANSI 77 standard. The BYTE statement has the same effect as the LOGICAL*1 statement.

If an item being declared in a BYTE statement is an array name with a dimension declarator, the length specifier precedes the dimension declarator.

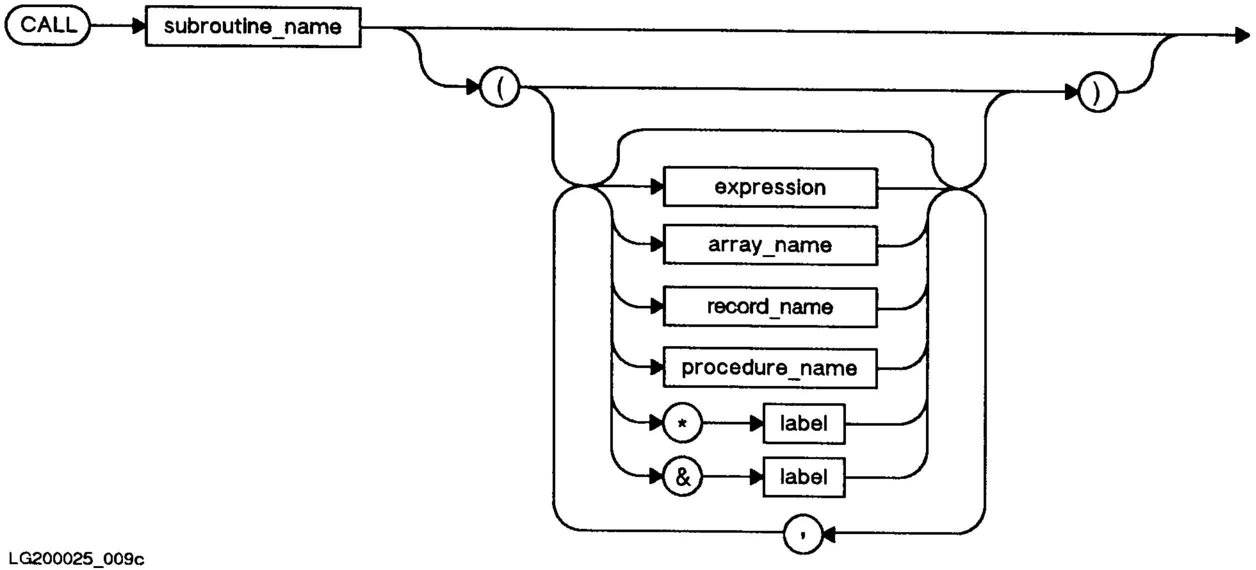
As an extension to the ANSI 77 standard, you can initialize variables or arrays in a type specification statement by enclosing the initialization values between slashes. The following examples illustrate this method of initialization:

```
BYTE I*1/25/,J/.TRUE./
BYTE array(10)/10*.FALSE./
```

See "DATA Statement (Nonexecutable)" for further information on initializing variables and arrays.

CALL Statement (Executable)

The CALL statement transfers control to a subroutine.



Item	Description/Default	Restrictions
* &	Alternate return.	None.
label	Statement label of an executable statement.	Must be in the same program unit as the CALL statement.

Semantics

When a CALL statement is executed, any expressions in the actual argument list are evaluated and control passes to the subroutine. For a normal return from the subroutine, execution continues with the statement following the CALL statement. When an alternate return is taken, execution continues with the statement label in the actual argument list that corresponds to the return number specified in the subroutine's RETURN statement.

The use of & as an alternate return is an extension to the ANSI 77 standards.

Examples	Notes
`CALL print_forms(top,lh,rh)`	Calls the subroutine `print_forms`. Passes three arguments.
`CALL exit`	Calls the subroutine `exit`. Passes no arguments.

`CALL test_data (m,n,val,*10)`
⋮
`10 total = val + 6.34`
⋮
`END`
 	
`SUBROUTINE test_data (j,k,w,*)`
⋮
`RETURN 1`
⋮
`END`
	Calls the subroutine `test_data`. Passes three arguments. `*10` means that the return point is the statement labeled 10 if the subroutine executes the alternate return `RETURN 1`.

As an extension to the FORTRAN 77 standard, a call can be made with arguments missing. The compiler passes a zero by value for any missing arguments. For example, the statement:

```
CALL fun (a,,b,,)
```

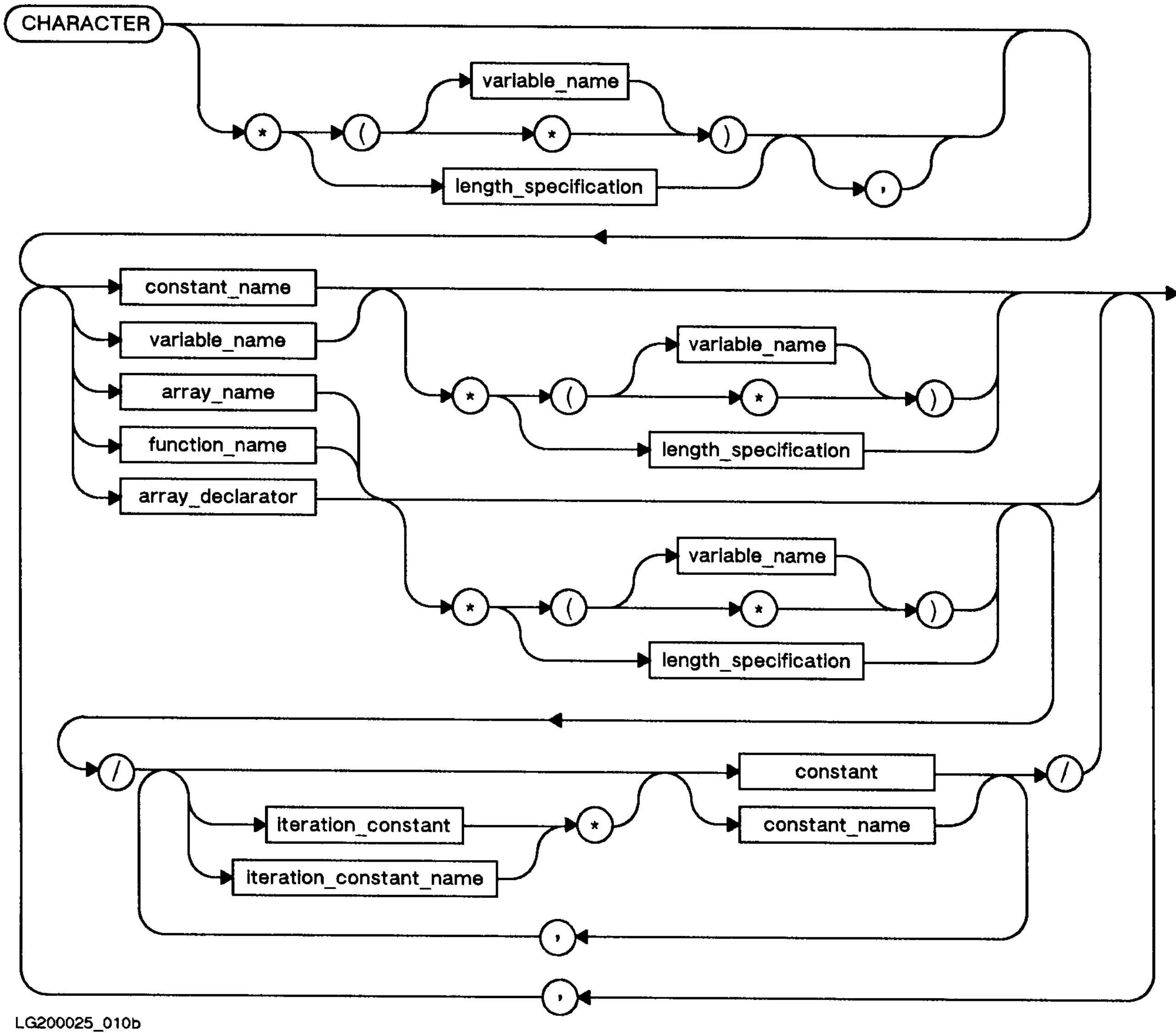
is equivalent to:

```
CALL fun (a,0,b,0)
```

with the zero being passed by value.

CHARACTER Statement (Nonexecutable)

The CHARACTER type specification statement explicitly assigns the CHARACTER data type to symbolic names, and optionally assigns initial values to the variables.



LG200025_010b

Semantics

If an array declarator is specified in a type statement, the declarator for that array must not appear in any other specification statement (such as DIMENSION). If only the array name is specified, an array declarator must appear within a DIMENSION or COMMON statement.

The length specification can be one of the following:

- An unsigned nonzero integer constant.

- An integer constant expression with a positive value. The integer expression must be enclosed in parentheses and cannot contain variable names. Example: `(-3 + 4)`.
- An asterisk enclosed in parentheses: `(*)`.
- A variable enclosed in parentheses. This is an extension to the ANSI 77 standard.

A length specification can be appended to the end of a symbolic name to designate its length. If the length specification is a variable, it must be a formal argument to the program unit in which the character type is declared. The variable must also be an integer type, and it must contain a positive value. If the length specification is omitted, the length is assumed to be one.

The `CHARACTER*(*)` form can be used only for named constants, formal arguments, function subprograms, or entry names in functions.

Note

If the symbolic name has an appended length specification, the specification overrides the length *n* in the CHARACTER**n* specification. Therefore, `CHARACTER*6 q`, `CHARACTER q*6`, and `CHARACTER*10 q*6` are all equivalent. Each reserves space for a character variable named `q` of length 6. Also, `CHARACTER*(*) name` and `CHARACTER name*(*)` are equivalent.

Examples	Notes
`CHARACTER*5 name(6)*10,zip(6)`	The variables `name` and `zip` are character arrays with six elements each. Each element in `name` has a length of 10. Each element in `zip` has a length of 5.

`CHARACTER*6 var`
`CALL sub (var)`
⋮
`SUBROUTINE sub (var1)`
	The variable `var` is defined as type CHARACTER and as six characters long.

`CHARACTER*(*) var1`	The variable `var1` is defined as being of type CHARACTER and as having the same length as the variable `var` in the calling subroutine.

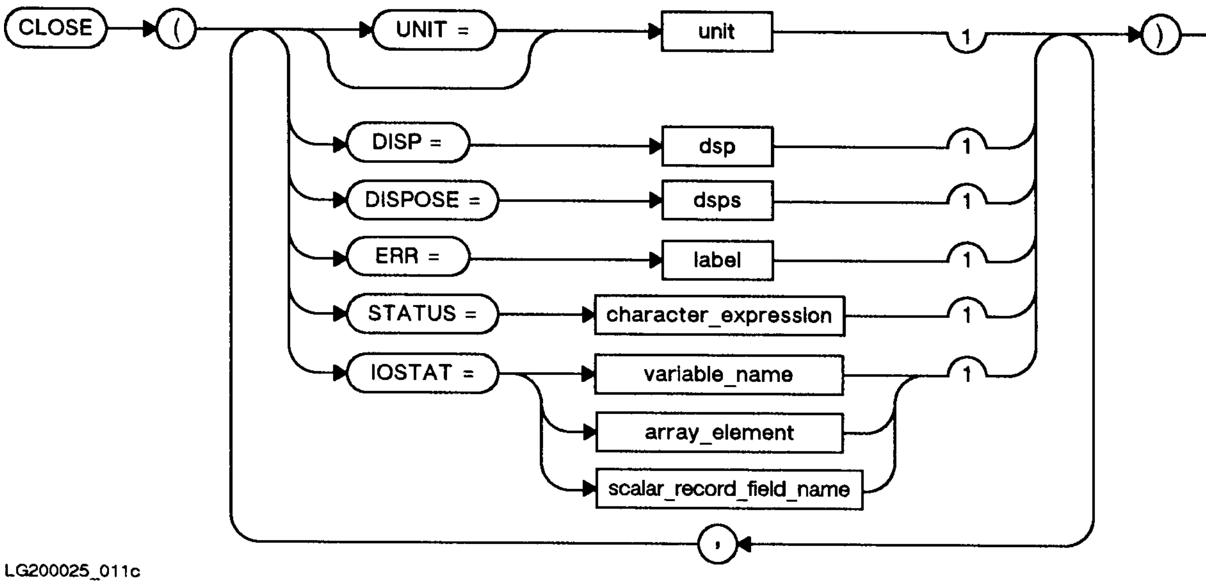
As an extension to the ANSI 77 standard, you can initialize variables or arrays in a type specification statement by enclosing the initialization values between slashes. The following examples illustrate this method of initialization:

```
CHARACTER*10 string/'0123456789'/
CHARACTER*25 stringtab(10)*10/10*'THIS IS IT'/,name*4/'BILL'/
```

See “DATA Statement (Nonexecutable)” for further information on initialization.

CLOSE Statement (Executable)

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



LG200025_011c

Item	Description/Default	Restrictions
unit	Specifies the unit number of the file.	Must be an integer expression.
dsp	See "Semantics".	None.
dsps	See "Semantics".	None.
label	Statement label of an executable statement.	Must be in the same program unit as the CLOSE statement.
character_expression	Character expression that determines the disposition of the file.	The value must be either `'KEEP'` or `'DELETE'`.
variable_name *array_element_name* *scalar_record_field_name*	Error code return.	Must be an integer type.

Semantics

If the ERR specifier is used and an error occurs during execution of a CLOSE statement, control transfers to the statement specified by *label* rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

If `STATUS='KEEP'` is specified, the file continues to exist after execution of the CLOSE statement. `'KEEP'` is the default for named files; that is, specifying `STATUS='KEEP'` or not specifying the STATUS parameter has the same effect. However, `STATUS='KEEP'` is an error if a scratch file is being closed.

The `STATUS='KEEP'` specifier is not allowed on scratch files because scratch files are deleted upon execution of CLOSE or at normal program termination.

In the STATUS specifier, only the first character is significant.

If `STATUS='DELETE'` is specified, the file does not exist after execution of the CLOSE statement.

The DISP and DISPOSE specifiers, which are extensions to the ANSI 77 standard, are included for compatibility with programs originally written in another version of FORTRAN. If used in a program, their syntax is checked, but they are otherwise ignored by the compiler.

A CLOSE statement must contain a unit number and at most one each of the other options.

A CLOSE statement need not be in the same program unit as the OPEN statement that connected the file to the specified unit. If a CLOSE statement specifies a unit that does not exist or has no file connected to it, no action occurs.

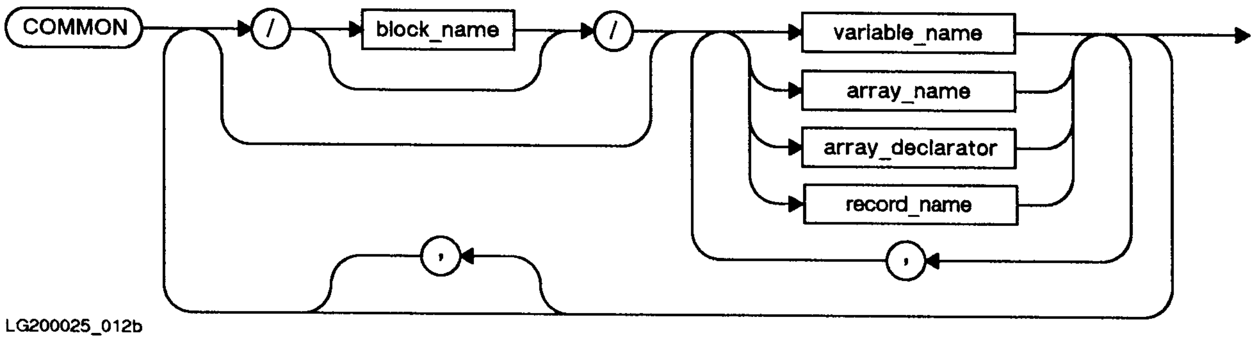
Examples	Notes
`CLOSE (10)`	Disconnects the file connected to unit 10. The file continues to exist.
`CLOSE (UNIT=6,STATUS='DELETE')`	Disconnects the file connected to unit 6. The file no longer exists.

`CHARACTER*6 cstat`
`cstat = 'DELETE'`
`CLOSE (UNIT=6,STATUS=cstat)`
	This produces the same results as the preceding statement.

`CLOSE (5,IOSTAT=io_error,ERR=100)`	Disconnects and keeps the file connected to unit 5. If an error occurs, control is transferred to statement 100 and the error code is stored in the variable `io_error`. If no error occurs, `io_error` is set to 0 and control transfers to the next statement.
`CLOSE (5,IOSTAT=io_error)`	Disconnects and keeps the file connected to unit 5. If an error occurs, the error code is stored in the variable `io_error`. If no error occurs, `io_error` is set to 0. In both cases, control transfers to the next statement.
`CLOSE (5,ERR=100)`	Disconnects and keeps the file connected to unit 5. If an error occurs, control transfers to statement 100. If no error occurs, control transfers to the next statement.

COMMON Statement (Nonexecutable)

The COMMON statement specifies a block of storage space that can be used by more than one program unit.



LG200025_012b

Item	Description/Default	Restrictions
block_name	Name of a labeled common block.	Must be different from all intrinsic names, subroutine names, and the program name.

Semantics

The following data items cannot appear in a COMMON statement:

- The names of formal arguments in a subprogram.
- A function, subroutine, or intrinsic function name.

As an extension to the ANSI 77 standard, a COMMON statement can contain a name that has been initialized in a DATA statement or a type statement.

A variable cannot be specified more than once in the COMMON statements within a program unit.

Each omitted block name specifies blank common, which means that a common block name is not specified but the compiler supplies a hidden name. The appearance of two slashes (//) with no block name between them declares the variables that follow to be in blank common.

A common block name or blank common specification can appear more than once in one or more COMMON statements in a program unit. The variable list following each successive appearance of the same common block name is treated as a continuation of the list for that block name.

For example, the COMMON statements:

```
COMMON a,b,c/x/y,z,d//w,r
COMMON /cap/hat,visor,//tax,/x/o,t
```

are equivalent to the following COMMON statement:

```
COMMON a,b,c,w,r,tax,/x/y,z,d,o,t,/cap/hat,visor
```

The length of a common block is determined by the number and type of the variables in the list associated with that block.

The total size of a COMMON block must be less than one gigabyte. (A gigabyte is 1,073,741,824 (2^{30}) bytes.)

Common block storage is allocated at link time. It is not local to any one program unit.

Note Global data declared in a COMMON block cannot be shared between a calling program and a subprogram in an executable library.

Each program unit that uses the common block must include a COMMON statement that contains the block name (if a name was specified). The list assigned to the common block by the program unit need not correspond to any other program unit by name, type, or number of elements. The only consideration is the size of the common blocks referenced by the different program units. The size of an unlabeled (blank) common block can differ between program units, but the size of a labeled common block should be the same in all program units.

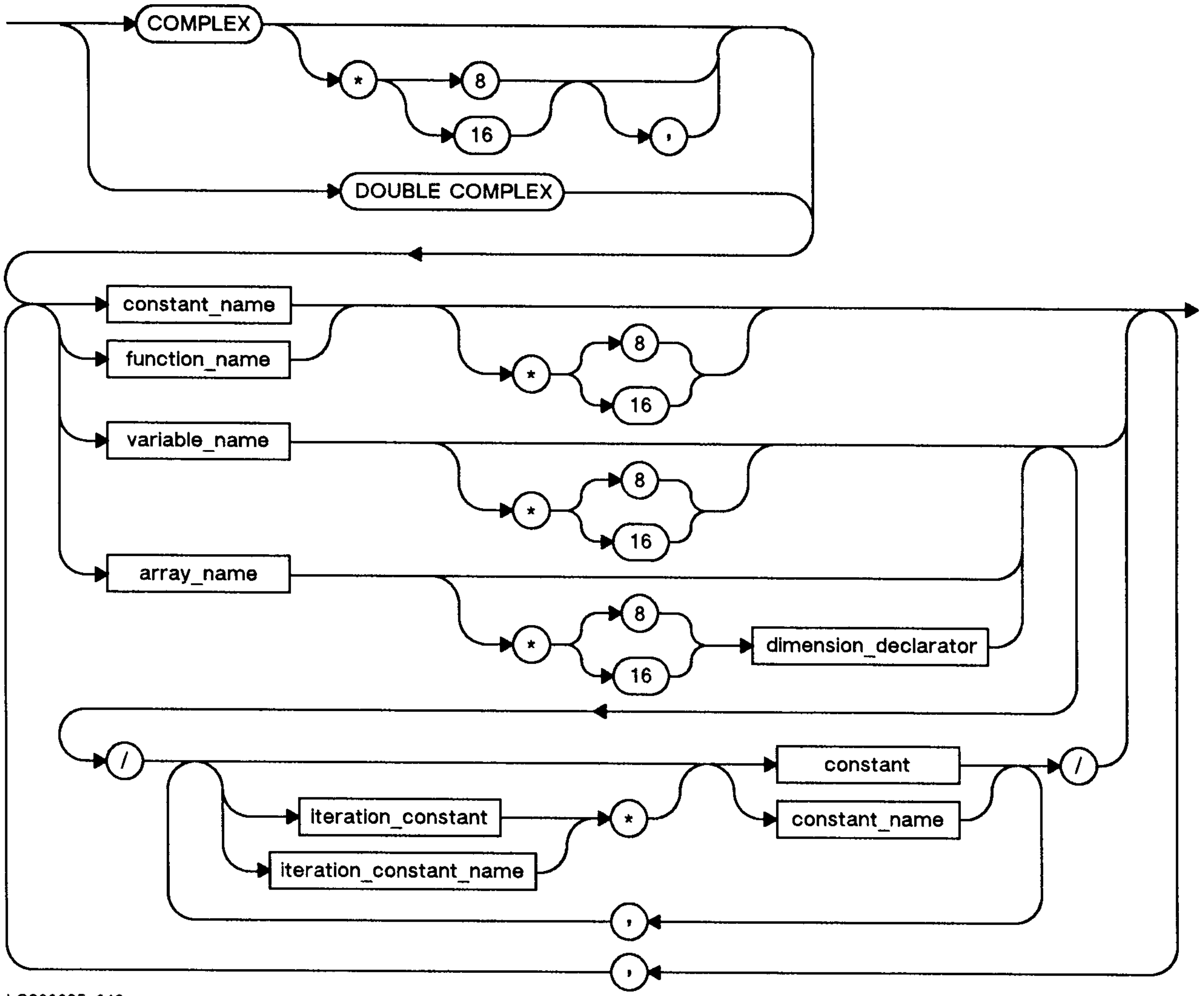
As an extension to the ANSI 77 standard, character and noncharacter data can be mixed in a given common block.

Examples	Notes
`COMMON a, b, c`	The variables `a`, `b`, and `c` are placed in blank common.
`COMMON pay, time, /color/red`	The variables `pay` and `time` are placed in blank common; the variable `red` is placed in common block `color`.
`COMMON /a/a1,a2,//x(10),y,/c/d`	The variables `a1` and `a2` are placed in common block `a`; `x(10)` and `y` are placed in blank common; `d` is placed in common block `c`.

COMPLEX Statement (Nonexecutable)

The COMPLEX type specification statement explicitly assigns the COMPLEX*8 and COMPLEX*16 data types to symbolic names, and optionally assigns initial values to variables.

The following syntax includes the COMPLEX, COMPLEX*8, COMPLEX*16, and DOUBLE COMPLEX statements.



LG200025_013c

Semantics

The COMPLEX*8 statement and the COMPLEX statement are equivalent. The COMPLEX*16 statement and the DOUBLE COMPLEX statement are equivalent. The COMPLEX*8 and COMPLEX*16 statements are extensions to the ANSI 77 standard.

As an extension to the ANSI 77 standard, a length specifier can follow the item being declared. This specifier overrides the data length implied by the type statement. If the item is an array name with a dimension declarator, the length specifier precedes the dimension declarator. The following example illustrates this syntax:

```
COMPLEX*16 a*8(10)
```

As an extension to the ANSI 77 standard, you can initialize variables or arrays in a type specification statement by enclosing the initialization values between slashes. The following example illustrates this method of initialization:

```
COMPLEX*8 num*16/(138.16,124.16)/
```

See "DATA Statement (Nonexecutable)" for further information on initialization.

COMPLEX*8 Statement (Nonexecutable)

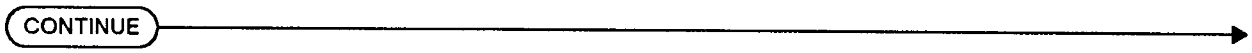
The COMPLEX*8 statement, which is an extension to the ANSI 77 standard, is a special case of the COMPLEX statement. See "COMPLEX Statement (Nonexecutable)" for details.

COMPLEX*16 Statement (Nonexecutable)

The COMPLEX*16 statement, which is an extension to the ANSI 77 standard, is a special case of the COMPLEX statement. See "COMPLEX Statement (Nonexecutable)" for details.

CONTINUE Statement (Executable)

The CONTINUE statement creates a reference point in a program unit.



LG200025_014

Semantics

The CONTINUE statement should always be written with a label; it marks a point in the program where a label is needed but the programmer does not want to associate the label with any specific action.

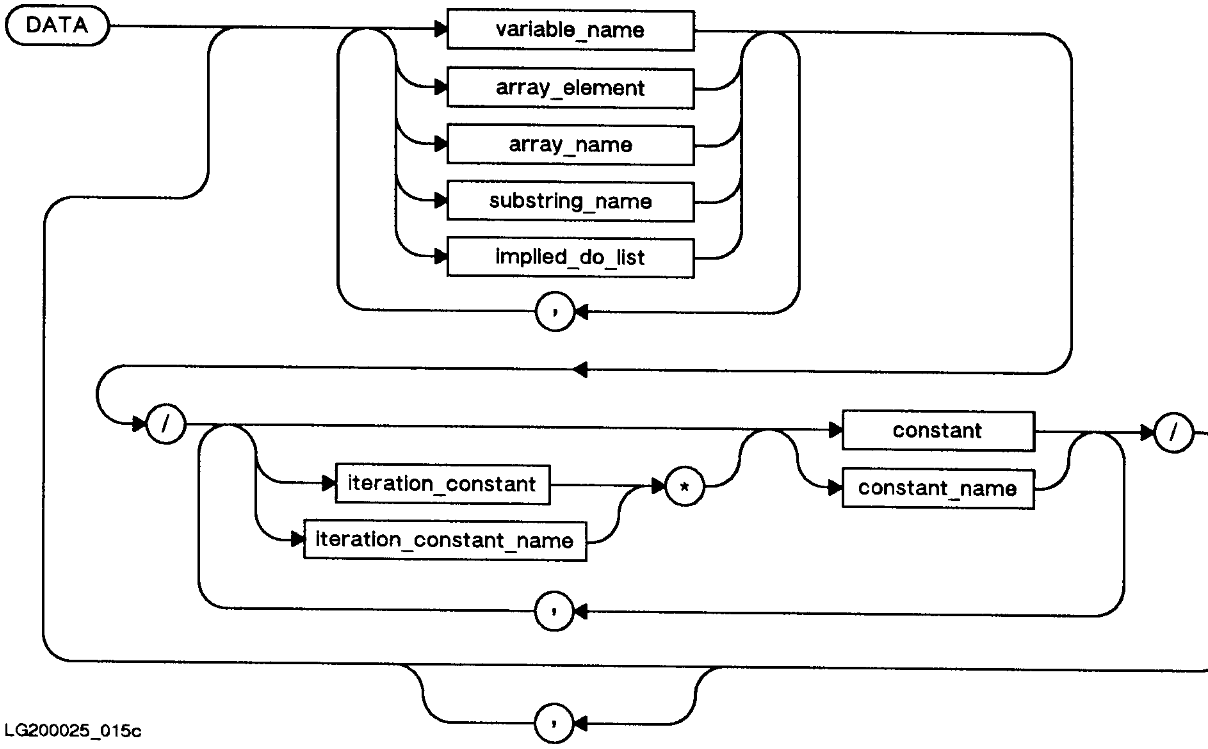
In programs in earlier versions of FORTRAN, the CONTINUE statement is usually the last statement in a labeled DO loop that otherwise would end in a prohibited statement such as a GOTO statement. As a MIL-STD-1753 standard extension to the ANSI 77 standard, the END DO statement now serves this purpose. (See "DO Statement (Executable)" for examples of the END DO statement.) If a CONTINUE statement appears elsewhere in a program or if it is not labeled, it performs no function and control passes to the next statement.

Examples	Notes

```
   DO 20 i = 1,10
10 x = x + 1
   y = SQRT(x)
   PRINT *, y
   IF (x .LT. 25.) GOTO 20
   GOTO 10
20 CONTINUE
```
	Because the last statement in the loop is a GOTO statement, a CONTINUE statement terminates the loop.

DATA Statement (Nonexecutable)

The DATA statement assigns initial values to variables before execution begins.



Item	Description/Default	Restrictions
iteration_constant	Nonzero unsigned integer. The default is one.	None.
iteration_constant_name	Named constant, defined by a PARAMETER statement, representing a nonzero unsigned integer. The default is one.	None.
*	Repeat specifier.	None.

Semantics

The number of items in the constant list must agree with the number of variables in the variable list. Each subscript in an array element in the variable list must be an integer or short integer constant expression, or an integer or short integer expression containing only constants and implied DO variables. If the variable list contains an array name without a subscript, one constant must be specified for each element of that array. The elements in the constant list are associated with the elements of the array in column-major order.

The assignment of constants in a DATA statement to their corresponding variables follows the rules of the assignment statement. (See "Assignment Statement (Executable)" for details.)

If a constant has a type of character or logical, the corresponding variable must be of the same type. Character variables can be initialized with octal, hexadecimal or hollerith constants as well. If a constant is of any numeric type (integer, real, or complex), the corresponding variable can be of any numeric type. Type conversion occurs automatically among the various numeric types, as in assignment statements.

The length of a character constant and the declared length of its corresponding character variable do not have to be the same. If the constant is shorter than the variable, the variable is filled on the right with blank characters. If the constant is longer than the variable, the constant is truncated, losing characters from the right.

Any local variable except a record can be initialized in a DATA statement, before execution of the program. A local variable mentioned in a DATA statement is treated the same way as one specified in a SAVE statement. Labeled common variables can also be initialized in a DATA statement within a block data subprogram.

The total size of local variables in a single subroutine must be less than one gigabyte. (A gigabyte is 1,073,741,824 (2^{30}) bytes.)

A variable or array element must not appear in a DATA statement more than once because a variable is initialized only once. If two variables (numeric or character) share the same storage space through the EQUIVALENCE statement, only one can appear in a DATA statement.

Each subscript expression in a DATA statement must be an integer constant expression, except for implied DO loop variables.

The use of octal and hexadecimal constants in DATA statements are MIL-STD-1753 standard extensions to the ANSI 77 standard.

DATA statements can be placed anywhere after specification statements in a program unit.

DATA statements cannot be used in procedures contained in executable libraries.

Examples	Notes
`DATA a,b,c,d/3.0,3.1,3.2,3.3/`	The values 3.0, 3.1, 3.2, and 3.3 are assigned to `a`, `b`, `c`, and `d`, respectively.

`DIMENSION i(3)`
`DATA i/3*2/`
	All three elements of `i` are assigned an initial value of 2.

`DIMENSION i(3)`
`DATA i(1)/2/i(2)/2/i(3)/2/`
	All three elements of `i` are assigned an initial value of 2. Equivalent to the previous example.

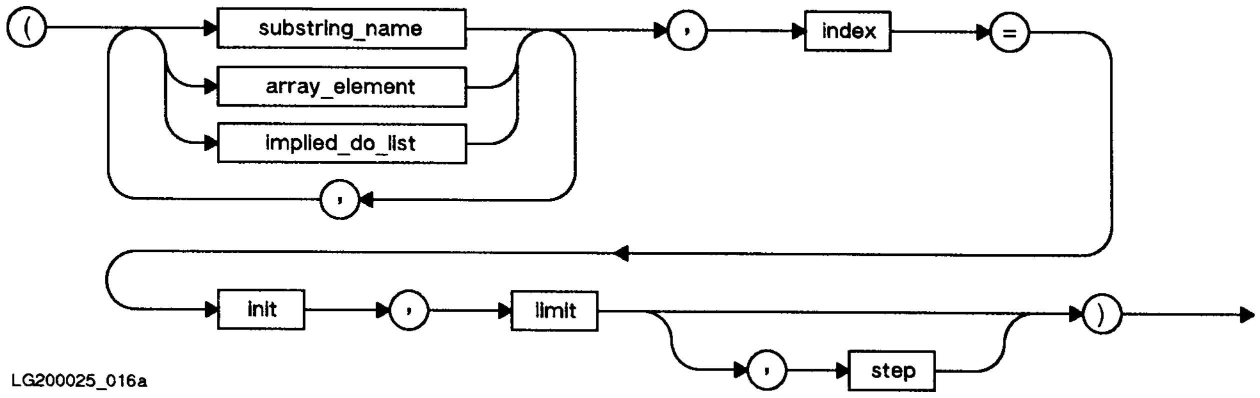
`DIMENSION i(3)`
`DATA i(1),i(2),i(3)/2,2,2/`
	All three elements of `i` are assigned an initial value of 2. Equivalent to the previous two examples.

`PARAMETER (init_val = -1)`
`DIMENSION m(10)`
`DATA m/10*init_val/`
	Each element of `m` is assigned an initial value of `init_val` that is a named constant previously defined in a PARAMETER statement.

Implied DO Loops in DATA Statements

The implied DO loop in a DATA statement acts like the implied DO loop in an input/output statement. It is executed at compilation time to initialize parts of arrays or generate a full variable list.

The format of an implied DO loop in a DATA statement is shown below:



Item	Description/Default	Restrictions
implied_do_list	Data list in the form of an implied DO loop.	Can contain other nested implied DO loops.
index	Variable that controls the number of times the elements in *substring_name*, *array_element_name*, or *implied_do_list* are read or written.	None.
init	Integer expression that is the initial value given to *index* at the start of the execution of the implied DO loop.	*init* can use only constants and the indexes of outer loops.
limit	Integer expression that is the termination value for *index*.	*limit* can use only constants and the indexes of outer loops.
step	Integer expression that is the increment by which *index* is changed after each execution of the DO loop; *step* can be positive or negative. Its default value is one.	*step* can use only constants and the indexes of outer loops. *step* should not equal 0. If it does, an error occurs, as setting *step* equal to 0 can cause the loop to be skipped entirely.

The index can be used in expressions for subscript values or position specifiers of character substrings. Inner implied DO loops can use the indexes of outer loops.

The iteration count in an implied DO loop in a DATA statement must be positive.

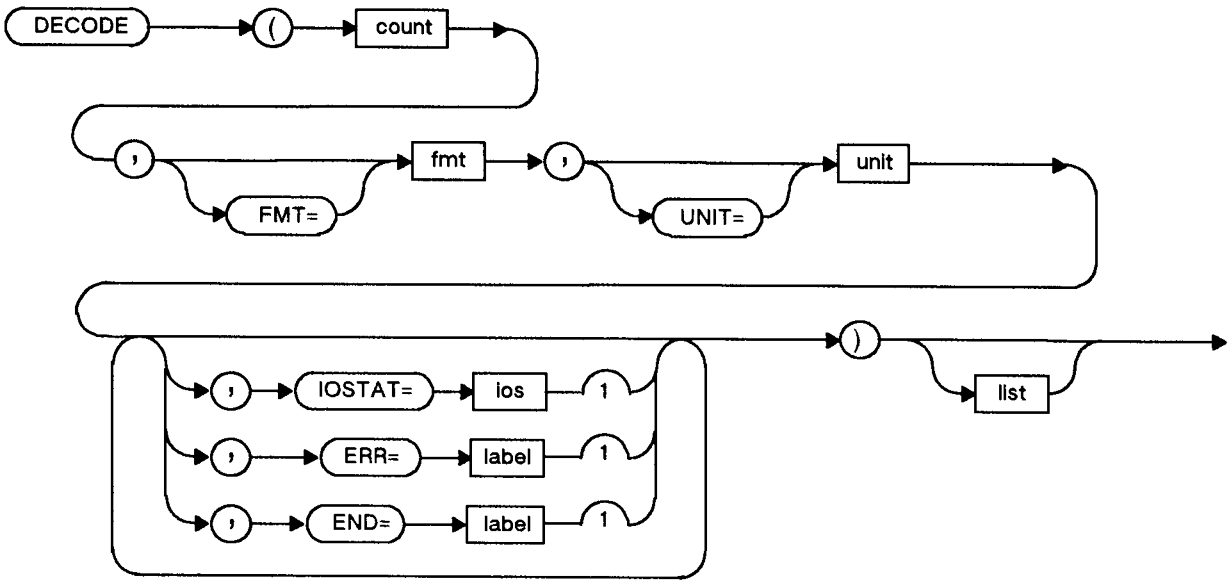
Examples	Notes

`DIMENSION i(3)`
`DATA (i(k),k=1,3)/3*2/`
	An implied DO loop assigns an initial value of 2 to all three elements of `i`. Equivalent to the previous three examples.

`CHARACTER k(10,5)`
`DATA ((k(i,j),j=1,5),i=1,10)/50*'x'/`
	Two nested implied DO loops assign the literal character `x` to each element in an array of 50 elements, `k(10,5)`.

DECODE Statement (Executable)

The DECODE statement transfers data from internal storage to variables according to a format specification.



LG200025_071a

Item	Description/Default	Restrictions
count	Integer expression that specifies the number of bytes to be translated.	Must be the first item.
fmt	Format designator.	Must be the second item. Must be as specified for the PRINT Statement.
unit	Internal storage designator.	Must be the third item. Must be a scalar record field name or array name. Assumed-size and adjustable-size arrays are not permitted.
ios	Integer variable or array element for error code return.	None.
label	Statement label of an executable statement. If an error occurs or an end-of-file is detected during execution of the DECODE statement, control transfers to the statement specified by this label rather than aborting the program.	None.
list	List specifying the data to be transferred.	Each list item must be a variable name, an array element, an array name, a substring, or a scalar record field name

Semantics

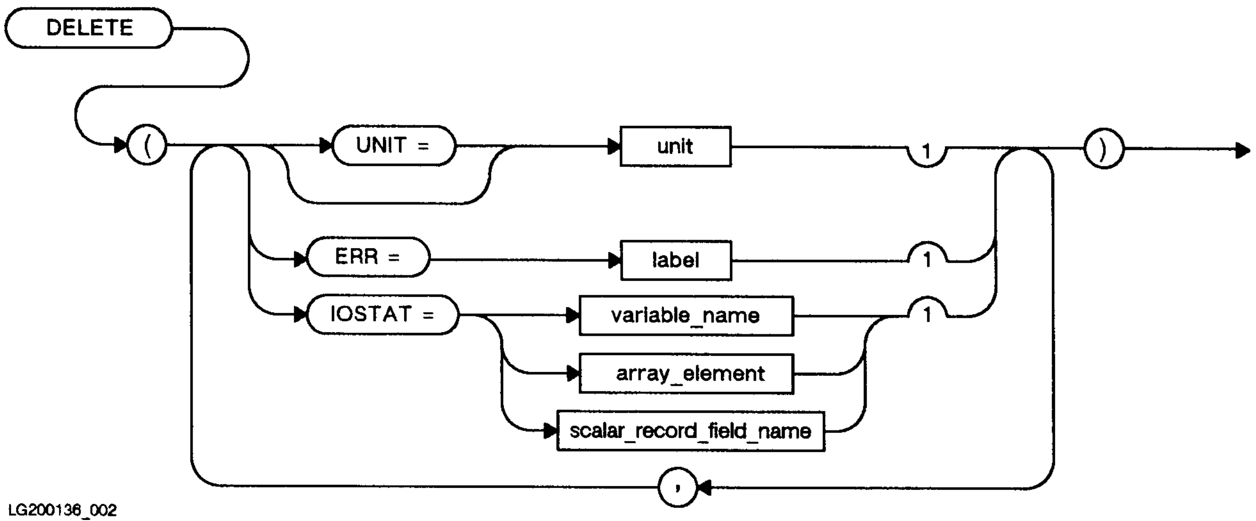
The DECODE statement is provided for compatibility with older versions of FORTRAN. It is a nonstandard statement and its use in new programs is strongly discouraged. If possible, use the internal file capabilities of the READ statement instead.

Examples	Notes

`CHARACTER*20 buf`
`INTEGER i, j, k`
`BUF = 'XX1234 45 -12XXXXXX'`
`DECODE (15,'(2x,3I4,1X)', buf) i,j,k`
	`i`, `j`, and `k` receive the values 1234, 45, and −12.

DELETE Statement (Executable)

The DELETE statement deletes a record from an indexed sequential access (ISAM) file.



LG200136_002

Item	Description/Default	Restrictions
unit	Expression specifying unit number of a connected file.	Must be a nonnegative integer.
variable_name *array_element* *scalar_record_field_name*	Error code return.	Must be an integer type.
label	Statement label.	Must be an executable statement in the same program unit.

Semantics

In the most recent operation on the file, the record to be deleted must have been read with a READ statement. If the most recent operation was not a read of a record, a run-time error occurs.

If the prefix `UNIT=` is omitted and the unit specifier is present, *unit* must be the first item in the list.

If the ERR specifier is present and an error occurs during execution of the DELETE statement, control transfers to the specified statement rather than aborting the program.

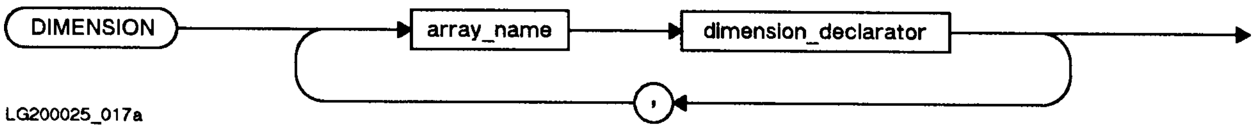
If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

Examples	Notes

`READ (10,key='111-22-333',KEYID=0) employee_rec`
`DELETE (10,ERR=555, IOSTAT=I)`
	Deletes a record with the key value `111-22-333`.

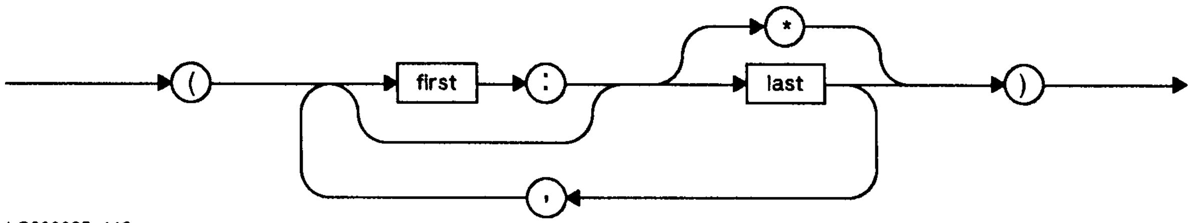
DIMENSION Statement (Nonexecutable)

The DIMENSION statement defines the dimensions and bounds of arrays.



LG200025_017a

Dimension Declarator



LG200025_112

Item	Description/Default	Restrictions
array_name	Symbolic name of the array.	None.
first	Lower dimension bound. Defaults to 1.	Must be less than or equal to *last*.
last	Upper dimension bound.	Must be greater than or equal to *first*.
*	Dynamic upper bound.	Can only appear in the last upper bound of a formal argument of a subprogram.

Semantics

There must be one dimension declarator for each dimension in the array.

When an array is defined in a DIMENSION statement, only the name of the array, not the complete declarator, can be used in a type or COMMON statement.

Only the upper dimension bound of the last dimension in an array declarator of a formal argument can be an asterisk.

As an extension to the ANSI 77 standard, an array can have an unlimited number of dimensions.

See "Array Declarators" in Chapter 2 for further details.

Examples	Notes

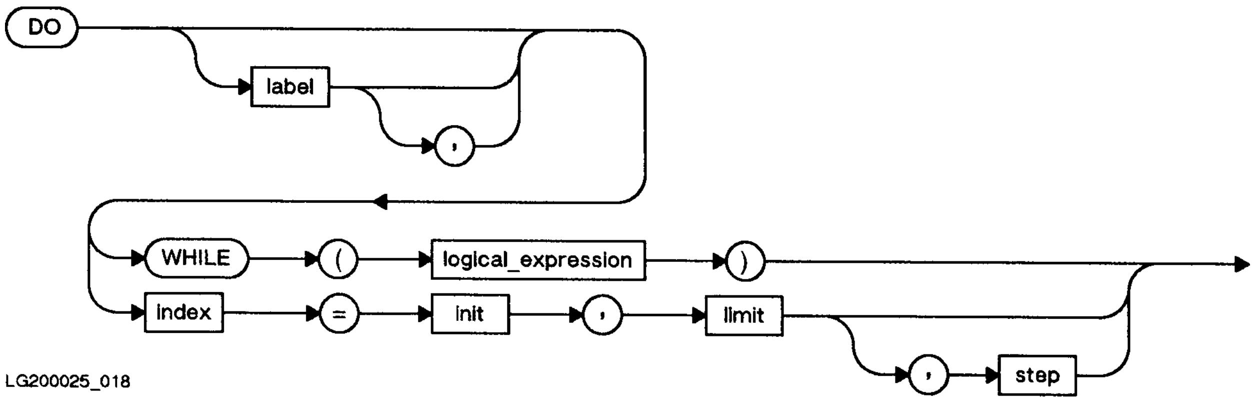
`INTEGER*2 arri`
`DIMENSION arri(-3:1,4)`
	The type statement specifies **arri** as a two-byte integer. Only the name of the array is used, not the complete array declarator.
	
	The DIMENSION statement causes 20 words of memory to be allocated for the array **arri**. An equivalent type specification would be:
	
	`INTEGER*2 arri(-3:1,4)`

`COMPLEX num(5,5)`
`DIMENSION num(5,5)`
	This construct is illegal because **num** is declared as an array twice.

DO Statement (Executable)

A DO statement defines the beginning of a DO loop. DO loops are groups of statements that are executed repeatedly zero or more times, or a list within one statement that is executed a specified number of times.

The maximum level to which DO statements, or a mixture of IF and DO statements, can be nested is 20. Exceeding this number makes programs unnecessarily complicated and can cause internal compiler errors stating that the statement is too complicated.



Item	Description/Default	Restrictions
label	Statement label of an executable statement.	Referenced statement terminates the DO loop.
index	Loop control variable.	Must be a simple variable of an integer or real data type.
init	Expression that is the initial value for *index*.	Must be an integer or real expression.
limit	Expression that is the termination value for *index*.	Must be an integer or real expression.
step	Expression that is the increment for *index* after each execution of the DO loop. *step* can be positive or negative. Its default value is one.	Must be an integer or real expression. *step* should not equal 0. If it does, an error occurs if the RANGE directive is ON. If RANGE is OFF, an infinite loop could result.

Semantics

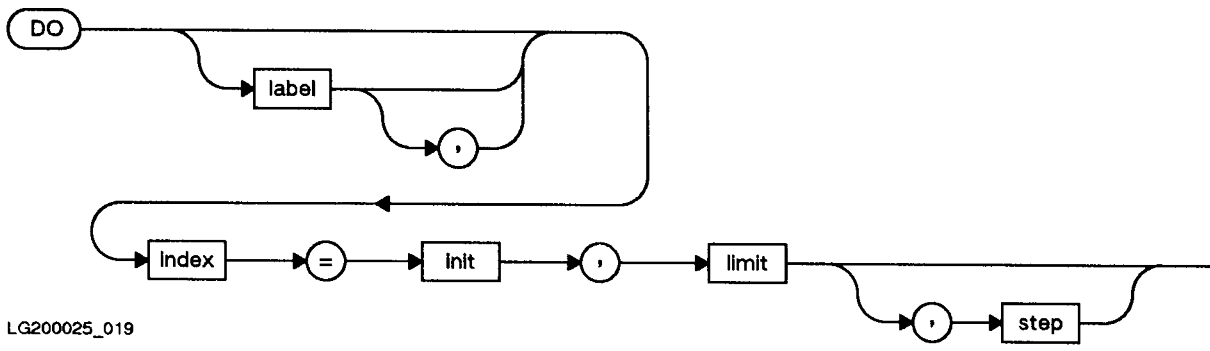
DO loops are grouped into four categories:

- Labeled DO loops
- Block DO loops
- Implied DO loops
- DO-WHILE loops

A labeled or block DO loop executes a group of statements a specified number of times. An implied DO loop is similar to a labeled DO loop, but it is used in a READ, WRITE, PRINT, or DATA statement. A DO-WHILE loop executes a group of statements while a specified condition is true.

Labeled and Block DO Loops

The labeled and block DO statements control execution of groups of statements by causing the statements to be repeated a specified number of times. The DO statement defines this repetition, or loop. The repeated statement or group of statements is known as the *range* of the DO loop.



LG200025_019

Semantics

In a labeled DO loop, the statement with the label must follow the DO statement in the sequence of statements within the same program unit.

In a block DO loop, the label is omitted and a following END DO statement terminates the loop. The block DO loop is an extension to the ANSI 77 standard.

init, *limit*, and *step* are indexing parameters as well as arithmetic expressions. *index*, *init*, *limit*, and *step* should all be of the same type. If they are not, *init*, *limit*, and *step* are converted to the same type as *index*. This can sometimes produce unexpected results, as shown in the examples.

Labeled DO Loop

A labeled DO loop begins with a DO statement that specifies the label of the terminating statement of the loop. The terminating statement of a labeled DO loop must follow the DO statement. It must not be one of the following:

- Another DO statement
- A DO-WHILE statement
- An assigned GOTO statement
- An unconditional GOTO statement
- An arithmetic IF statement
- Any of the four statements associated with the block IF statement:
  - IF-THEN statement
  - ELSE statement
  - ELSE IF statement
  - ENDIF statement
- A RETURN statement
- A STOP statement
- An END statement
- Any nonexecutable statement

The terminating statement of a labeled DO loop can be a logical IF statement.

A labeled DO loop can be terminated with an END DO statement. As in all labeled DO loops, this terminating END DO statement must have a label that matches the label of the DO statement.

Examples	Notes

`    DO 100 i = 1,10`
`      ⋮`
`100 CONTINUE`
	The group of statements terminating with the one labeled 100 is repeated 10 times.

`    DO 200 j = 1,10,2`
`      ⋮`
`200 IF (a(j) .EQ. 0) STOP`
	The group of statements terminating with the one labeled 200 is repeated five times.

`    DO 300 r = 1.0,2.0,.1`
`      ⋮`
`300 END DO`
	The group of statements terminating with the one labeled 300 is repeated 11 times. Although this loop ends with an END DO statement, it is not a block DO loop. Notice that the label in the DO statement corresponds with the one on the END DO statement.

`    DO 10 i = 1,10,2`
`     WRITE (6,'("i =",I2)') i`
`     i = i-2`
` 10 CONTINUE`
	Error. Attempted modification of `i` in this example produces a compilation error.

Block DO Loop

A block DO loop, an extension to the ANSI 77 standard, functions the same as a labeled DO loop. It differs in not using a label in its DO statement. Each block DO loop must be terminated with an END DO statement, which does not require a label.

Block DO loops can be nested (as described in "Nesting DO Loops" later in this section), but each level of nesting must be terminated by a separate END DO statement.

Examples	Notes

`DO j = 10,1,-2`
`⋮`
`END DO`
	Block DO loop. The group of statements terminating with the END DO statement is repeated five times.

`DO j = 10,1,2`
`⋮`
`END DO`
	Block DO loop. The group of statements terminating with the END DO statement is not executed. (The DO loop is skipped entirely unless the ONETRIP directive is ON.)

DO Loop Execution

When a DO statement is executed, the following actions occur:

1. *limit* and *step* are evaluated, then *index* and *init* are evaluated. If necessary, *init*, *limit*, and *step* are converted to the same type as *index*. The value of *init* is assigned to *index*.
2. If the number of times the loop would execute is negative or zero, the loop is skipped and control transfers to the statement following the termination statement of the DO loop. This occurs when *init* exceeds *limit* and *step* is positive, or *init* is less than *limit* and *step* is negative.
3. The range of the loop is executed.
4. *index* is incremented by the value of *step*. Note that this is done before testing if the loop has been executed the correct number of times.
5. If the loop has been executed fewer than the correct number of times, steps 3 through 5 are repeated.

Within the range of a DO loop, modification of *init*, *limit*, or *step* does not affect the number of iterations of the loop, because these values are established when the loop is entered. Modification of *index* within the range of the loop is not permitted. Refer to the examples above under "Labeled DO Loop" for such an attempt.

Upon normal completion of the DO loop, the value of the control variable is defined to be the next value assigned as a result of the incrementation. For example, in the loop:

```
DO i=1,5
   ⋮
END DO
```

the value of `i` after normal completion of the loop is 6.

In the loop:

```
DO i=1,10,3
   ⋮
END DO
```

the value of `i` after normal completion of the loop is 13.

Upon premature exit from the DO loop, the control variable retains its value at the time of exit.

Implied DO Loop

Implied DO loops are found in input/output statements (READ, WRITE, and PRINT) and in DATA statements. An implied DO loop contains a list of data elements to be read, written, or initialized, and a set of indexing parameters.

Inner loops can use the indexes of outer loops.

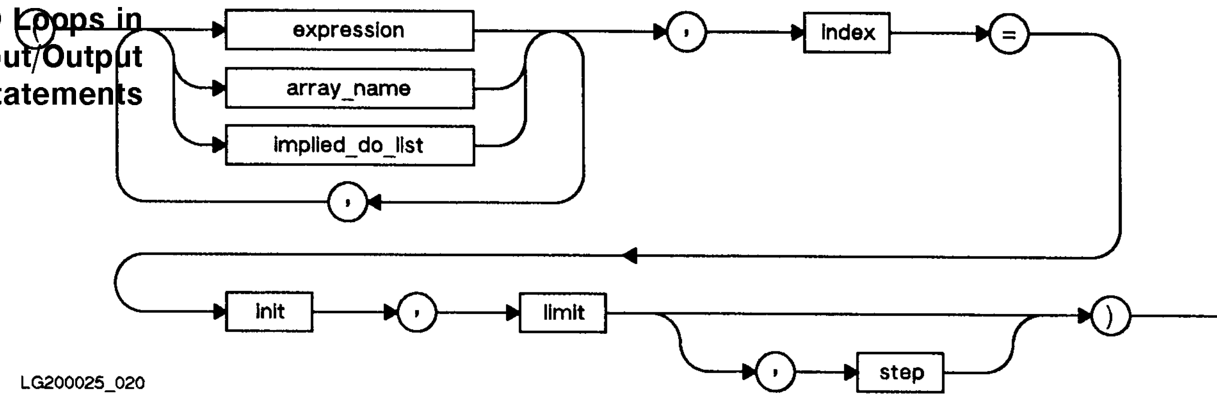
For DATA statements, only integer index variables and expressions can be used. For READ, WRITE, and PRINT statements, real index variables are also permitted.

Examples

```
DATA a, b, (vector(i), i = 1,10), k /2.5,-1.0,10*0.0,999/

DATA ((matrix(i,j), i = 0,5), j = 5,10) /36*-1/
```

The syntax of implied DO loops in DATA statements, is described in “DATA Statement (Nonexecutable)”. The syntax of implied DO loops in input/output statements is described below.

Implied DO Loops in Input/Output Statements



LG200025_020

Item	Description/Default	Restrictions
index	Loop variable that controls the number of times the preceding element list is read or written.	Must be a simple variable whose type is integer or real.
init	Expression that is the initial value for *index*.	Must be an integer or real expression.
limit	Expression that is the termination value for *index*.	Must be an integer or real expression.
step	Expression that is the increment for *index* after each execution of the DO loop. *step* can be positive or negative. Its default value is one.	Must be an integer or real expression. *step* should not equal 0. If it does, an error occurs if the RANGE directive is ON. If RANGE is OFF, an infinite loop could result.

Semantics

The implied DO loop acts like a labeled or block DO loop. The range of the implied DO loop is the list of elements to be input or output. The implied DO loop can transfer a list of simple variables, array elements, or any combination of allowable data elements. The control variable (*index*) is assigned the value of *init* at the start of the loop. Execution continues as for DO loops.

Implied DO loops can also be nested. Nested implied DO loops follow the same rules as other nested DO loops. For example, the statement:

```
WRITE (6,*) ((a(i,j), i = 1,2), j = 1,3)
```

produces the following output:

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

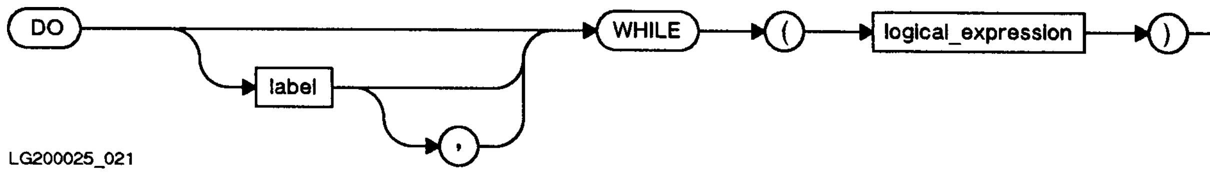
The first, or nested, DO loop is satisfied once for each execution of the outer loop. (Refer to "Nesting DO Loops" later in this section.)

Collapsed Implied DO Loop

If an implied DO loop meets certain criteria, the DO loop is collapsed and only one internal I/O call is required. The reduced number of I/O calls can yield significantly faster execution time. An implied DO loop is collapsed if the initial index value is less than or equal to the final index value and if the step value is less than or equal to one.

DO-WHILE Statement (Executable)

As a MIL-STD-1753 standard extension to the ANSI 77 standard, the DO-WHILE statement controls execution of a group of statements by causing the statements to be repeated while a logical expression is true. The DO-WHILE construct is an important element of structured programming.



LG200025_021

Each DO-WHILE loop must be terminated by a separate END DO statement, which does not require a label. Note that if the DO-WHILE statement uses the label option, the END DO statement that terminates the DO loop must have a label, and the two labels must match.

A DO-WHILE loop evaluates as follows: the logical expression is evaluated and tested at the beginning of the DO loop. If the expression evaluates to true, the group of statements between the DO-WHILE statement and the corresponding END DO statement, referred to as the range of the DO-WHILE loop, is executed and the logical expression is tested again. If the logical expression evaluates to false, the DO-WHILE loop terminates and execution continues with the statement following the END DO statement.

The rules for transfers into the range of a DO-WHILE loop are the same as for other DO loops. (Refer to "Ranges of DO Loops" later in this section.)

Examples	Notes

`DO WHILE (i .NOT. 999)`
`  READ(5,33) i`
`  :`
`END DO`
	Repeatedly reads input until entry of a terminating flag (999 in this example).

```
 index = 1
 DO WHILE (array(index) .NE. value
+       .AND. index .LE. limit)
   index = index + 1
 END DO
```

Nesting DO Loops

DO loops can contain other DO loops. This is called *nesting*. The only restriction is that each level (that is, each successive loop) must be completely contained within the preceding loop.

In a labeled DO loop, the last statement of an inner (nested) loop must either be the same as, or occur before, the last statement of the outer loop. (For programming clarity, always use a separate terminating statement for each loop.)

DO loops can be nested as long as the range of statements in any DO loop does not overlap the range of the preceding loop. Refer to the examples for an illustration of such an illegal construction.

Combinations of DO loops and IF blocks can be nested to a depth that is system dependent.

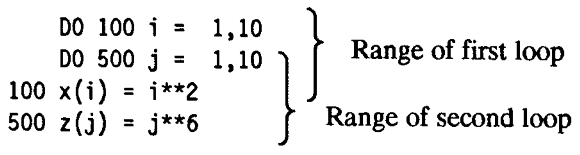
Ranges of DO Loops

The range of the DO loop is defined as the first statement following the DO statement up to and including the terminating statement defined by *label* or, in the absence of *label*, up to and including the next unmatched END DO statement.

A DO loop can be exited at any time. Normally, a DO loop is exited when the loop has been completed. Control continues at the statement following the loop's termination statement. A DO loop can be exited prematurely with, for example, a GOTO statement that transfers control out of the loop.

It is illegal to transfer control into the range of a DO loop from outside the range unless you have previously jumped out of the loop.

The following example shows an illegal construction, one in which the ranges of two loops overlap.



```
    DO 100 i =  1,10
    DO 500 j =  1,10
100 x(i) = i**2
500 z(j) = j**6
```

LG200025_098

Here is an example that shows the unexpected results possible when *index*, *init*, *limit*, and *step* are not all of the same type. If they are not, *init*, *limit*, and *step* are converted to the same type as *index*.

Example	Notes

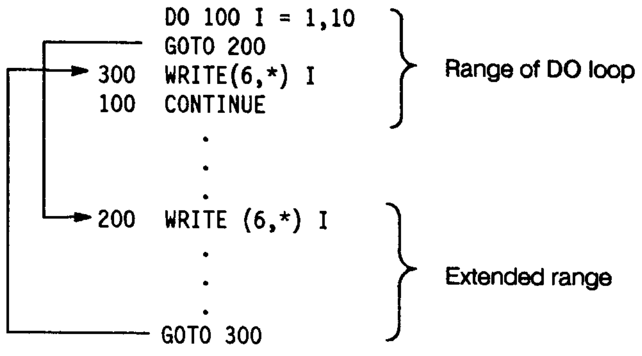
`DO i = 1,3,.1`
`   WRITE (6,*) i`
`END DO`
	The programmer intends to increment `i` by 10ths. Instead, when `.1` is converted to type integer, it becomes zero, which creates an error because *step* must not be zero.

Extended Range DO Loop

As an extension to the ANSI 77 standard, the range of a DO loop can be extended outside the loop. A control statement in the DO loop can transfer control out of the loop, and, after execution of any number of statements, control can branch back into the DO loop. The return to the DO loop must be made from a statement in the extended range of the loop.

A DO loop index cannot be modified in the loop's extended range.

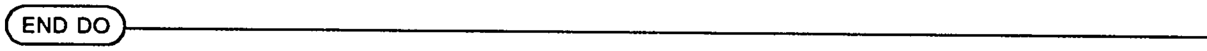
The following example illustrates the use of an extended range DO loop:



LG200025_104

END DO Statement (Executable)

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



LG200025_070

If used to terminate a labeled DO loop, the END DO statement must be labeled. See the examples under "Labeled DO Loop".

DOUBLE COMPLEX Statement (Nonexecutable)

The DOUBLE COMPLEX statement is the same as the COMPLEX*16 statement. For more information, refer to "COMPLEX Statement (Nonexecutable)".

DOUBLE PRECISION Statement (Nonexecutable)

The DOUBLE PRECISION statement is the same as the REAL*8 statement. For more information, refer to "REAL Statement (Nonexecutable)".

ELSE Statement (Executable)

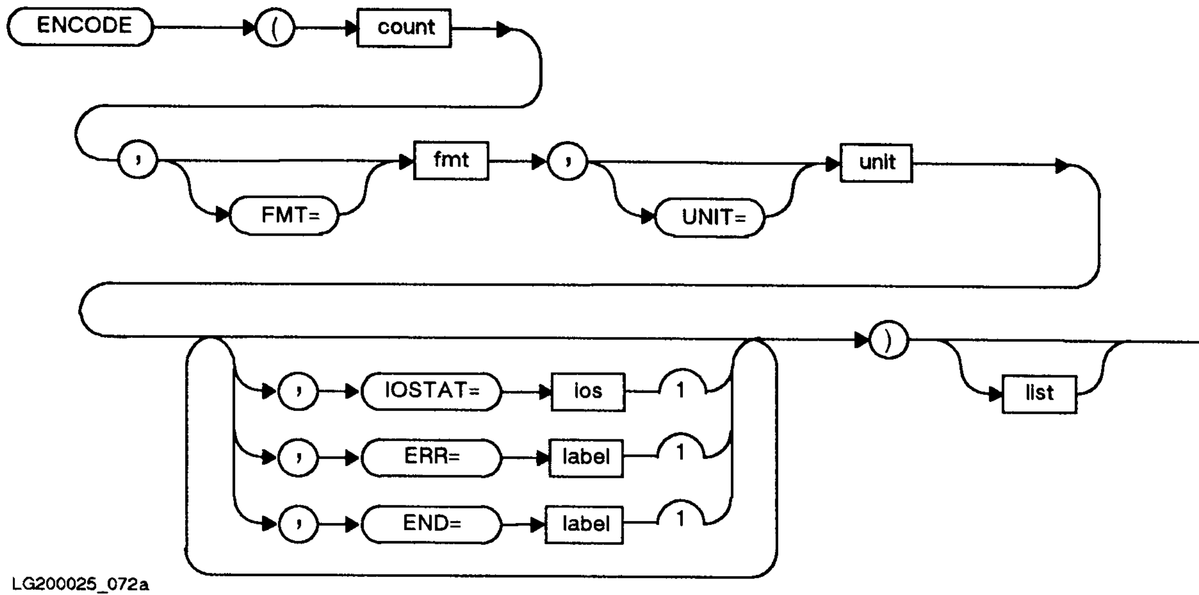
The ELSE statement is part of the block IF construct. For more information, refer to "IF Statement (Executable)".

ELSE IF Statement (Executable)

The ELSE IF statement is part of the block IF construct. For more information, refer to "IF Statement (Executable)".

ENCODE Statement (Executable)

The ENCODE statement transfers data from variables to internal storage according to a format specification.



LG200025_072a

Item	Description/Default	Restrictions
count	Integer expression that specifies the number of bytes to be translated.	Must be the first item.
fmt	Format designator.	Must be the second item; must be as specified for the PRINT statement.
unit	Internal storage designator.	Must be the third item; must be a scalar variable or array name; assumed-size and adjustable-size arrays are not permitted.
ios	Integer variable, array element, or scalar record field for error code return.	None.
label	Statement label of an executable statement; if an error occurs or an end-of-file is detected during execution of the ENCODE statement, control transfers to the statement specified by this label rather than aborting the program.	None.
list	List specifying the data to be transferred. Each item must be a variable name, an array element name, an array name, a scalar record field name, a substring, or an expression.	If *list* contains a function reference, that function must not contain any PRINT, READ, WRITE, ENCODE, or DECODE statements.

Semantics

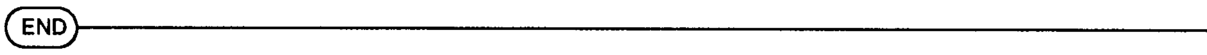
The ENCODE statement is provided for compatibility with older versions of FORTRAN. It is a nonstandard statement, and its use in new programs is strongly discouraged. If possible, use the internal file capabilities of the WRITE statement instead.

Examples	Notes

`CHARACTER*20 buf`
`ENCODE (15,'(2x,3I4,1X)', buf) 1234,45,-12`
	After this ENCODE statement, `buf` contains:
	`ΔΔ1234ΔΔ45Δ-12ΔΔΔΔΔΔ`
	where `Δ` represents a blank.

END Statement (Executable)

The END statement indicates the end of a program unit, that is, the end of a program, subroutine, function, or block data subprogram.



LG200025_022

Semantics

If an END statement is executed in a subprogram, it has the same effect as a RETURN statement. If an END statement is executed in a main program, the program terminates.

An END statement can be labeled, but it cannot be continued. It must be the last statement in a program unit.

Example	Notes

`PROGRAM xtest`
`INTEGER i`
`i = 1`
`PRINT *, i`
`END`
	The END statement terminates program `xtest`.

END DO Statement (Executable)

The END DO statement terminates the block DO and DO-WHILE statement blocks. It is described in “DO Statement (Executable)”.

END MAP Statement (Nonexecutable)

The END MAP statement terminates a MAP statement block. For more information, refer to “STRUCTURE Statement (Nonexecutable)”.

END STRUCTURE Statement (Nonexecutable)

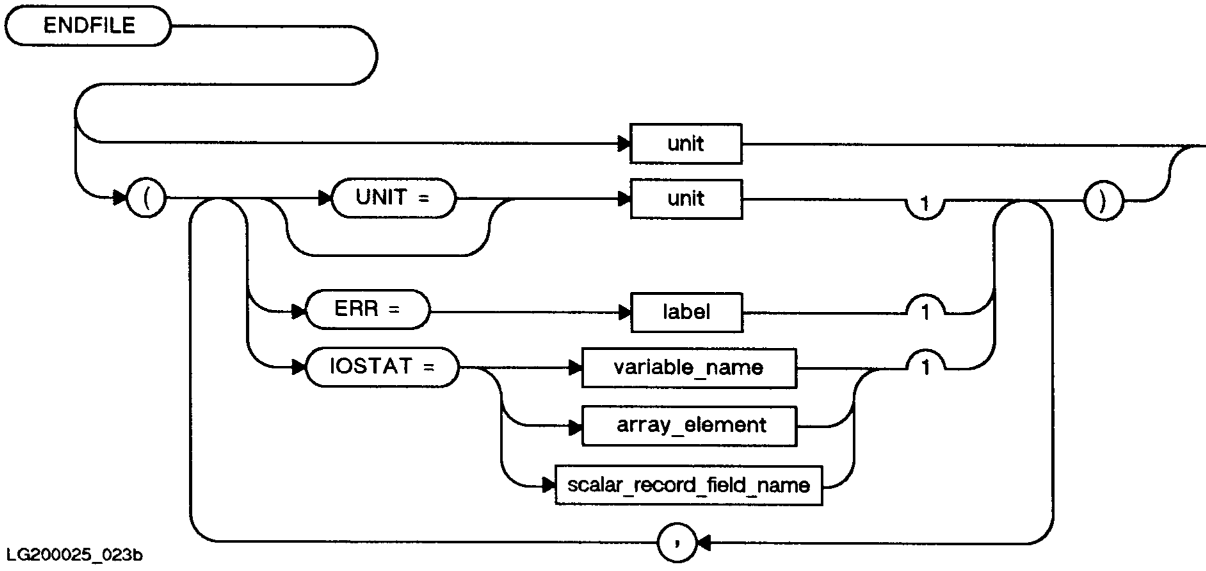
The END STRUCTURE statement terminates a STRUCTURE statement block. For more information, refer to “STRUCTURE Statement (Nonexecutable)”.

END UNION Statement (Nonexecutable)

The END UNION statement terminates a UNION statement block. For more information, refer to “STRUCTURE Statement (Nonexecutable)”.

ENDFILE Statement (Executable)

The ENDFILE statement writes an end-of-file record to the specified sequential file or device.



Item	Description/Default	Restrictions
unit	Unit number of a connected file.	None.
variable_name *array_element* *scalar_record_field_name*	Error code return.	Must be an integer data type.
label	Statement label of an executable statement.	Must be in the same program unit.

Semantics

If the `UNIT=` part of the UNIT specifier does not appear and other specifiers do, *unit* must be the first element.

If the ERR specifier is used and an error occurs during execution of the ENDFILE statement, control transfers to the specified statement rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

In a disk file, an end-of-file record can occur only as the last record. After execution of an ENDFILE statement, the file is positioned beyond the end-of-file record. Some devices (magnetic tape units, for example) can have multiple end-of-file records, with or without intervening data records.

As an extension to the ANSI 77 standard, ENDFILE operations are allowed on files open for direct access.

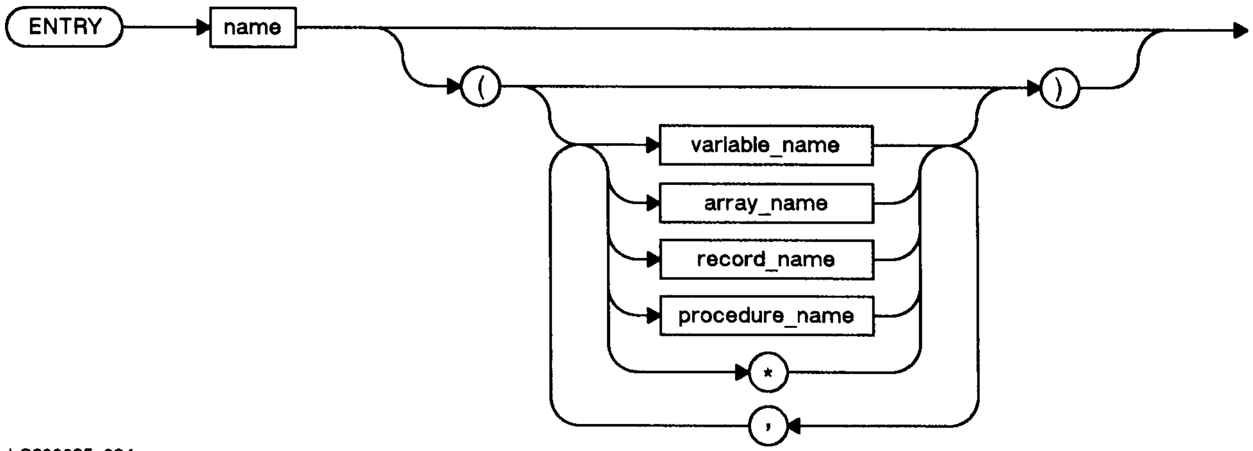
Examples	Notes
`ENDFILE 10`	An end-of-file record is written to the file connected to unit 10.
`ENDFILE (UNIT=12,IOSTAT=j,ERR=100)`	An end-of-file record is written to the file connected to unit 12. If an error occurs, control transfers to statement 100 and the error code is stored in variable `j`. If no error occurs, `j` is set to zero and control transfers to the next statement.
`ENDFILE (UNIT=12,IOSTAT=j)`	An end-of-file record is written to the file connected to unit 12. If an error occurs, the error code is stored in variable `j`. If no error occurs, `j` is set to zero. In both cases, control transfers to the next statement.
`ENDFILE (UNIT=12,ERR=100)`	An end-of-file record is written to the file connected to unit 12. If an error occurs, control transfers to statement 100. If no error occurs, control transfers to the next statement.

ENDIF Statement (Executable)

The ENDIF statement is part of the block IF construct. For more information, refer to “IF Statement (Executable)”.

ENTRY Statement (Nonexecutable)

The ENTRY statement provides an alternate name, argument list, and starting point for a function or subroutine. It can appear only in a subroutine or function subprogram, not in a main program or block data subprogram.



LG200025_024a

Item	Description/Default	Restrictions
name	Name for the alternate starting point.	None.
*	Placeholder for alternate return points.	Asterisk is permitted only in a subroutine.

Semantics

The formal arguments in an ENTRY statement can differ in order, number, type, and name from the formal arguments in the FUNCTION statement, SUBROUTINE statement, or other ENTRY statements. However, for each call to the subprogram through a given entry point, only the formal arguments of that entry point can be used.

When records are passed as arguments to entry points, all the fields in the record must agree in type, order, and dimension with the declared formal arguments.

If no formal arguments are listed after a particular ENTRY statement, no arguments are passed to the subprogram when a call to that ENTRY name is made.

The ENTRY statement name cannot appear as a variable in any statement prior to the ENTRY statement, except in a type statement within a function subprogram.

Within a subprogram, an entry name must not appear both as an entry name in an ENTRY statement and as a formal argument in a FUNCTION or SUBROUTINE statement, or another ENTRY statement. An entry name must not appear in an EXTERNAL statement.

An ENTRY statement can appear anywhere in a subprogram after the FUNCTION or SUBROUTINE statement, with the exception that the ENTRY statement must not appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the end of its DO loop.

A subprogram can have zero or more ENTRY statements. An ENTRY statement is a nonexecutable statement. If control falls into an ENTRY statement, the statement is treated as an unlabeled CONTINUE statement; that is, control moves to the next statement.

Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable, if any, whose name is also the name of the function subprogram. Therefore, any such variable that becomes defined causes all associated variables of the same type to become defined, and all those of a different type to become undefined. Such variables are not required to be of the same type unless the type is character, but the variable whose name references the function must be in a defined state when a RETURN or END statement is executed in the subprogram. An associated variable of a different type must not become defined during execution of the function reference.

The asterisks in an ENTRY statement are similar to those of the SUBROUTINE statement.

Example

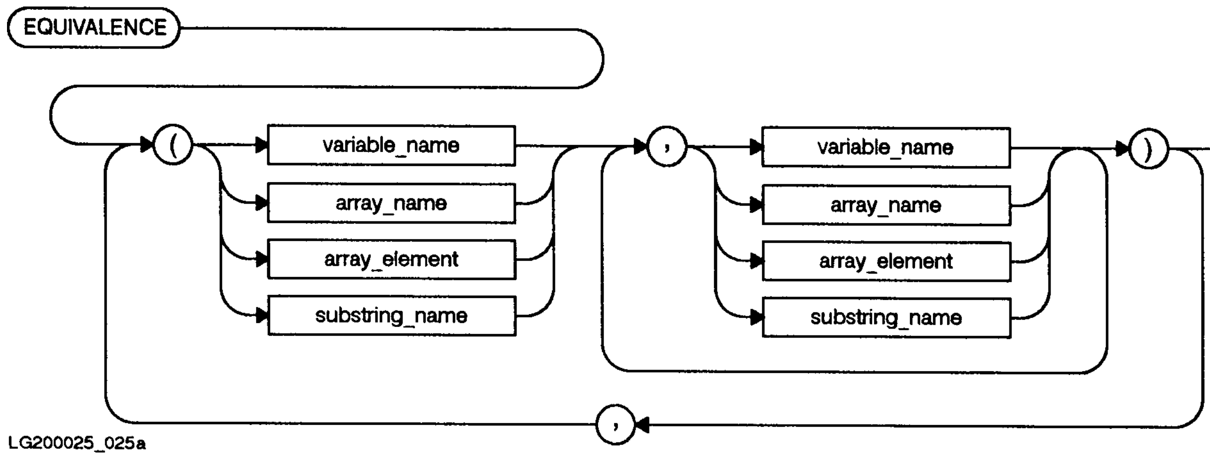
(User input is underlined.)

```
PROGRAM sum
INTEGER i,j
WRITE (6,*) 'Enter two numbers: '
READ (5,*) i,n
IF (i .EQ. 0) THEN
    CALL sum1(j)
ELSE IF (j .EQ. 0) THEN
    CALL sum1(i)
ELSE
    CALL sum2(i,j)
ENDIF
END

SUBROUTINE sum2(i,j)
WRITE (6,*) 'Neither number equals 0.'
i = i + j
ENTRY sum1(i)
WRITE (6,*) 'The sum of the numbers is', i, '.'
RETURN
END
```

```
Enter two numbers: 9 0
The sum of the numbers is 9.

Enter two numbers: 1 2
Neither number equals 0.
The sum of the numbers is 3.
```

EQUIVALENCE Statement (Nonexecutable)

The EQUIVALENCE statement associates variables so that they share the same storage space.



Semantics

Function names, formal arguments, dynamic arrays, and record names must not appear in an EQUIVALENCE statement. Each array or substring subscript must be an integer constant expression.

The EQUIVALENCE statement conserves storage. For example, arrays that are manipulated at different times in the same program can share the same storage space through the EQUIVALENCE statement. Thus, the same storage space is used for each array.

Equivalenced data items can be of different types. As an extension to the ANSI 77 standard, character and noncharacter data items can share the same storage space through the EQUIVALENCE statement.

The EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If an array and a simple variable share the same storage space through the EQUIVALENCE statement, the array does not have the characteristics of a simple variable and the simple variable does not have the characteristics of an array. They only share the same storage space. Care should be taken when data types of different sizes share the same storage space, because the EQUIVALENCE statement specifies that each data item in a list has the same first storage unit. For example, if an INTEGER*4 variable and a REAL*8 variable share the same storage space, the integer value occupies the same space as the leftmost word of the two-word real value.

Equivalence of Character Variables

As an extension to the ANSI 77 standard, character and noncharacter data items can share the same storage space.

Example	Notes
`EQUIVALENCE (a,b),(c(2),d,e)`	The variables `a` and `b` share the same storage space; `c(2)`, `d`, and `e` share the same storage space.

Multi-Dimensioned Equivalence

As an extension to the ANSI 77 standard, it is possible to indicate the element of an array with two or more dimensions by specifying its position as if it were a single dimension array.

Example

```
BYTE message (4,10)
INTEGER name (10)
EQUIVALENCE (name, message(1))
```

Memory Storage Locations for Message and Name

Array MESSAGE	Storage Space Byte Number	Array NAME

MESSAGE(1,1)
MESSAGE(2,1)
MESSAGE(3,1)
MESSAGE(4,1)
	1 through 4	NAME(1)

MESSAGE(1,2)
MESSAGE(2,2)
MESSAGE(3,2)
MESSAGE(4,2)
	5 through 8	NAME(2)

MESSAGE(1,3)
MESSAGE(2,3)
MESSAGE(3,3)
MESSAGE(4,3)
	9 through 12	NAME(3)

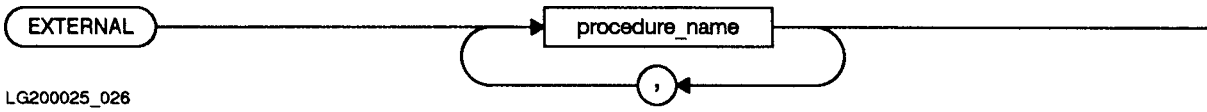
MESSAGE(1,4)
MESSAGE(2,4)
MESSAGE(3,4)
MESSAGE(4,4)
	13 through 16	NAME(4)

⋮	⋮	⋮

MESSAGE(1,10)
MESSAGE(2,10)
MESSAGE(3,10)
MESSAGE(4,10)
	37 through 40	NAME(10)

EXTERNAL Statement (Nonexecutable)

The EXTERNAL statement identifies a name as representing a subprogram name and permits the name to be used as an actual argument in subprogram calls.



LG200025_026

Item	Description/Default	Restrictions
procedure_name	Name of a subprogram.	Each name can appear once only in a given EXTERNAL statement, and in at most one EXTERNAL statement in a given program unit.

The EXTERNAL statement provides a means of using the names of subroutine subprograms and function subprograms as actual arguments. The EXTERNAL statement is necessary to inform the compiler that these names are subprograms or function names, not variable names. Whenever a subprogram name is passed as an actual argument, it must be placed in an EXTERNAL statement in the calling program. If an intrinsic function name appears in an EXTERNAL statement, the compiler assumes that a user subprogram by that name exists; the intrinsic function is not available to that program unit. A name cannot appear in both an EXTERNAL and INTRINSIC statement.

Examples	Notes

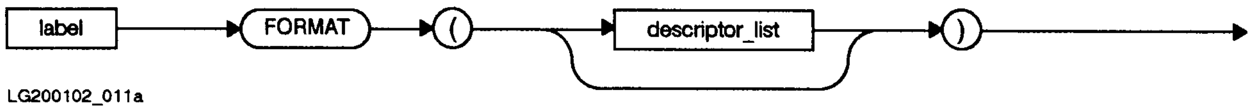
`PROGRAM my_sin`
`EXTERNAL sin`
`REAL sin, x, y`
`READ(5,*) y`
`x = sin(y)`
`WRITE(6,*) x`
`END`
	This call is to the *user-written* function named `sin`, not to the *intrinsic* function SIN. A statement function name must not appear in an EXTERNAL statement.

`EXTERNAL b1`
`CALL sub(a,b1,c)`
`⋮`
`END`
	The EXTERNAL statement declares `b1` to be a subprogram name. The call to `sub` passes the values of `a` and `c`, and passes the name of the subprogram (`b1`).

`SUBROUTINE sub(x,y,z)`
`z = y(z)`
`RETURN`
`END`
	The reference to `y` causes `b1` to be called.

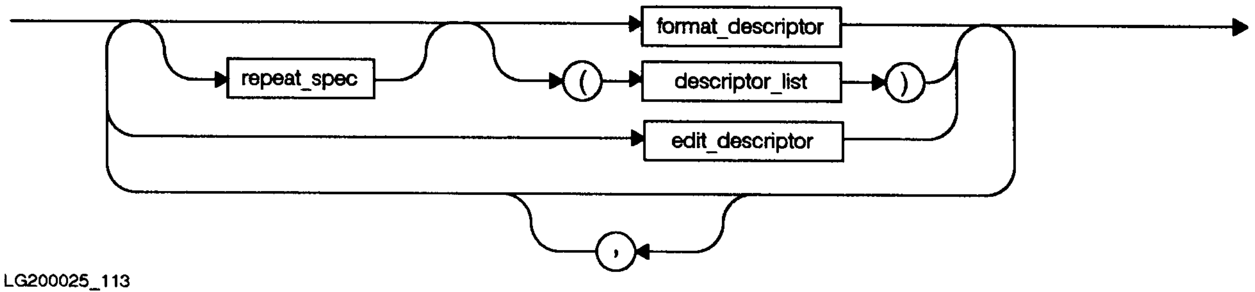
FORMAT Statement (Nonexecutable)

The FORMAT statement describes how input and output information is to be arranged.



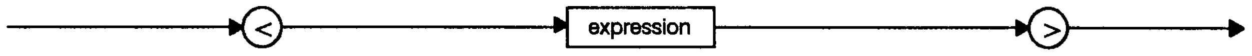
LG200102_011a

Descriptor List



LG200025_113

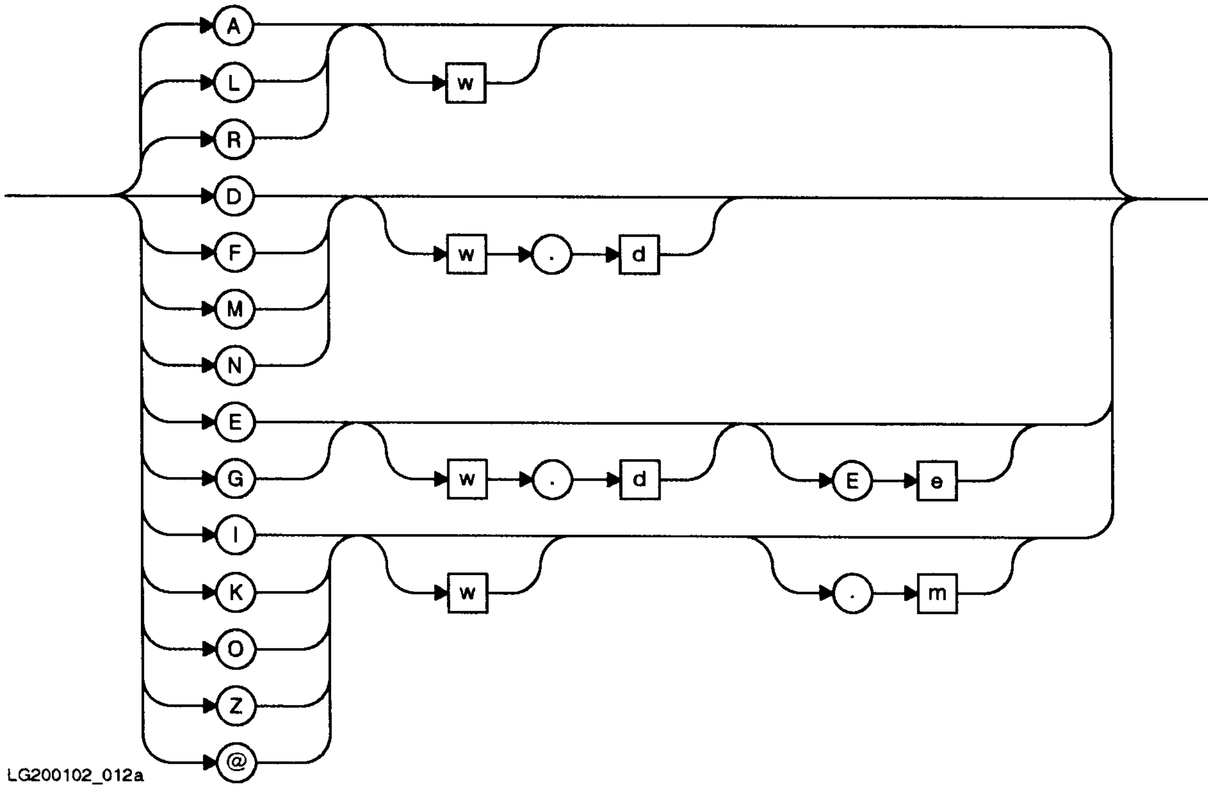
Variable Format Descriptor



LG200025_114

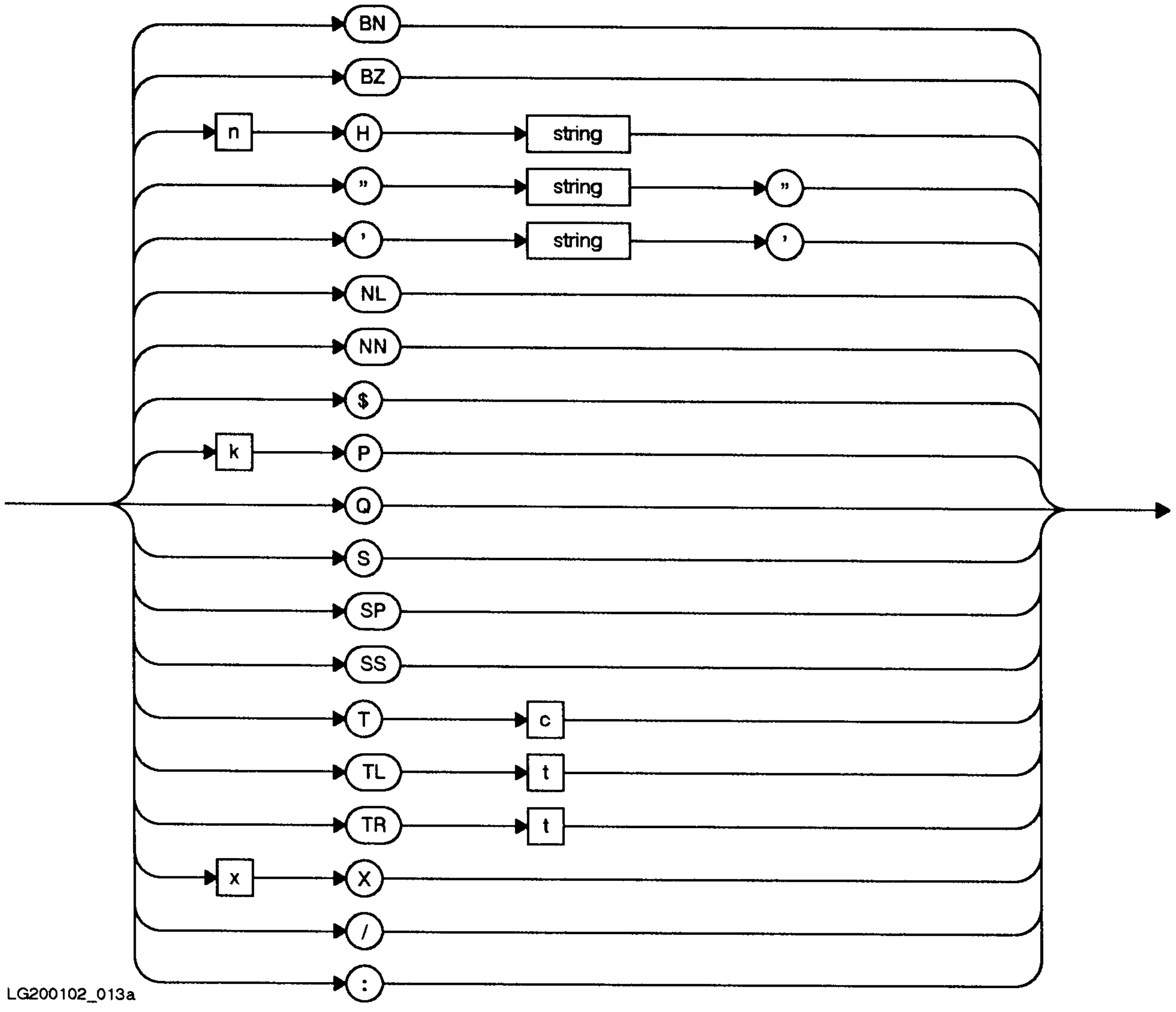
Item	Description/Default	Restrictions
label	Statement label.	Required.
repeat_spec	Repeat specification.	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
format_descriptor	Format descriptor. See the following Format Descriptor syntax diagram and table for details.	None.
edit_descriptor	Edit descriptor. See the following Edit Descriptor syntax diagram and table for details.	None.
descriptor_list	A list of format and edit descriptors.	None.
expression	A positive integer expression.	None.

Format Descriptor



Item	Description/Default	Restrictions
d	Number of digits in fractional part. If omitted, the default value is based on the data type of the I/O list element. (Refer to Table 3-5 for default values.)	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
e	Number of digits in exponent part. If omitted, the default value is based on the data type of the I/O list element. (Refer to Table 3-5 for default values.)	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
m	Minimum number of digits to be output. The default is one.	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
w	Field width. If omitted, the default value is based on the data type of the I/O list element. (Refer to Table 3-5 for default values.)	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.

Edit Descriptor



Item	Description/Default	Restrictions
c	Column position.	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
k	Scale value.	Must be an integer constant or a variable format descriptor.
n	Number of characters.	Must be an unsigned positive integer constant.
string	A series of one or more ASCII characters.	The `H` format has the form of a Hollerith constant. The `"` and `'` formats have the form of character constants.
t	Number of columns to skip.	Must be an unsigned positive integer constant or a variable format descriptor whose value is positive.
x	Number of columns to skip.	Must be an unsigned positive integer constant.

Table 3-5. Default Format Descriptor Field Values

Format Descriptor	List Element Type	*w*	*d*	*e*
`@`, `I`, `K`, `O`, `Z`	INTEGER*2, LOGICAL*1, LOGICAL*2	7		
`@`, `I`, `K`, `O`, `Z`	INTEGER*4, LOGICAL*4	12		
`L`	LOGICAL*1, LOGICAL*2, LOGICAL*4	2		
`I`	REAL*4, COMPLEX*8	12		
`I`	REAL*8, COMPLEX*16	23		
`I`	REAL*16	44		
`D`, `E`, `F`, `G`, `M`, `N`	INTEGER*2, LOGICAL*1, LOGICAL*2	15	7	2
`D`, `E`, `F`, `G`, `M`, `N`	INTEGER*4, LOGICAL*4	25	16	2
`D`, `E`, `F`, `G`, `M`, `N`	REAL*4, COMPLEX*8	15	7	2
`D`, `E`, `F`, `G`, `M`, `N`	REAL*8, COMPLEX*16	25	16	2
`D`, `E`, `F`, `G`, `M`, `N`	REAL*16	42	33	3

Note

When using field descriptors without a field width value, the descriptors must be separated by commas. For example, `(I4 I4)` is allowed because the first field width value is present. However, `(I I4)` is invalid. The descriptors must be separated with a comma, as in `(I,I4)`, because the first field width value is not present.

The format descriptors are summarized in Table 3-6.

Table 3-6. Format Descriptors

Descriptor	List Element Data Type
`A`[*w*]	Any character or Hollerith
`R`[*w*]	Any character or Hollerith
`D`[*w* . *d*]	Any real or complex
`E`[*w* . *d*[`E`*e*]]	Any real or complex
`F`[*w* . *d*]	Any real or complex
`G`[*w* . *d*[`E`*e*]]	Any real or complex
`M`[*w* . *d*]	Any real
`N`[*w* . *d*]	Any real
`I`[*w*[. *m*]]	Any integer; decimal format
`@`[*w*[. *m*]]	Any integer; octal format
`K`[*w*[. *m*]]	Any integer; octal format
`O`[*w*[. *m*]]	Any integer; octal format
`Z`[*w*[. *m*]]	Any integer; hexadecimal digits
`L`[*w*]	Any logical
d, *e*, *m*, and *w* are described above in the item list for the Descriptor List syntax.	

Each of the format descriptors can be preceded by a repeat specification (the `4` in `4I7`, for example). See *repeat_spec* in the item list for the Format Descriptor syntax.

Example	Notes
`10 FORMAT(I3,5F12.3)`	The specification is for an integer number with a field width of 3, and five real numbers with a field width of 12 and three significant digits to the right of the decimal point.

The edit descriptors are summarized in Table 3-7.

Table 3-7. Edit Descriptors

Descriptor	Function
`BN`	Ignore blanks.
`BZ`	Treat blanks as zeros.
n`H`*string*	Hollerith literal.
`"`*string*`"`	Literal editing.
`'`*string*`'`	Literal editing.
`NL`	Restore newline.
`NN`	No newline.
`$`	No newline; same as `NN`.
k`P`	Scale factor.
`Q`	Number of characters remaining in current input record.
`S`	Processor determines sign output; same as `SS`.
`SP`	Output optional plus signs.
`SS`	Inhibit optional plus sign output.
`T`*c*	Skip to column *c*.
`TL`*t*	Skip *t* positions to the left.
`TR`*t*	Skip *t* positions to the right.
x`X`	Skip *x* positions to the right
`/`	Begin new record.
`:`	Terminate format if I/O list empty.
c, *k*, *n*, *string*, *t*, and *x*, are described above in the item list for the Edit Descriptor syntax.	

Both apostrophes and quotation marks can be used in input/output statements and in FORMAT statements. For example:

```
   WRITE (6,'("Average is ",I5)') iaverage
```

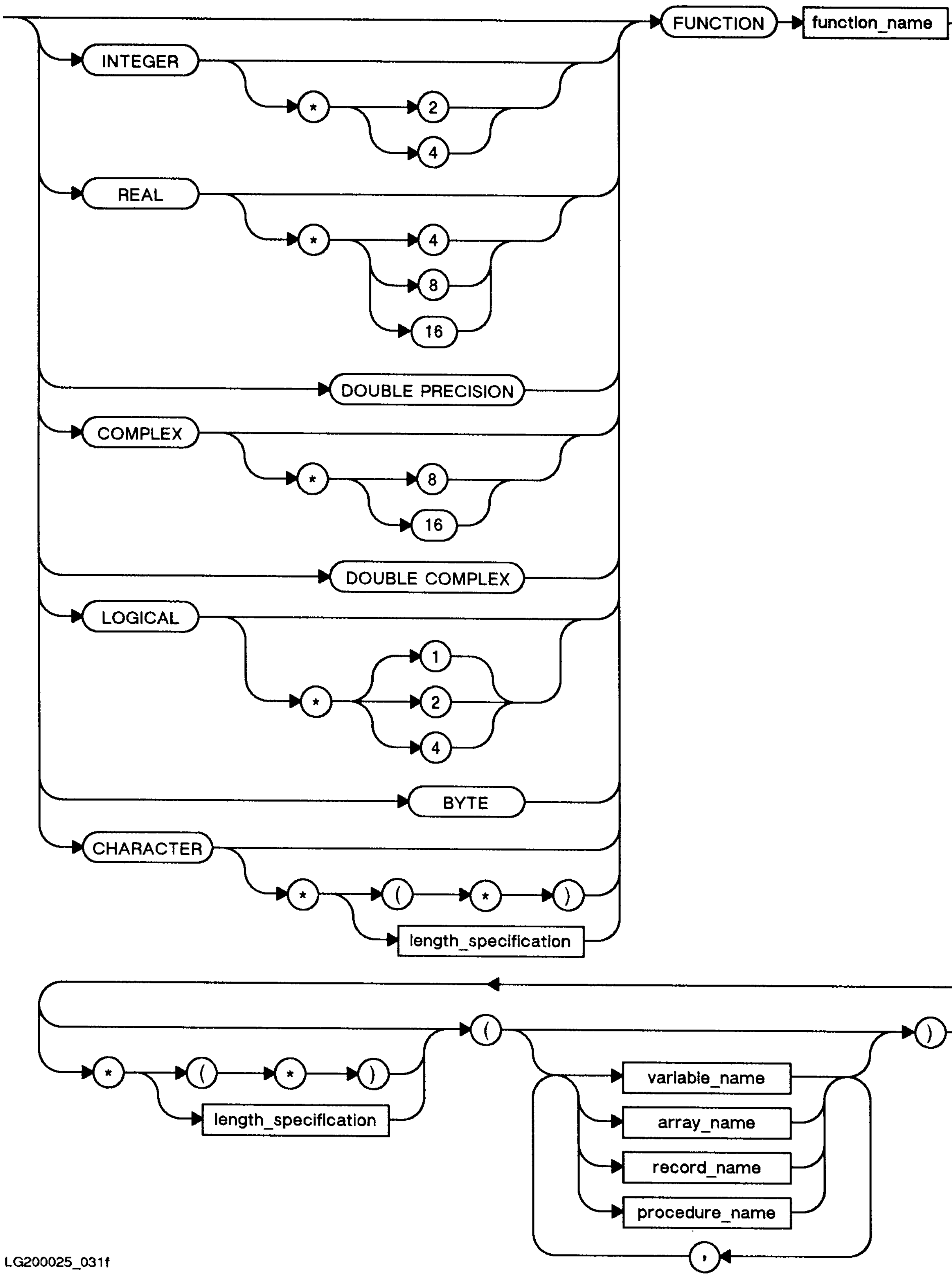
and:

```
   WRITE (6,'(''Average is '',I5)') iaverage
```

are equivalent.

FUNCTION Statement (Nonexecutable)

The FUNCTION statement identifies a program unit as a function subprogram.



LG200025_031f

Semantics

For more information on types, refer to the discussions of each type elsewhere in this chapter. If the type is not specified, the name is typed the same way as the variables.

The formal arguments in a FUNCTION statement can be used as:

- Variables.
- Array names.
- Subprogram names.
- Record names.

The formal arguments should be of the same type as the actual arguments that are passed to the function from the calling program unit. When passing records as arguments to a function, all fields must be the same type, order, and dimension as the declared formal arguments.

If a formal argument of type character has a length of `(*)` declared, the formal argument assumes the length of the associated actual argument for each reference of the function. If the function is of type `CHARACTER*(*)`, it assumes the length declared for it by the calling program. (The length may be subsequently redefined within the calling program.)

Examples	**Notes**
`FUNCTION comp()`	Defines a function `comp` with no arguments.
`INTEGER FUNCTION timex(a,b,k)`	Defines an integer function `timex` with three arguments.
`CHARACTER*6 FUNCTION namex(q)`	Defines a character function `namex` six characters long, with one argument.

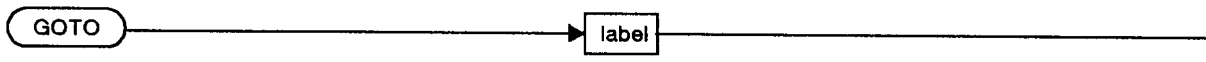
GOTO Statement (Executable)

The GOTO statement transfers control to a labeled statement in the same program unit. It has three forms, which are described separately below:

- Unconditional GOTO
- Computed GOTO
- Assigned GOTO

Unconditional GOTO Statement (Executable)

The unconditional GOTO statement transfers control to the specified statement.

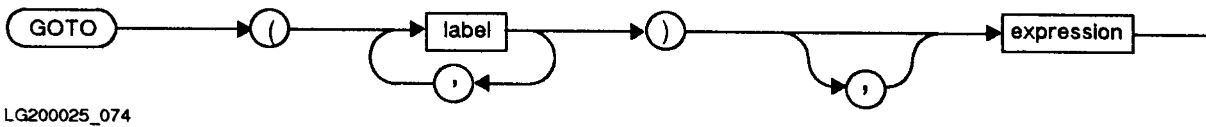


LG200025_073

Item	Description/Default	Restrictions
label	Label of an executable statement.	None.

Computed GOTO Statement (Executable)

The computed GOTO statement transfers control to one of several statements, depending on the value of an expression.



LG200025_074

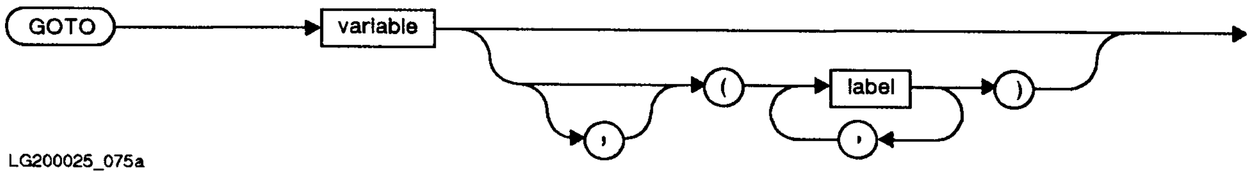
Item	Description/Default	Restrictions
label	Label of an executable statement.	None.
expression	Arithmetic expression.	Any integer or real type.

The use of noninteger expressions is an extension to the ANSI 77 standard. The computed GOTO statement passes control to one of several labeled statements depending on the result of an evaluation. The expression is evaluated and truncated to an integer value (the index). The index selects the statement label in the label list.

For example, if the index is 1, control passes to the statement whose label appears in the first position in the list of labels. If the index value is 2, the second label in the list is used, and so on. If the value of the expression is less than 1 or greater than the number of labels in the label list, control passes to the statement following the computed GOTO.

Assigned GOTO Statement (Executable)

The assigned GOTO statement transfers control to the statement whose label is stored in the variable by an ASSIGN statement.



LG200025_075a

Item	Description/Default	Restrictions
label	Label of an executable statement.	None.
variable	Integer simple variable.	Must be INTEGER*4.

The *variable* must be given a label value of an executable statement through an ASSIGN statement prior to execution of the GOTO statement. When the assigned GOTO statement is executed, control transfers to the statement whose label matches the label value of *variable*.

The optional labels following *variable* form a list of label values that *variable* might assume. If you use the list of labels, the compiler can sometimes produce more efficient object code.

However, if the label list is specified and the value in *variable* is not in the list, the results are not defined. A run-time range error can result if the RANGE compiler directive is ON.

Examples	Notes
`GOTO 20`	In this unconditional GOTO statement, control passes to the statement labeled 20 when the GOTO statement is executed. Statement 20 can be before or after the GOTO statement, but must be present in the same program unit.

`a = 3`
`GOTO (30,60,50,100) a`
	In this computed GOTO statement, because `a` has a value of 3, control passes to statement 50.

`b = 1.5`
`z = 1`
`GOTO (10,20,40,40) b + z`
	In this computed GOTO statement, because `INT(b + z) = 2`, control passes to statement 20.

`ASSIGN 10 TO age`
`GOTO age`
	In this assigned GOTO statement, control transfers to statement 10 when the GOTO statement is executed.

`ASSIGN 100 TO time`
`GOTO time (90,100,150)`
	In this assigned GOTO statement, control transfers to statement 100 when the GOTO statement is executed.

IF Statement (Executable)

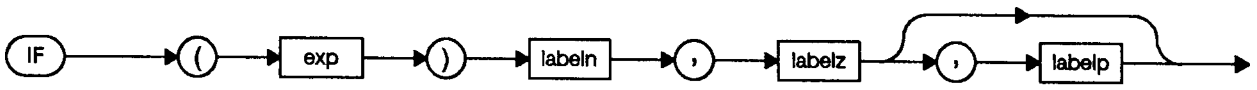
The IF statement provides a means for decision making. There are three types of IF statements:

- Arithmetic IF
- Logical IF
- Block IF

The maximum level to which a mixture of IF and DO statements, can be nested is 20. Exceeding this number makes programs unnecessarily complicated and can cause internal compiler errors stating that the statement is too complicated.

Arithmetic IF Statement (Executable)

An arithmetic IF statement transfers control to one of two or three labeled statements, depending on whether an expression evaluates to a negative, zero, or positive value.



LG200025_033a

Item	Description/Default	Restrictions
exp	Arithmetic expression.	Any type except complex, logical, or character.
labeln *labelz* *labelp*	Label of an executable statement.	None.

When an arithmetic statement is executed, *exp* is evaluated. If the value is negative, control passes to the statement whose label is *labeln*. If the value is zero, control passes to the statement whose label is *labelz*. If the value is positive, control passes to the statement whose label is *labelp*.

As an extension to the ANSI 77 standard, *labelp* is optional. If *labelp* is omitted, control transfers to *labeln* when the value of *exp* is negative or to *labelz* when the value of *exp* is zero or positive.

If the value of the expression exceeds the range of the expression, an overflow condition occurs. Overflow conditions are not detected by the compiler. A hardware overflow during evaluation of any expression causes a run-time error, which halts the program. Overflow conditions can often be avoided by using logical instead of arithmetic IF statements.

Two of the labels in the label list can be the same; control branches to one of two possible statements rather than three. In fact, all of the labels in the list can be the same, in which case control branches to the statement bearing the label, regardless of the results of the evaluation.

If two of three labels are the same, and one of them indicates the next statement, the statement should be rewritten as a logical IF for improved readability.

For example, these arithmetic IF statements:

```
   IF (exp) 10,10,20
10   ...
   IF (exp) 10,20,10
20   ...
```

are the same as these logical IF statements:

```
   IF (exp .GT. 0) GOTO 20
10   ...
   IF (exp .NE. 0) GOTO 10
20   ...
```

Examples	Notes

`testa = 0.`
`IF (testa) 50,100,50`
	Because `testa` equals zero, control passes to statement 100.

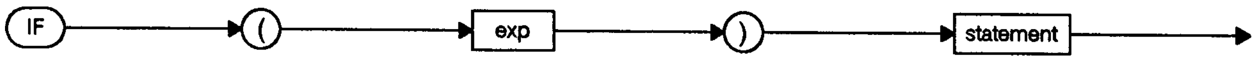
`i = 10`
`j = -(15)`
`IF (i + j) 10,20,30`
	Because `i + j` is negative, control passes to statement 10.

`z = 10.`
`a = 60.`
`IF (a + z) 100,100,60`
	Because `a + z` is positive, control passes to statement 60.

`IF (a + b) 10,20,30`	Control passes to statement 10, 20, or 30 depending on the value of `a + b`.

Logical IF Statement (Executable)

The logical IF statement evaluates a logical expression and executes one statement if the expression is true.



LG200025_034

Item	Description/Default	Restrictions
exp	Logical expression.	None.
statement	Executable statement.	Cannot be a DO, END, block IF, or logical IF statement.

The logical IF statement is a two-way decision maker. If the logical expression contained in the IF statement is true, the statement contained in the IF statement is executed and control passes to the next statement. If the logical expression is false, the statement contained in the IF statement is not executed and control passes to the next statement in the program.

Examples	Notes

`a = b`
`IF (a .EQ. b) GOTO 100`
	Because the expression `a .EQ. b` is true, control passes to statement 100.

`IF (p .AND. q) res=10.5`	If `p` and `q` are both true, the value of `res` is replaced by 10.5; otherwise the value of `res` is unchanged.

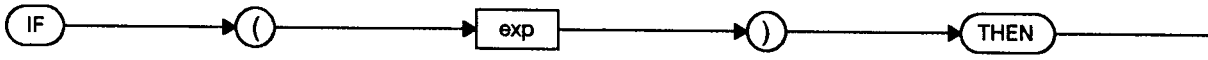
Block IF Statement (Executable)

The block IF statement block is an extension of the logical IF statement, allowing one of a set of blocks of statements to be executed, depending on the value of one or more logical expressions. The blocks within the block IF statement block are managed by four statements:

- IF-THEN statement
- ELSE statement
- ELSE IF statement
- ENDIF statement

IF-THEN Statement (Executable)

The IF-THEN statement evaluates a logical expression and permits the execution of two specified blocks of statements, depending on the results.



LG200025_035

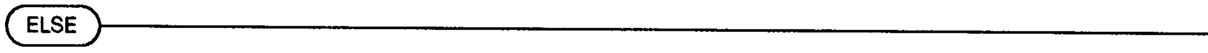
Item	Description/Default	Restrictions
exp	Logical expression.	None.

Semantics

The IF-THEN statement, like the logical IF statement, is a two-way decision maker. If the logical expression in the IF-THEN statement evaluates to true, the block of statements between the IF-THEN statement and the next following ELSE, ELSE IF, or ENDIF statement is executed. If the logical expression in the IF-THEN statement evaluates to false, the block of statements between the corresponding ELSE or ELSE IF and ENDIF statements is executed. If no ELSE or ELSE IF block is present, control passes to the statement following the ENDIF statement.

ELSE Statement (Executable)

The ELSE statement terminates the block of the corresponding IF-THEN statement and marks the beginning of the ELSE block.

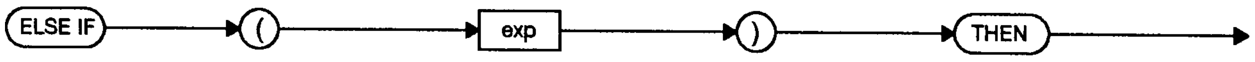


LG200025_036

The ELSE statement serves as a beginning marker for the block of statements to be executed if the logical expression in its corresponding IF-THEN or ELSE IF statement evaluates to false. If so, the block of statements between the ELSE and its corresponding ENDIF statement is executed. If the logical statement evaluates to true, the statements in the IF-THEN block are executed until an ELSE statement is encountered. Control then transfers to the statement following the ENDIF statement.

ELSE IF Statement (Executable)

The ELSE IF statement is a special case of an ELSE statement. It functions the same as an ELSE statement that has an IF-THEN statement as the first statement of its ELSE block.



LG200025_037

Item	Description/Default	Restrictions
exp	Logical expression.	None.

ENDIF Statement (Executable)

The ENDIF statement terminates an IF-THEN or ELSE statement block.

LG200025_038

When an IF-THEN block is terminated by an ELSE or ENDIF statement, or an ELSE block is terminated by its associated ENDIF statement, control transfers to the statement following the ENDIF statement.

Nesting IF Statements

One block IF statement can contain any number of ELSE IF sub-blocks but only one ELSE sub-block. The depth to which combinations of DO loops and IF blocks can be nested is system dependent.

Using ELSE IF does not change the nesting level of the IF block. (The term *nesting level* refers to the number of preceding IF-THEN statements minus the number of preceding ENDIF statements.) The nesting level must be equal to zero at the end of each program unit. An IF-THEN statement increases the nesting level by one, while the ENDIF statement decreases the nesting level by one.

Example	Nesting Level
`IF (`*exp1*`) THEN`	0
`   :`	1
`   IF (`*exp2*`) THEN`	:
`   :`	2
`   ELSE IF (`*exp3*`) THEN`	:
`   :`	2
`   ELSE`	:
`   :`	2
`   ENDIF`	:
`   :`	1
`ELSE IF (`*exp4*`) THEN`	:
`   :`	1
`ENDIF`	:
	0

Examples	Notes

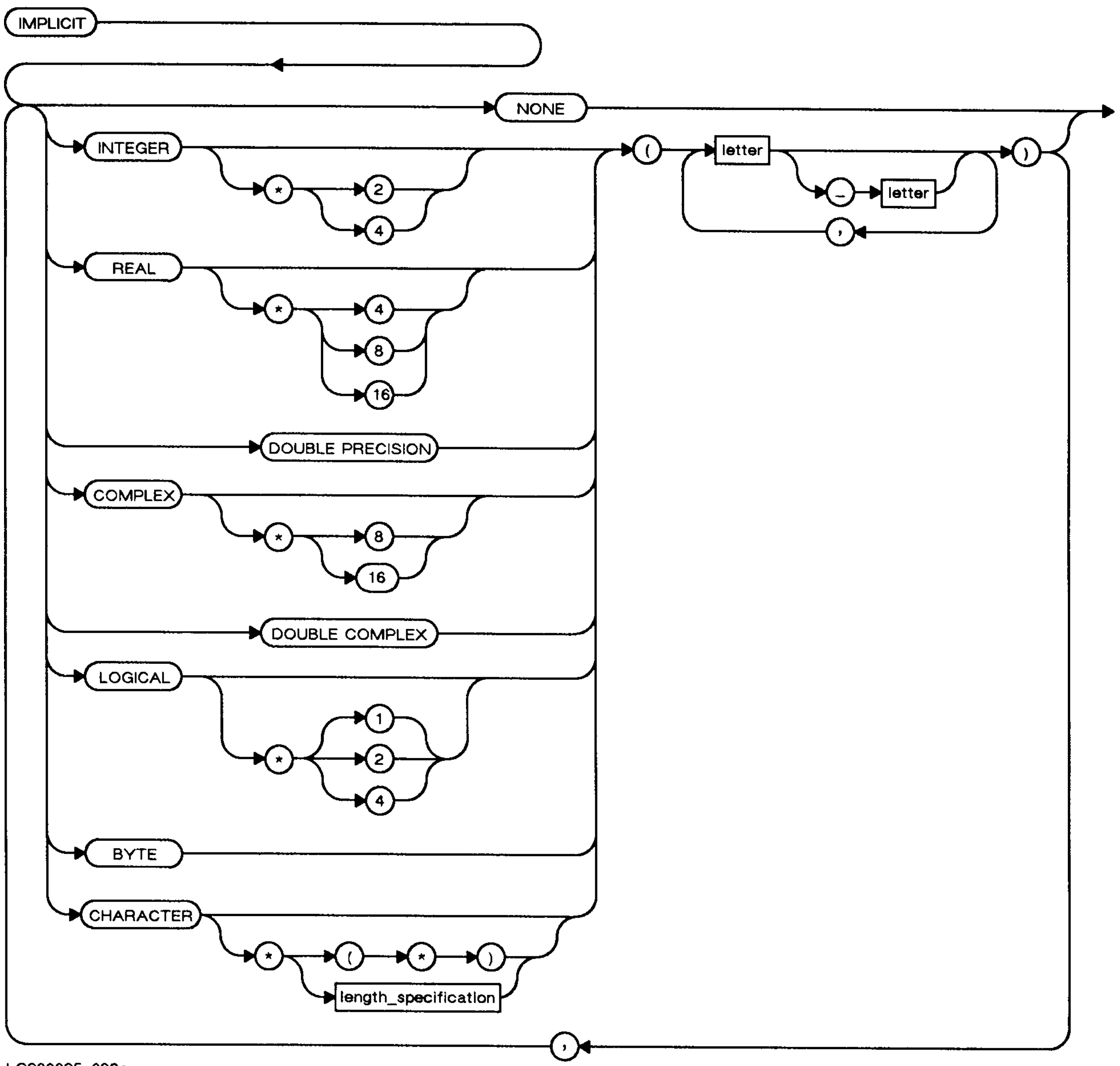
`x = y`
`IF (x.EQ.y) THEN`
`x=x+1`
`ENDIF`
	Because `x = y`, the value of `x` is replaced by the value of `x+1`. Note that this is equivalent to the following logical IF statement:
	`IF (x.EQ.y) x=x+1`

`IF (x.LT.0) THEN`
`   y = SQRT (ABS(x))`
`   z = x+1-y`
`ELSE`
`   y = SQRT (x)`
`   z = x-1`
`ENDIF`
	If `x < 0`, one block of code is executed; if `x` $\geq$ `0`, a different block of code is executed.

`IF (n(i).EQ.0) THEN`
`   n (i) = n (j)`
`   j = j+1`
`   IF (j.LT.k) THEN`
`      k = k-1`
`   ELSE IF (j.EQ.k) THEN`
`      k = k+1`
`   ENDIF`
`ELSE`
`   m = i`
`   k = n(i)`
`ENDIF`
	This example demonstrates nesting of IF blocks using the construct:
	`IF (`*exp1*`) THEN`
	`:`
	`IF (`*exp2*`) THEN`
	`:`
	`ELSE IF (`*exp3*`) THEN`
	`:`
	`ENDIF`
	`ELSE`
	`:`
	`ENDIF`

IMPLICIT Statement (Nonexecutable)

The IMPLICIT statement overrides or confirms the default type associated with the first letter of a variable name.



LG200025_039c

Semantics

For more information on types, refer to the discussions of each type, elsewhere in this chapter.

An IMPLICIT statement specifies a default type for all variables, arrays, named constants, function subprograms, ENTRY names in function subprograms, and statement functions that begin with any letter that appears in an IMPLICIT statement and are not explicitly given a type. It does not change the type of any intrinsic functions.

The IMPLICIT statement itself can be overridden for specific names when these names appear in a type statement. For example,

```
   IMPLICIT INTEGER (A)
```

specifies that items whose names start with the letter `A` default to type INTEGER. However, a subsequent type statement, such as

```
   REAL ABLE
```

overrides the IMPLICIT defaults and sets the variable `ABLE` to type REAL.

Uppercase and lowercase letters are equivalent in arguments to the IMPLICIT statement. Therefore, both the following

```
   IMPLICIT INTEGER (Q)
   IMPLICIT INTEGER (q)
```

are the same.

An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the function name. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

Note

A variable in parentheses cannot be used as a length specifier for a character data type in an IMPLICIT statement.

The IMPLICIT NONE form is a MIL-STD-1753 extension to the ANSI 77 standard. If IMPLICIT NONE is specified, implicit data typing is disabled and all variables, arrays, named constants, function subprograms, entry names in function subprograms, and statement functions (but not intrinsic functions) must be explicitly typed. If IMPLICIT NONE is specified, it must be the only IMPLICIT statement in the program unit. The types of intrinsic functions are not affected. The IMPLICIT NONE statement is recommended for general use as an excellent structured programming construct, because it forces the declaration of all user-defined names.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements, except PARAMETER statements. A letter must not be specified more than once, whether singly or in a range of letters, in all the IMPLICIT statements in a program unit.

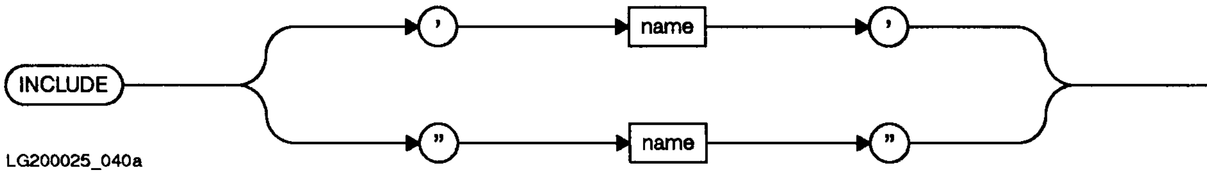
Specifying a range of letters (for example, `A-E`) has the same effect as writing a list of single letters (for example, `A,B,C,D,E`).

Examples	Notes
`IMPLICIT COMPLEX*16(i,j,k),INTEGER*2(a-c)`	All undefined symbolic names beginning with `i`, `j`, or `k` default to type COMPLEX*16. Those beginning with `a`, `b`, or `c` default to type INTEGER*2.

`IMPLICIT NONE`
`INTEGER i,j,k`
`REAL x,y,z`
`a = x+y`
`STOP`
`END`
	All names must be declared. An error occurs in the fourth line because `a` was not declared.

INCLUDE Statement (Nonexecutable)

The INCLUDE statement is a MIL-STD-1753 extension to the ANSI 77 standard. It causes the compiler to include and process subsequent source statements from a specified file or device. When end-of-file is read from this file or device, the compiler continues processing at the line following the INCLUDE statement.



Item	Description/Default	Restrictions
name	File name.	None.

INCLUDE statements cannot be continued. Include files can be nested nonrecursively; that is, an INCLUDE statement cannot mention an active include file. The maximum number of include files that can be open at one time is eight.

Line numbering within the listing of an included file begins with one. A plus sign is displayed just to the left of the line number in the include file. When the included file listing ends, the include level decreases appropriately and the previous line numbering resumes.

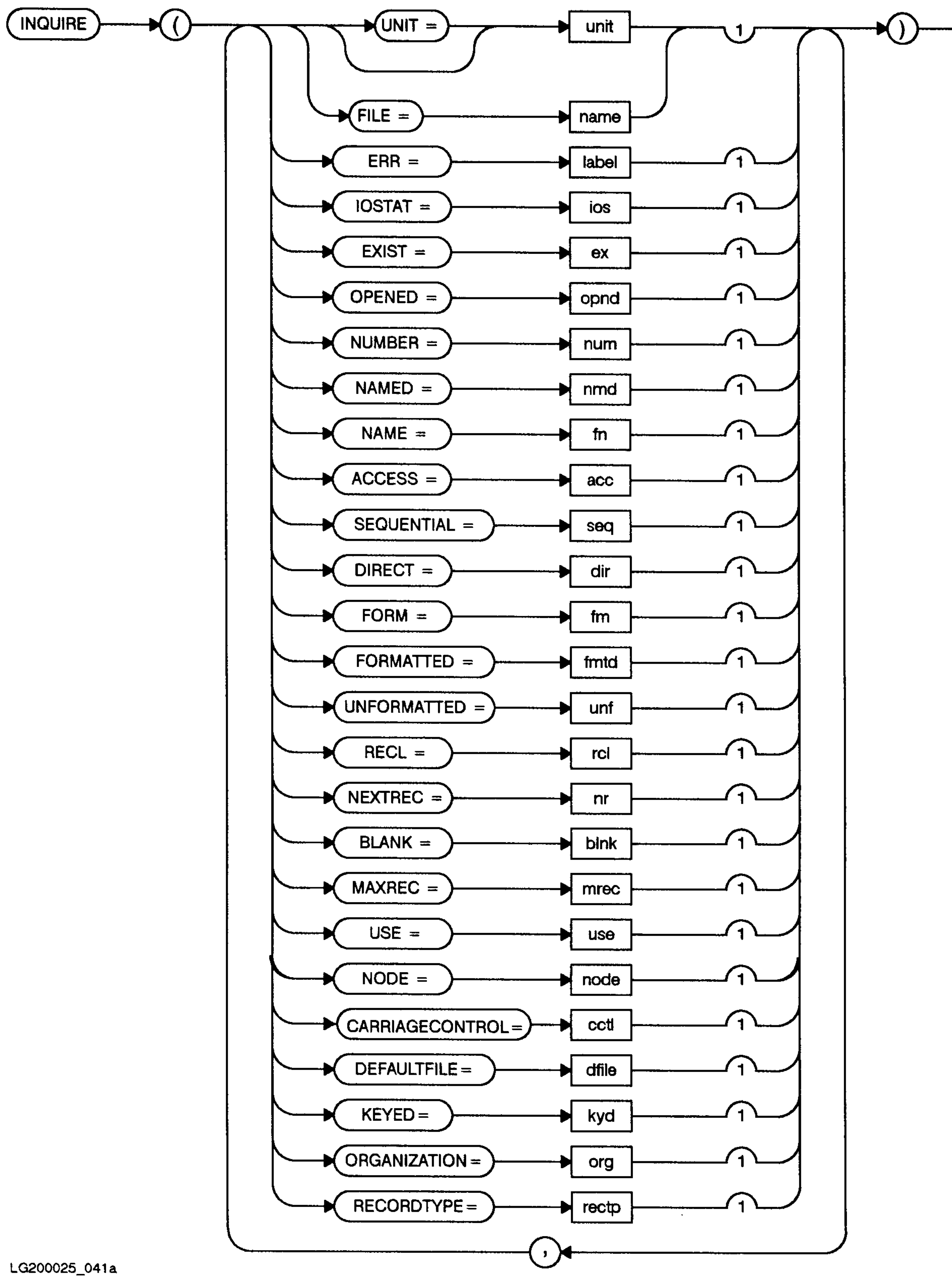
Files may also included with a compiler directive. Refer to “INCLUDE Directive” in Chapter 7 for more information.

Example

```
INCLUDE 'specs'
```

INQUIRE Statement (Executable)

The INQUIRE statement provides information about selected properties of a file or unit number.



LG200025_041a

Item	Description/Default	Restrictions
unit	Unit number of a sequential file.	Integer expression ≥ 0.
name	Specifies file name for inquiry by file name.	Character expression.
label	Control transfers to the specified executable statement if an error condition exists on the named file or unit.	Must be the statement label of an executable statement in the same program unit.
ios	*ios* = zero if no error; *ios* = positive value if error condition exists.	Integer variable, array element, or scalar record field.
ex	*ex* = true if named file exists; *ex* = false otherwise.	LOGICAL*4 variable, array element, or scalar record field.
opnd	*od* = true if named file or unit has been opened; *od* = false otherwise.	LOGICAL*4 variable, array element, or scalar record field.
num	FORTRAN logical unit number of the external named file; if no unit is connected to the named file, *num* is undefined.	INTEGER*4 variable, array element, or scalar record field.
nmd	*nmd* = true if specified unit is not a scratch file; *nmd* = false otherwise.	LOGICAL*4 variable, array element, or scalar record field.
fn	Returns predefined system file name or name used in OPEN statement. If the file has no name or is not connected, *fn* is undefined.	Character variable, array element, substring, or scalar record field.
use	See "Semantics".	Character variable, array element, substring, or scalar record field.
acc	Returns `'SEQUENTIAL'` or `'DIRECT'`, depending upon whether specified unit or file is connected for sequential or unit access, respectively. If the file is not connected, *acc* is undefined.	Character variable, array element, substring, or scalar record field.
seq	Returns `'YES'` if connected for sequential access, `'NO'` if not connected for sequential access, and `'UNKNOWN'` if the processor is unable to determine the access type.	Character variable, array element, substring, or scalar record field.
dir	Returns `'YES'` if connected for direct access, `'NO'` if not connected for direct access, and `'UNKNOWN'` if the	Character variable, array element, substring, or scalar record field.

Item	Description/Default	Restrictions
fm	Returns `'FORMATTED'` if connected for formatted data transfer, `'UNFORMATTED'` if connected for unformatted data transfer, and is undefined if the file is not connected.	Character variable, array element, substring, or scalar record field.
fmtd	Returns `'YES'` if connected for formatted data transfer, `'NO'` if connected for unformatted data transfer, and `'UNKNOWN'` if the processor is unable to determine the format of data transfer.	Character variable, array element, substring, or scalar record field.
unf	Returns `'YES'` if connected for unformatted data transfer, `'NO'` if connected for formatted data transfer, and `'UNKNOWN'` if the processor is unable to determine the form of data transfer.	Character variable, array element, substring, or scalar record field.
rcl	Returns the record length of the specified unit or file connected for direct access, measured in bytes;. If the file is not connected for direct access, *rcl* is undefined.	INTEGER*4 variable, array element, or scalar record field.
nr	*nr* is assigned the next record number to be read or written on the specified unit or file; if no records have been read or written, *nr* = 1; if the file is not connected for direct access or its status is indeterminate, *nr* is undefined.	INTEGER*4 variable, array element, or scalar record field.
blnk	Returns `'ZERO'` or `'NULL'`, depending upon the blank control in effect. If the specified file is not connected or not connected for formatted data transfer, *blnk* is undefined.	Character variable, array element, substring, or scalar record field.
mrec	See "Semantics".	Integer variable, array element, or scalar record field.
node	See "Semantics".	Integer variable, array element, or scalar record field.
cctl	See "Semantics".	Value not checked.
dfile	See "Semantics".	Value not checked.

Semantics

If the ERR specifier is present and an error occurs during the execution of the INQUIRE statement, control transfers to the specified statement rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the *ios* variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

Either the UNIT or FILE specifier, but not both, must be present in the specifier list. If the prefix `UNIT=` is omitted and the unit specifier is present, *unit* must be the first item in the list.

Most of the information described in the syntax table is assigned through the OPEN statement; see "OPEN Statement (Executable)".

When a variable is specified in the syntax table as undefined, its value may or may not be changed from its previous value by the INQUIRE statement. Therefore, the value is meaningless if the file or unit either does not exist or cannot be accessed at the time the INQUIRE is executed.

The following specifiers, extensions to the ANSI 77 standard, are included for compatibility with programs originally written in another version of FORTRAN.

CARRIAGECONTROL	MAXREC	RECORDTYPE
DEFAULTFILE	NODE	USE
KEYED	ORGANIZATION	

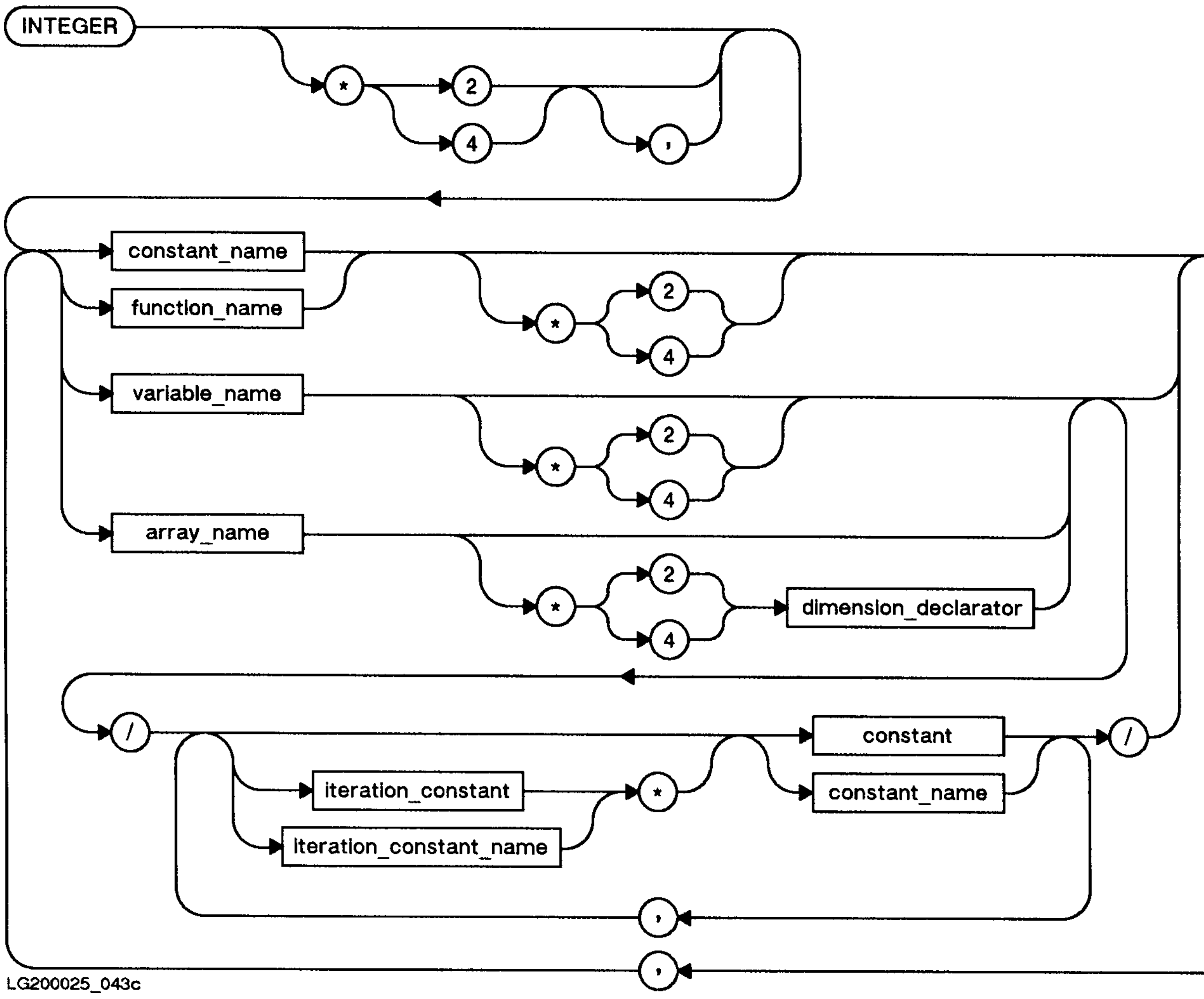
If used in a program, their syntax is checked, but they are otherwise ignored by the compiler.

ACCESS=acc will return acc='SEQUENTIAL' for files opened with ACCESS='APPEND'.

INTEGER Statement (Nonexecutable)

The INTEGER statement is a type specification statement that explicitly assigns the INTEGER*2 and INTEGER*4 data types to symbolic names, and optionally assigns initial values to variables.

The following syntax includes the INTEGER, INTEGER*2, and INTEGER*4 statements.



Semantics

As an extension to the ANSI 77 standard, a length specifier can follow the item being declared. This specification overrides the length implied by the type statement. If the item being declared is an array name with a dimension declarator, the length specifier precedes the dimension declarator.

The INTEGER*2 statement, an extension to the ANSI 77 standard, declares items to be 2-byte integers. The INTEGER*4 statement, also an extension to the ANSI 77 standard, declares items to be 4-byte integers.

Note

By default, the INTEGER statement is equivalent to the INTEGER*4 statement. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make INTEGER equivalent to INTEGER*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See Chapter 7, "Compiler Directives" for further details.

If an array declarator is specified in a type statement, the declarator for that array must not appear in any other specification statement (such as DIMENSION). If only the array name is specified, an array declarator must appear within a DIMENSION or COMMON statement.

Each symbolic name can appear in a type statement only once.

Examples	Notes
`INTEGER run,time`	The variables `run` and `time` are 4-byte integers.
`INTEGER*2 rn,hours(4,5)`	The variable `rn` and each element of the two- dimensional array `hours` are short integers.
`INTEGER counter*2,index*4,matx(4,5)*2`	The variable `counter` and each element of the two-dimensional array `matx` are short integers. `index` is a long integer.

INTEGER*2 Statement (Nonexecutable)

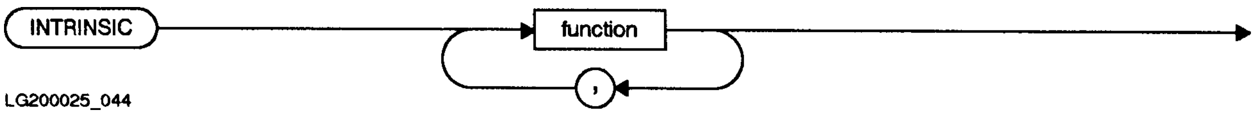
The INTEGER*2 statement, which is an extension to the ANSI 77 standard, is a special case of the INTEGER statement. See "INTEGER Statement (Nonexecutable)" for details.

INTEGER*4 Statement (Nonexecutable)

The INTEGER*4 statement, which is an extension to the ANSI 77 standard, is a special case of the INTEGER statement. See "INTEGER Statement (Nonexecutable)" for details.

INTRINSIC Statement (Nonexecutable)

The INTRINSIC statement identifies a name as representing an intrinsic function and permits the name to be used as an actual argument.



LG200025_044

Item	Description/Default	Restrictions
function	Name of an intrinsic function.	Each name can appear once only in a given INTRINSIC statement and in at most one INTRINSIC statement within a given program unit.

The INTRINSIC statement provides a means of using intrinsics as actual arguments. The INTRINSIC statement is necessary to inform the compiler that these names are intrinsic names and not variable names. Whenever an intrinsic name is passed as a function parameter, it must be placed in an INTRINSIC statement in the calling program.

The names of intrinsic functions for type conversion— CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, SNGL— for logical relationships— LGE, LGT, LLE, LLT— and for choosing the largest or smallest value— AIMAX0, AIMIN0, AJMAX0, AJMIN0, AMAX0, AMAX1, AMIN0, AMIN1, IMAX0, IMAX1, IMIM0, JMAX0, JMAX1, JMIN0, JMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1— must not be used as actual arguments.

A name must not appear in both an EXTERNAL and an INTRINSIC statement in the same program unit.

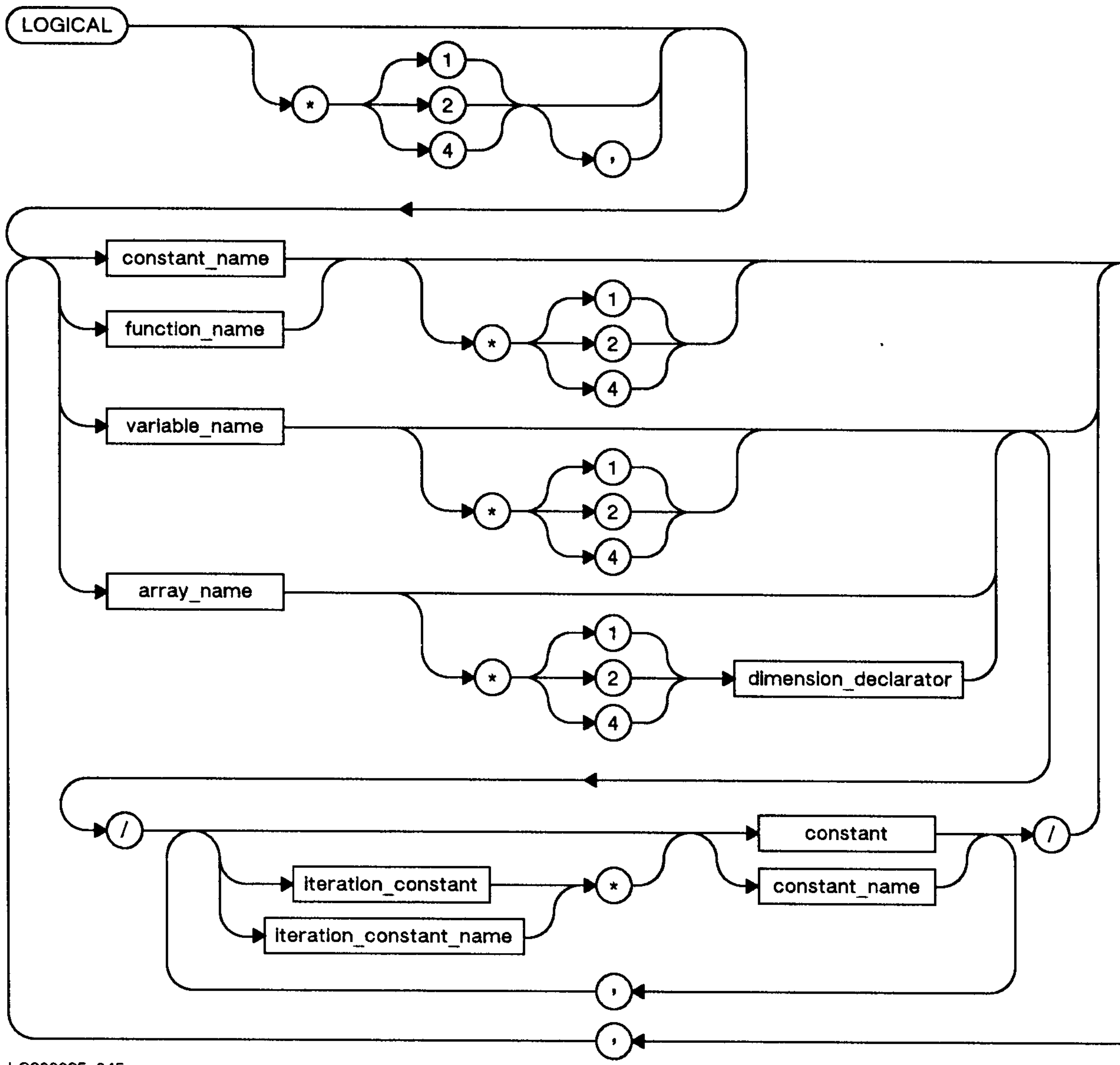
Example	Notes

`INTRINSIC SIN,TAN`
`CALL MATH(SIN,TAN)`
	The INTRINSIC statement informs the compiler that `SIN` and `TAN` are intrinsics.

LOGICAL Statement (Nonexecutable)

The LOGICAL type specification statement explicitly assigns the LOGICAL*1, LOGICAL*2, and LOGICAL*4 data types to symbolic names, and optionally assigns initial values to variables.

The following syntax includes the LOGICAL, LOGICAL*1, LOGICAL*2, and LOGICAL*4 statements. See also "BYTE Statement (Nonexecutable)".



LG200025_045c

Semantics

The LOGICAL*1, LOGICAL*2, and LOGICAL*4 statements are extensions to the ANSI 77 standard. LOGICAL*1 declares items to be 1-byte logicals, LOGICAL*2 declares them to be 2-byte logicals, and LOGICAL*4 declares them to be 4-byte logicals.

Note

By default, the LOGICAL statement is equivalent to the LOGICAL*4 statement. This is the same as the effect of the LONG compiler directive. The SHORT compiler directive may be used to make LOGICAL equivalent to LOGICAL*2. See Chapter 7 for further details. In addition, compiler run-string options can have the same effect. See "Compiler Options" for further details.

If an array declarator is specified in a type statement, the declarator for that array must not appear in any other specification statement (such as DIMENSION). If only the array name is specified, an array declarator must appear within a DIMENSION or COMMON statement.

As an extension to the ANSI 77 standard, the length specifier can follow the item being declared. This specification overrides the length implied by the type statement. If the item being declared is an array name with a dimension declarator, the length specifier precedes the dimension declarator.

Each symbolic name can appear in a type statement only once.

As an extension to the ANSI 77 standard, you can initialize variables or arrays in a type declaration statement by enclosing the initialization values between slashes. See "DATA Statement (Nonexecutable)" for details about initialization.

Examples	**Notes**
`LOGICAL is_ok*2,error*4,bool`	`is_ok` is a 2-byte logical variable. `error` and `bool` are both 4-byte logical variables.
`LOGICAL ok*4/.TRUE./`	`ok` is a 4-byte logical variable, initialized to the value true.
`LOGICAL*2 bool(10)/10*.FALSE./`	`bool` is an array of 10 2-byte logical elements, each initialized to the value false.

LOGICAL*1 Statement (Nonexecutable)

The LOGICAL*1 statement, which is an extension to the ANSI 77 standard, is a special case of the LOGICAL statement. See “LOGICAL Statement (Nonexecutable)” for details. The LOGICAL*1 statement is equivalent to the BYTE statement, which is described in “BYTE Statement (Nonexecutable)”.

LOGICAL*2 Statement (Nonexecutable)

The LOGICAL*2 statement, which is an extension to the ANSI 77 standard, is a special case of the LOGICAL statement. See “LOGICAL Statement (Nonexecutable)” for details.

LOGICAL*4 Statement (Nonexecutable)

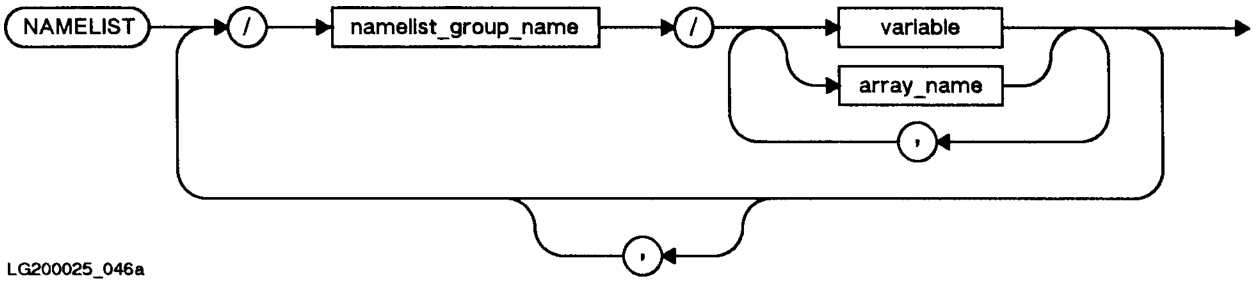
The LOGICAL*4 statement, which is an extension to the ANSI 77 standard, is a special case of the LOGICAL statement. See “LOGICAL Statement (Nonexecutable)” for details.

MAP Statement (Nonexecutable)

The MAP statement begins a MAP statement block. For more information, refer to “STRUCTURE Statement (Nonexecutable)”.

NAMELIST Statement (Nonexecutable)

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



Item	Description/Default	Restrictions
namelist_group_name	Symbolic name for the variables or arrays to be read or written in namelist-directed I/O.	None.
variable *array_name*	List of variables or array names separated by commas, that are associated with the preceding *namelist_group_name*.	Array elements, assumed-sized arrays, adjustable arrays, record references, and character substrings are not permitted.

The namelist variables or arrays can be of any data type and can be explicitly or implicitly typed. A variable or array name can appear more than one namelist.

It is not necessary that an input record be read in for every entity in an associated namelist. However, input of variables names not found in the namelist is an error. Records are written in the order they appear in the namelist.

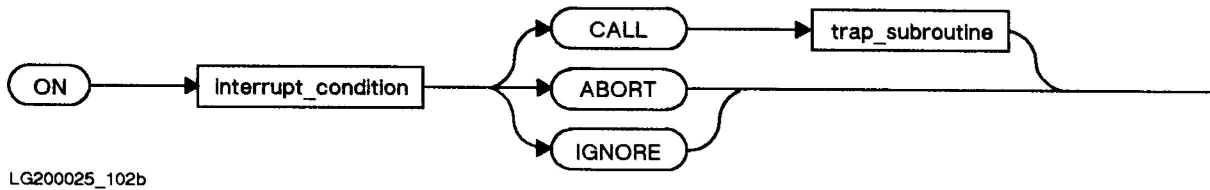
Example	Notes

`INTEGER I,K(30)`
`CHARACTER*20 A`
`NAMELIST /FOO/I,K,A/BOO/A,I`
	This NAMELIST statement specifies two group names, `FOO` and `BOO`.

See "Namelist-Directed Input/Output" in Chapter 4 for further information about namelist-directed I/O.

ON Statement (Executable)

The ON statement specifies the action to be taken following a subsequent interruption of program execution.



LG200025_102b

The *interrupt_condition* specifies the interrupt to be handled, such as an arithmetic error or a keyboard interrupt.

Parameters

interrupt_condition	Keywords specifying an interrupt condition, as given in Table 3-8.
trap_procedure	A procedure that will be executed if the specified *interrupt_condition* occurs following the execution of the ON statement.

Before an interrupt can be trapped, the flow of control must pass through an ON statement that specifies the particular interrupt condition. Once established, an interrupt trap can only be changed by another ON statement that specifies the same interrupt condition.

Table 3-8. Interrupt Conditions

Type of Trap	Interrupt_Condition Keywords	Equivalent Keywords
Arithmetic	REAL*4 DIV 0	REAL DIV 0
	REAL*4 OVERFLOW	REAL OVERFLOW
	REAL*4 UNDERFLOW	REAL UNDERFLOW
	REAL*4 INEXACT	REAL INEXACT
	REAL*4 ILLEGAL	REAL ILLEGAL
	REAL*8 DIV 0	DOUBLE PRECISION DIV 0
	REAL*8 OVERFLOW	DOUBLE PRECISION OVERFLOW
	REAL*8 UNDERFLOW	DOUBLE PRECISION UNDERFLOW
	REAL*8 INEXACT	DOUBLE PRECISION INEXACT
	REAL*8 ILLEGAL	DOUBLE PRECISION ILLEGAL
	REAL*16 DIV 0	(none)
	REAL*16 OVERFLOW	(none)
	REAL*16 UNDERFLOW	(none)
	REAL*16 INEXACT	(none)
	REAL*16 ILLEGAL	(none)
	INTEGER*2 DIV 0	INTEGER DIV 0 [2]
	INTEGER*2 OVERFLOW [1]	INTEGER OVERFLOW [1,2]
	INTEGER*4 DIV 0	INTEGER DIV 0 [2]
	INTEGER*4 OVERFLOW [1]	INTEGER OVERFLOW [1,2]
System	SYSTEM ERROR	(none)
Basic External Function	EXTERNAL ERROR	(none)
Internal Function	INTERNAL ERROR	(none)
Control-Y	CONTROLY	(none)

Notes:

1. There is no check for integer overflows unless the CHECK_OVERFLOW directive is included to generate the overflow-checking code.
2. If INTEGER is specified, the trap handling is set for the default integer type, as defined by the SHORT or LONG compiler directive.

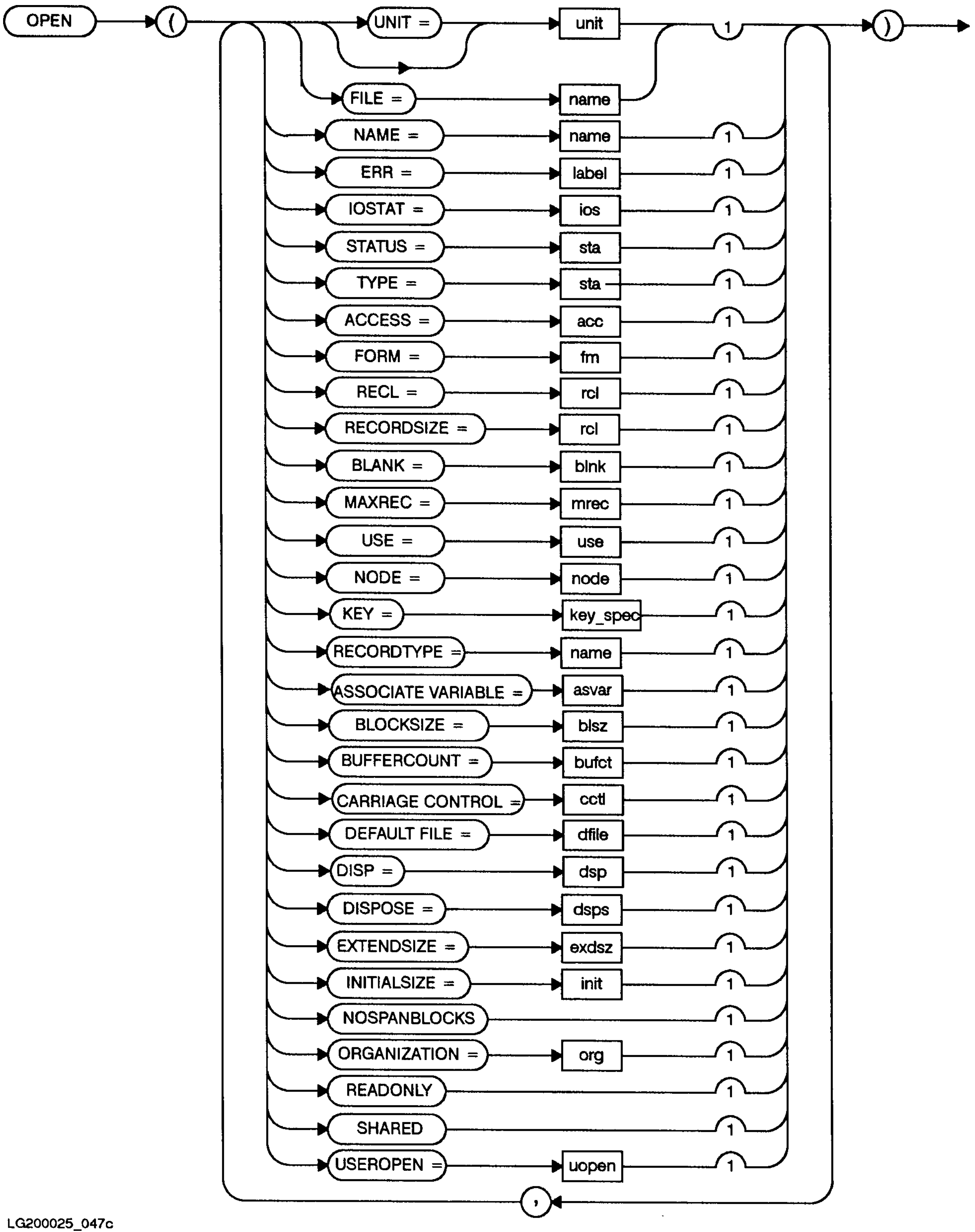
There are three possible actions:

- The ABORT option causes the program to abort.
- The CALL option specifies a subroutine to be executed.
- The IGNORE option causes the interrupt to be ignored.

For further information about the range of values for *interrupt_condition* and the details of the actions, as well as sample programs for trapping external and internal errors, see "Trapping Run-Time Errors" in Chapter 9 and refer to the *HP Compiler Library/iX Reference Manual*.

OPEN Statement (Executable)

The OPEN statement establishes a connection between a unit number and a file. It also establishes or verifies the properties of a file.



LG200025_047c

Item	Description/Default	Restrictions
unit	Specifies unit number.	Integer expression ≥ 0.
name	Character variable.	May be fixed or variable.
label	Control transfers to specified executable statement if error encountered on OPEN.	Must be the statement label of an executable statement in the same program unit.
ios	*ios* = 0 if no error; *ios* = positive value if error condition exists.	Integer variable, array element, or scalar record field.
sta	Specifies file as `'OLD'`, `'NEW'`, `'SCRATCH'`, or `'UNKNOWN'` (default). See Note 1.	Character variable, array element, substring, or scalar record field.
acc	Specifies file access to be `'DIRECT'`, `'KEYED'`, or `'SEQUENTIAL'` (default). See Note 2.	Character variable, array element, substring, or scalar record field.
fm	Specifies data format to be `'FORMATTED'` or `'UNFORMATTED'`. If absent and `ACCESS='SEQUENTIAL'` is specified, `'FORMATTED'` is assumed; If absent and `ACCESS='DIRECT'` is specified, `'UNFORMATTED'` is assumed. If absent and `KEYED` is specified, `UNFORMATTED` is assumed.	Character variable, array element, substring, or scalar record field.
rcl	Specifies record length for direct access and ISAM files; length is measured in bytes.	Numeric expression.
blnk	Specifies treatment of blanks within numbers in input. If `'NULL'` (default), blanks are ignored. If `'ZERO'`, blanks are treated as zeros.	Character variable, array element, substring, or scalar record field.
mrec	See "Semantics".	Integer variable, array element, or scalar record field.
use	See "Semantics".	Character variable, array element, substring, or scalar record field.
node	See "Semantics".	Integer variable, array element, or scalar record field.
key_spec	See "Semantics".	Integer variable, character expression, or scalar record field.
asvar	See "Semantics".	Value not checked.
blsz	See "Semantics".	Value not checked.
bufct	See "Semantics".	Value not checked.
cctl	See "Semantics".	Value not checked.
dfile	See "Semantics".	Value not checked.

Item	Description/Default	Restrictions
exdsz	See "Semantics".	Value not checked.
init	See "Semantics".	Value not checked.
org	See "Semantics".	Value not checked.
uopen	See "Semantics".	Value not checked.

Note 1			
If	**=**	**then the**	**and**
`STATUS`	`'OLD'`	FILE specifier is required	The file must exist.
	`'NEW'`	FILE specifier is required	The file named must not exist.
	`'SCRATCH'`	FILE specifier must not be present	A scratch file is created.
If	**=**	**and if the**	**then**
`STATUS`	`'UNKNOWN'`	FILE specifier is present	The file named is created if it does not already exist.
		FILE specifier is not present	A nondisk unit is connected to the unit specified.

Note 2			
If	**=**	**then the**	**and the**
`ACCESS`	`'SEQUENTIAL'`	RECL specifier may be present	File is opened for sequential access.
	`'DIRECT'`	RECL specifier is required	File is opened for direct access.
	`'KEYED'`	RECL specifier is required. RECORDTYPE must be fixed.	File is opened as a fixed length ISAM file.
	`'APPEND'`	File is opened for sequential access beginning after the last record of the file.	If ACCESS='APPEND' is specified with READONLY, a runtime error will occur.

Semantics

The name field can also be the ASCII representation of a device file.

The UNIT specifier is required in the keyword list. If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. At most one each of the other items can appear in the keyword list.

If the ERR specifier is present and an error occurs during execution of the OPEN statement, control transfers to the specified statement rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the *ios* variable and the program is not aborted. Refer to Appendix A for IOSTAT error codes.

For the character expressions used with STATUS, ACCESS, FORM, and BLANK, only the first character in each is significant.

The following specifiers, extensions to the ANSI 77 standard, are included for compatibility with programs originally written in another version of FORTRAN.

ASSOCIATEVARIABLE	DISPOSE	NOSPANBLOCKS
BLOCKSIZE	EXTENDSIZE	ORGANIZATION
BUFFERCOUNT	INITIALSIZE	RECORDSIZE
CARRIAGECONTROL	MAXREC	TYPE
DEFAULTFILE	NAME	USE
DISP	NODE	USEROPEN

If used in a program, their syntax is checked, but they are otherwise ignored by the compiler.

Once a file is connected to a unit number, the unit can be referenced by any program unit in the program. If a unit is already connected to an existing file, execution of another OPEN statement for that unit is permitted. If the FILE specifier is absent or the file name is the same, the current file remains connected. Otherwise, an automatic close is performed before the new file is connected to the unit. A redundant OPEN call can be used to change only the value of the BLANK option. However, attempts to change the values of any other specifiers with a redundant OPEN are ignored. A redundant OPEN does not affect the current position of the file.

The same file cannot be connected to two different units. An attempt to open a file that is connected to a different unit by the same name causes an error.

As an extension to the ANSI 77 standard, indexed sequential access (ISAM) is allowed on an OPEN statement. Indexed files can be accessed with a key, which is part of the record. The specifier `KEY=`*key_spec* specifies the length of the key. *key_spec* has the form:

exp1 : *exp2* [: *data_type*]

where *exp1* is the first byte position of the key and *exp2* is the last byte position of the key. *data_type* is the data type of the key and must be integer or character. The length of the key is determined by the expression:

exp2 `-` *exp1* `+ 1`

The following table shows the use of ACCESS, RECL, and RECORDTYPE to determine whether an indexed file is variable or fixed length.

ACCESS	RECL	RECORDTYPE	File Type
'SEQUENTIAL'	Absent		Variable length file
'SEQUENTIAL'	Present		Variable length file (RECL = maximum record length)
'DIRECT'	Absent		Error
'DIRECT'	Present		Fixed length file
'KEYED'	Present	Variable	Error (RECL = maximum record length)
'KEYED'	Present	Fixed	Fixed length index sequential access file
'KEYED'	Absent	Variable	Error (Maximum record length is 2048 bytes)
'KEYED'	Absent	Fixed	Error

By default, files are opened for shared read/write access.

Rewinding a file opened with ACCESS='APPEND' repositions the file pointer at the beginning of the file.

Backspacing a file that is opened with ACCESS='APPEND' can reposition the file pointer beyond the initial access point.

An inquire with ACCESS=acc returns acc='SEQUENTIAL' for files opened with ACCESS='APPEND'.

If ACCESS='APPEND' is specified with READONLY, a runtime error will occur.

The READONLY specifier causes the file to be opened for read only access. READONLY can be specified on a file to prevent writing into it by accident. Any attempt to write to a read-only file generates a "FILE SYSTEM ERROR" message.

The SHARED specifier explicitly sets the file for shared access. This permits the file to be shared by multiple programs. Since shared is also the default condition, SHARED has no effect.

Note

When a file is opened with the unit specifier specified, but with no file specifier, a scratch file is opened. Therefore, the following two statements are equivalent:

```
OPEN (UNIT=19)
OPEN (UNIT=19, STATUS='SCRATCH')
```

Examples	Notes

`OPEN (10,FILE='inv',`
`1ACCESS='SEQUENTIAL',`
`2ERR=100,IOSTAT=ios)`
	The file **inv** is connected to unit 10 as a sequential file. If an error occurs, control transfers to statement 100 and the error code is placed in the variable *ios*.

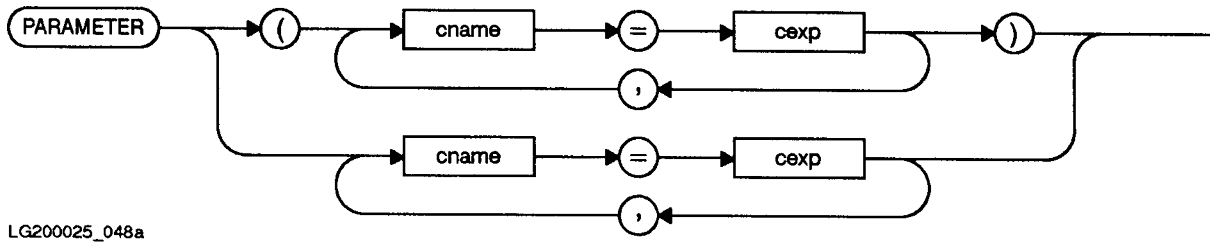
`OPEN (ACCESS='DIRECT',`
`1UNIT=4,RECL=50,`
`2FORM='FORMATTED',FILE=next1)`
	The character variable **next1** contains the name of the file to be connected to unit 4 as a formatted, direct access file with a record length of 50 characters.

```
        program append
        character*2 FN
        character*20 STR, line
        integer recnum

        PARAMETER(LU=15)
        PARAMETER(FN='Afile')
        PARAMETER(STR='This is record')

C       Open file and write to it sequentially
        CALL OPEN_AND_WRITE(LU,FN,STR)

C       Open existing file for APPEND access
        OPEN(unit=LU,file=FN,access='append',iostat=ios,err=99)

        DO I = 26, 50
           WRITE(LU,500) STR, i
        END DO

        REWIND(LU)


        DO I = 1, 50
           READ(LU,500) line, recnum
           IF (line .ne. STR) THEN
              WRITE(6,*) 'line = ',line
              STOP 'READ FAILED - read back incorrect'
           END IF
           IF (recnum .ne. i) THEN
              WRITE(6,*) 'recnum = ',recnum
              STOP 'READ FAILED - read back incorrect'
           END IF
        END DO

        CLOSE(LU,status='keep')

        STOP
```

```
 99   continue
      WRITE(6,*) 'iostat value = ',ios
      STOP 'APPEND OPEN FAILED'
500   FORMAT(2X,A20,I4)
      END

      subroutine OPEN_AND_WRITE(LUNIT,FNAME,STR)
      character*2 FNAME
      character*20 STR

      OPEN(unit=LU,file=FNAME,access='sequential',iostat=ios,err=98)

      DO I = 1, 25
         WRITE(LU,499) STR, i
      END DO



      CLOSE(LU)

 98   continue
      WRITE(6,*) 'iostat value = ',ios
      STOP 'OPEN FAILED'
499   FORMAT(2X,A20,I4)
      END
```

PARAMETER Statement (Nonexecutable)

The PARAMETER statement defines named constants. After a name is defined in a PARAMETER statement, subsequent uses of the name are treated as if the value of the constant was used.



LG200025_048a

Item	Description/Default	Restrictions
cname	Symbolic name that represents a constant.	Name cannot appear in any statement before PARAMETER, except a type statement.
cexp	Constant expression or intrinsic function.	If *cexp* is an intrinsic function, its arguments must be constants.

PARAMETER statements must precede any statement function and executable statements in a program unit.

If the symbolic name *cname* is an integer, real, complex, or logical data type, the corresponding expression *cexp* must be an arithmetic or logical constant expression. If the symbolic name *cname* is a character data type, the corresponding expression *cexp* must be a character constant expression.

As an extension to the ANSI 77 standard, the following FORTRAN intrinsics can be used in *cexp*. When used in *cexp*, these intrinsics must have constant arguments and the type of their return value must be the same as that of *cname*.

ABS	IAND	MAX
CHAR	ICHAR	MIN
CMPLX	IMAG	MOD
CONJG	IOR	NOT
DCMPLX	ISHFT	SIGN
DIM	IXOR	

Each *cname* is the symbolic name of a constant that is defined with the value of the expression *cexp* appearing to the right of the equal sign, in accordance with the rules for assignment statements. Any symbolic name of a constant that appears in an expression *cexp* must have been defined previously in the same or a different PARAMETER statement in the same program unit.

A symbolic name of a constant must not be defined more than once in a program unit.

If a symbolic name of a constant is not of the default implied type, its type must be specified by a type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. If the length specified for the symbolic name of a constant of type character is not the default length of one, its length must be specified in a type statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constant. Its type and length must not be changed by subsequent statements, including IMPLICIT statements. If a symbolic name of type `CHARACTER*(*)` is defined in a PARAMETER statement, its length is the length of the expression assigned to it.

Once such a symbolic name is defined, that name can appear in any subsequent statement of the defining program unit as a constant in an expression or DATA statement. A symbolic name of a constant must not be part of a format specification.

A symbolic name in a PARAMETER statement can identify only the corresponding constant in that program unit.

Examples	Notes
`   PARAMETER (minval=-10,maxval=50)`	
`   PARAMETER (debug=.TRUE.)`	
`   PARAMETER (file='WELCOM')`	

`   INTEGER lower,upper`
`   PARAMETER (lower=0, upper=7)`
`   DIMENSION a (lower:upper)`
`   DO 10 i=lower,upper`
`   a (i) = 1.0`
`10 CONTINUE`	

`   PARAMETER (pi=3.14159)`
`   radius = diameter/2`
`   area = pi *(radius**2)`	

`   CHARACTER bell`
`   PARAMETER (bell = CHAR(7))`
	`CHAR` in constant expression.

`   INTEGER case_shift`
`   PARAMETER (case_shift = ICHAR('a') - ICHAR('A'))`
	`ICHAR` in constant expression

`   COMPLEX complex_two`
`   PARAMETER (complex_two = 2)`
	Arithmetic conversion performed.

`   PARAMETER (limit = 1000)`
`   PARAMETER (limit_plus_1 = limit+1)`
	Legal use of previously defined name.

Example

The following program:

```
      PROGRAM parameters

      LOGICAL      first_name_greater, scnd_name_greater
      CHARACTER    ch*(*), name1*(*), name2*(*)
      INTEGER      length

      PARAMETER    (ch     = 'Guess my length')
      PARAMETER    (name1  = 'William',
     +              name2  = 'David')

      PARAMETER    (length = LEN(ch))

C  Either form of lexical compare is allowed in PARAMETER

      PARAMETER    (first_name_greater = LGT(name1, name2),
     +              scnd_name_greater  = name2 .GT. name1)

      WRITE (6,10) ch, length
      IF (first_name_greater) THEN
         WRITE (6,*) name1, 'is lexically greater than', name2
      ELSE IF (scnd_name_greater) THEN
         WRITE (6,*) name2, 'is lexically greater than', name1
      ELSE
         WRITE (6,*) name1, 'and', name2, 'have the same name'
      END IF
 10   FORMAT (' The length of ''',(A),''' is ',I2)
      END
```

produces the following output:

```
The length of 'Guess my length' is 15
William is lexically greater than David
```

Alternate PARAMETER Statement (Nonexecutable)

An alternate version of the PARAMETER statement is included for compatibility with other versions of FORTRAN. The alternate version differs from the ANSI 77 standard in two ways:

- The parameter list is not bounded by parentheses.
- The type of the constant *cexp* determines the type of *cname* (regardless of explicit or implicit typing).

Alternate PARAMETER statements must precede any executable statements in a program unit.

The following example illustrates the alternate PARAMETER statement. The output follows the example.

Example

```
PROGRAM showpars
IMPLICIT INTEGER (i), REAL (r)
PARAMETER i1 = 'AB'   ! Alternate; i1 is type character
PARAMETER (i2 = 2.0)  ! Standard;  i2 is type integer
PARAMETER i3 = 3.0    ! Alternate; i3 is type real
PARAMETER r1 = 6      ! Alternate; r1 is type integer
PARAMETER (r2 = 6)    ! Standard;  r2 is type real
i4 = 4                ! First executable statement
PARAMETER i5 = 5      ! Assignment statement, not PARAMETER
                      ! PARAMETER i5 is a variable
WRITE (*,*) i1, i2, i3, i4, i5, PARAMETER i5, r1, r2
END
```

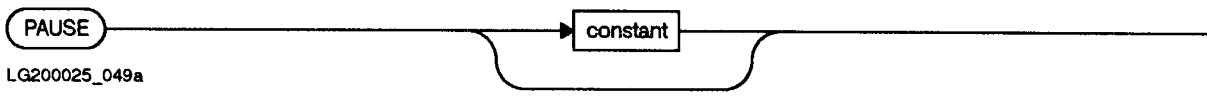
Output:

```
AB 2 3.0 4 0 5.0 6 6.0
```

Note that `PARAMETER i5` in the example above is a variable because it follows an executable statement. (FORTRAN assumes meaning from context and has no reserved words.)

PAUSE Statement (Executable)

The PAUSE statement causes a temporary break in program execution.



LG200025_049a

Item	Description/Default	Restrictions
constant	Integer or character constant to be displayed in the PAUSE message.	Cannot be a constant name or a constant expression.

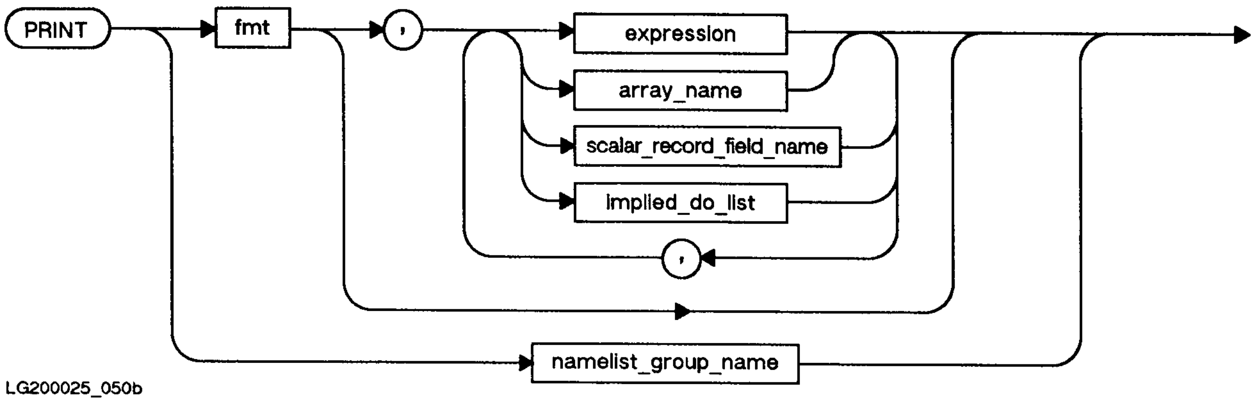
The PAUSE statement optionally writes a message and, if standard input is a terminal, waits for the user to request the program to continue.

If *constant* is omitted, no message is written. If a constant is given, the message `PAUSE` followed by the constant value is written.

On MPE/iX, the PAUSE statement causes a program break if the program is operating in interactive mode, but does not break if the program is operating in batch mode. In either case, the constant given with the PAUSE statement is printed on the standard list device. In interactive mode, the PAUSE statement serves the same function as using the BREAK key and all MPE/iX commands allowed in BREAK mode can be used. To resume execution of the program in interactive mode, enter the RESUME command.

PRINT Statement (Executable)

The PRINT statement only transfers data from memory to the standard output unit. (FORTRAN unit 6 is preconnected to the standard output device.)



LG200025_050b

Item	Description/Default	Restrictions
fmt	Format designator.	See "Semantics".
implied_do_list	An implied DO loop. Refer to "DO Statement (Executable)".	None.

Semantics

The format designator must be one of the following:

- The statement label of a FORMAT statement.
- An INTEGER*4 variable to which the statement label of a FORMAT statement has been assigned by an ASSIGN statement.
- A character or noncharacter array name that contains the representation of a format specification. The use of a noncharacter array is an extension to the ANSI 77 standard.
- A character expression that evaluates to a valid format string. Variable format descriptors are allowed only in character *constant* expressions.
- An asterisk to specify list-directed output (refer to "List-Directed Input/Output" in Chapter 4).

Each item in the list of data to be transferred must be one of the following:

- A constant
- A variable name
- An expression
- An array name

- An array element
- A substring
- A scalar record field name
- An implied DO loop

The PRINT statement is equivalent to the TYPE statement.

Examples	Notes
`PRINT 10,num,des`	`num` and `des` are printed according to FORMAT statement 10.
`PRINT *,'x=',x`	`'x='` and the value of `x` are printed according to list-directed formatting.

`INTEGER fmt`
`ASSIGN 200 TO fmt`
`PRINT fmt,rat,cat`
	`rat` and `cat` are printed according to FORMAT statement 200.

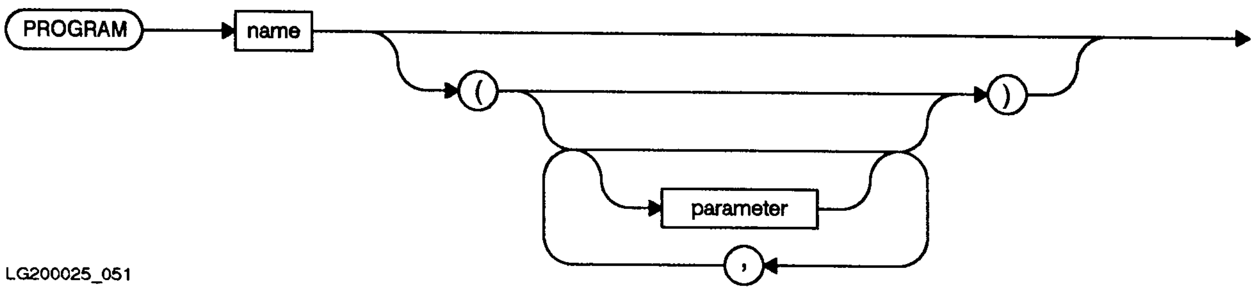
`PRINT '(4I3)',i,j,k*2,330`	`i`, `j`, `k*2`, and the constant `330` are printed according to the format specification in the PRINT statement itself.

`PRINT 100`
`100 FORMAT ("End of report")`
	The character constant in the FORMAT statement is printed.

`PRINT '(" x  SIN(x)   COS(x)"//(I3,2F7.3))',`
`+(i,SIN(i/57.3),COS(i/57.3),i=0,360,15)`
	Prints a literal heading and 25 rows of values as indicated by the implied DO in the output list.

PROGRAM Statement (Nonexecutable)

The PROGRAM statement defines the name of a program. Optionally, as an extension to the ANSI 77 standard, it also defines the formal arguments of the main program in which the statement appears.



Item	Description/Default	Restrictions
name	Name of the program (and its main entry point).	None.
parameter	Optional program argument.	Must be of type CHARACTER*N.

The PROGRAM statement must be the first noncomment statement in a module, except for certain compiler directives, described in Chapter 7.

The MPE/iX RUN command has two optional parameters, PARM and INFO, whose values you can pass to any FORTRAN 77 program. The PARM field is a 16-bit or 32-bit signed integer. The INFO field is a character string of up to 255 characters, including the apostrophes (') or quotation marks ("). You can obtain the values of PARM or INFO in a FORTRAN 77 program by specifying appropriate parameters in the PROGRAM statement. These parameters can specify a variable for PARM, a variable for INFO, or both.

After placing the variables in the PROGRAM statement, you should declare the variables as the correct types. The variables must be local variables in the main program unit. The variable for PARM must be type INTEGER*2 or INTEGER*4. The variable for INFO must be a character variable, expressed as CHARACTER*(*).

The PROGRAM statement can have at most two parameters and they must be used as mentioned above (but can be in any order).

Examples	Notes
`PROGRAM main`	Specifies `main` as the name of the program.
`PROGRAM main()`	Specifies `main` as the name of the program, with no arguments.
`PROGRAM runit(a,b)`	Specifies `runit` as the name of the program, and specifies its arguments as `a,b`. Note that one of these arguments must be an INTEGER*2 or INTEGER*4 and the other must be a CHARACTER*(*).

READ Statement (Executable)

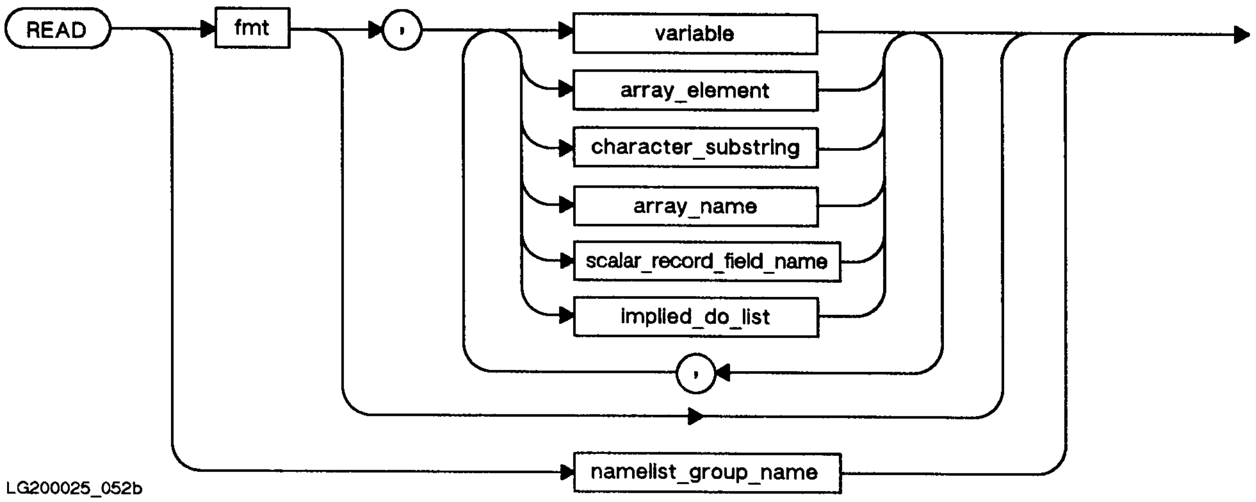
The READ statement transfers data from a file to program variables. There are two kinds of READ statements:

- Standard input READ
- File READ

The standard input READ statement complements the PRINT statement. The file READ statement complements the WRITE statement. A more detailed description of the READ statement is found in Chapter 4.

Standard Input READ Statement (Executable)

The standard input READ statement transfers data to memory from a unit that is designated as the standard input unit. (FORTRAN unit 5 is preconnected to the system standard input, usually the user's terminal.)



Item	Description/Default	Restrictions
fmt	Format designator.	See "Semantics".
namelist_group_name	Symbolic name specifying a list of variables or arrays previously declared in a NAMELIST statement.	None.
implied_do_list	An implied DO loop. Refer to "DO Statement (Executable)".	None.

Semantics

The format designator must be one of the following:

- The statement label of a FORMAT statement.
- An INTEGER*4 variable to which the statement label of a FORMAT statement has been assigned by an ASSIGN statement.
- A character or noncharacter array name that contains the representation of a format descriptor list enclosed in parentheses. The use of a noncharacter array is an extension to the ANSI 77 standard.
- A character expression that evaluates to the representation of a format descriptor list enclosed in parentheses.
- An asterisk, which specifies list-directed output. See "List-Directed Input/Output" in Chapter 4 for details.

Each item in the variable list specifying where the data is to be transferred must be one of the following:

- A variable name.
- An array element name.
- An array name.
- A substring.
- An implied DO loop containing the above items only.
- A scalar record element name.

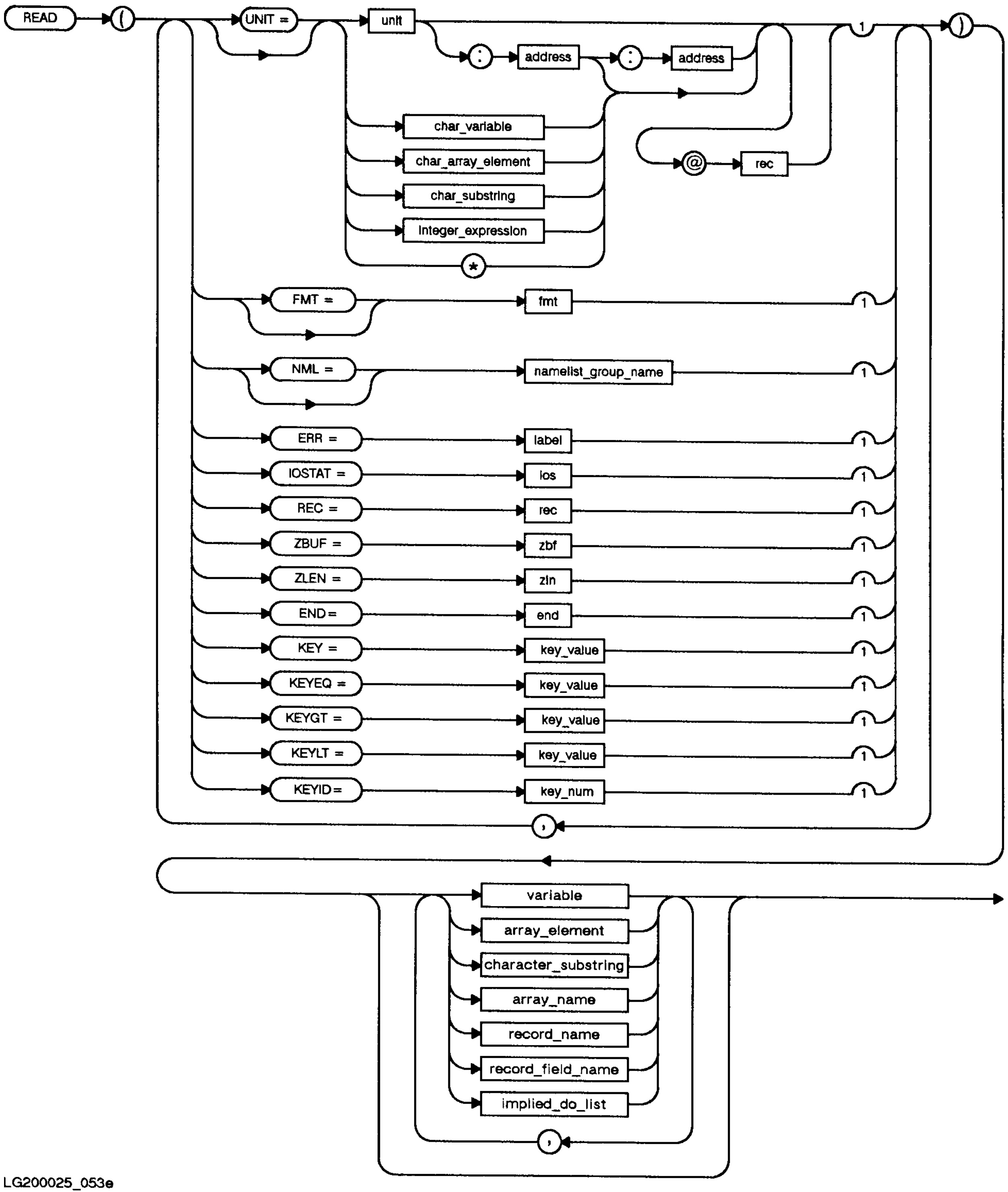
Examples	Notes
`READ 10,num,des`	Reads the values of `num` and `des` according to FORMAT statement 10.
`READ *,a,b,n`	Reads the values of `a`, `b`, and `n` according to list-directed formatting.

`ASSIGN 100 TO fmt`
`READ fmt,al,h1`
	Reads the values of `al` and `h1` according to FORMAT statement 100.

`READ '(3I3)',i,j,k`	Reads the values of `i`, `j`, and `k` according to the format specification in the READ statement itself.
`READ 5`	Skips a record on the standard input device.

File READ Statement (Executable)

The file READ statement transfers data from a file to memory.



LG200025_053e

Item	Description/Default	Restrictions
unit	Arithmetic expression of type integer.	See “Semantics”.
address	Expression specifying unit number of a sequential file.	Must be an integer: zero or positive.
char_variable	Internal file from which input is taken.	Character variable or scalar record field.
char_array_element	Internal file from which input is taken.	Character array element or scalar record field.
char_substring	Internal file from which input is taken.	Character substring.
integer_expression	Expression specifying the unit number of a sequential file.	Must be an integer: zero or positive.
*	Asterisk indicates that the standard input device (unit 5, usually the terminal) is to be used.	None.
fmt	Format designator.	*fmt* must be as specified for a standard input READ above.
namelist_group_name	Symbolic name specifying a list of variables or arrays previously declared in a NAMELIST statement.	Cannot appear in a statement containing a format specifier.
ios	Integer variable or integer array element name for error return.	Must be an integer type.
label	Statement label of an executable statement.	Must be the label of a statement in the same program unit.
rec	Specifies the record number in a direct access file.	If *fmt* is an asterisk, a record specifier must not be present.
zbf	Variable, array name, or array element name.	Extension to ANSI 77 standard; cannot be a character type.
zln	Integer expression used with ZLEN.	Extension to ANSI 77 standard.
end	Statement label of an executable statement.	None.
key_value	See “Semantics”.	Integer variable, character expression, or scalar record field.
key_num	See “Semantics”.	None.

Semantics

A file READ statement must contain a unit specifier and at most one of each of the other specifiers.

If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. This is the unit number for the input device or file.

If the prefix `FMT=` is omitted, *fmt* must be the second item in the list and *unit* (without a prefix) must be the first item.

If *fmt* is omitted and no NML specifier is present, the access is unformatted (binary). *record_name* and *aggregate* variables can only be used in unformatted reads.

If a record number is specified, the unit must be connected for direct access. You can specify a record number through the REC specifier. Note that REC cannot appear with the END or NML specifiers nor with the `FMT=*` form of the FMT specifier. You can also specify a record number with the `@` specifier.

If the ERR specifier is present and an error occurs during execution of the READ statement, control transfers to the specified statement rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for the IOSTAT error codes.

If the END specifier is present and an end-of-file is encountered in a sequential file during the execution of the READ statement, control transfers to the specified statement. In this case, *ios* is set to `-1`.

The ZBUF and ZLEN specifiers and the *address* alternative used as a parameter for the UNIT specifier are extensions to the ANSI 77 standard, and are included for compatibility with programs originally written in another version of FORTRAN. If used in a program, their syntax is checked, but they are otherwise ignored by the compiler.

As an extension to the ANSI 77 standard, indexed sequential access (ISAM) is allowed with a READ statement. The following specifiers are used to establish the desired match criterion to read a record from an indexed file:

`KEY   =` *key_value*
`KEYEQ =` *key_value*
`KEYGT =` *key_value*
`KEYLT =` *key_value*

where *key_value* is an integer value or character expression. Any one of the specifiers can appear in a READ statement. The specifiers can be omitted. If a specifier is not present, the primary key is assumed if it is the first read of the file. Otherwise, the file is read sequentially from the last position of the previous read.

If the KEYID specifier is not present, the primary key is assumed. The subsequent reads do not assume the previous KEYID value for the current read.

As an extension to the ANSI 77 standard, sequential reads (without the REC specifier) are allowed on files open for direct access. If the REC specifier is omitted, a READ statement reads the next record.

Examples	Notes
`READ (8,10)a,b,c`	Reads the values of `a`, `b`, and `c` from the file connected to unit 8 according to FORMAT statement 10.

`ASSIGN 4 TO num`
`READ (UNIT=3,ERR=50,FMT=num)`
	Reads the value of `z` from the file connected to unit 3 according to FORMAT statement 4. If an error occurs, control transfers to statement 50.

`READ (10)x`	Reads the value of `x` from the file connected to unit 10. Because *fmt* is omitted, the data is unformatted.
`READ (10,FMT=*,END=60)b`	Reads the value of `b` from the file connected to unit 10, according to list-directed formatting. If an end-of-file is encountered, control passes to statement 60.
`READ (2,'(I3)',REC=10)i`	Reads the value of `i` from the 10th record of the direct access file connected to unit 2, according to the format specification in the READ statement itself.
`READ (10)`	Skips a record in the file connected to unit 10.

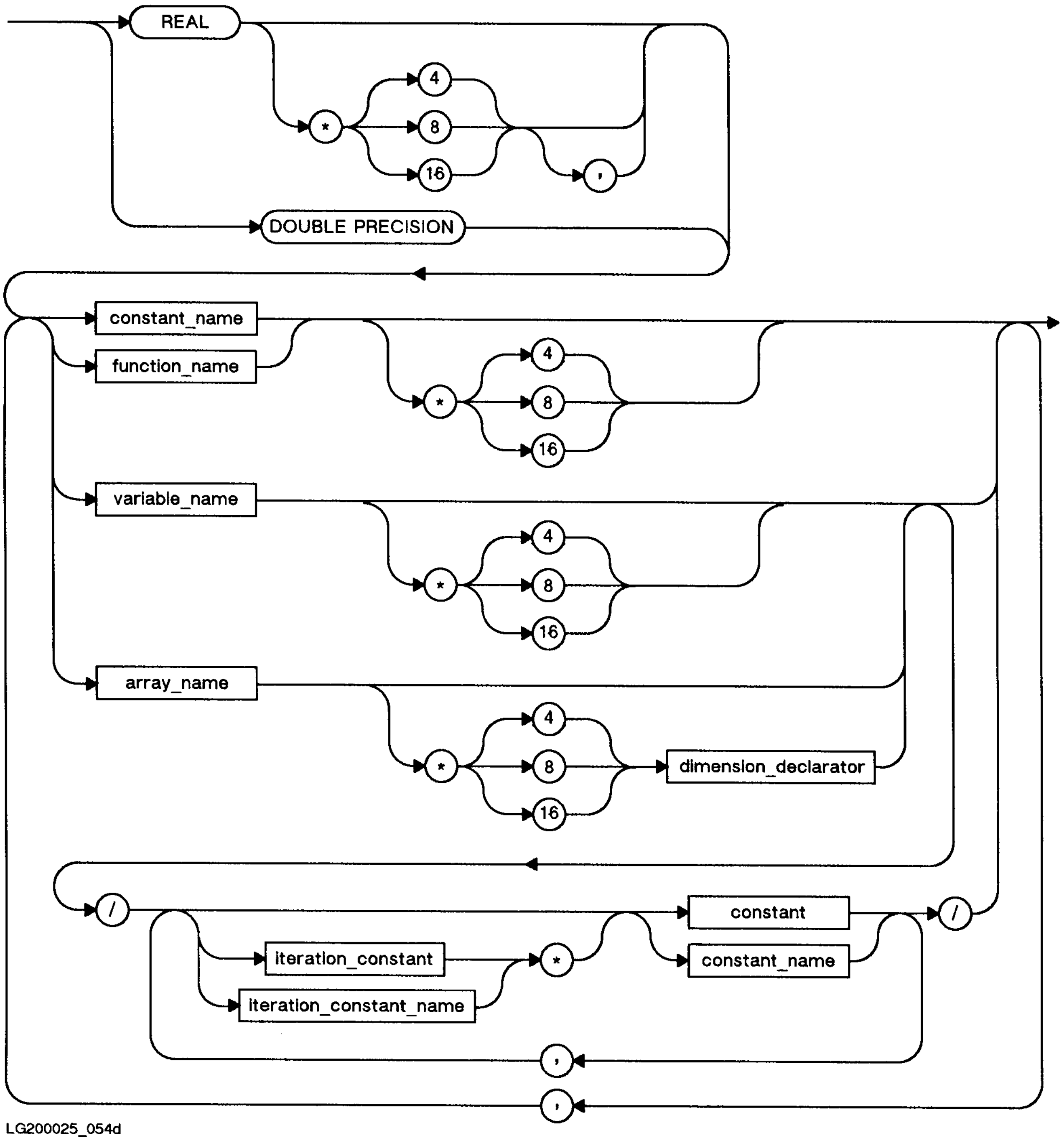
`CHARACTER*8 a`
`REAL b`
`a = '  $27.97'`
`READ (a(4:8),'(F5.2)') b`
	Reads the value of `b` from the character variable `a` according to the format specification in the READ statement itself.

`READ (10,KEYID=0,KEYEQ='100',ERR=101)buf`	Reads ISAM record with primary key value `'100'` into `buf`.

REAL Statement (Nonexecutable)

The REAL type specification statement explicitly assigns the REAL*4, REAL*8, and REAL*16 data types to symbolic names, and optionally assigns initial values to variables.

The following syntax includes the REAL, REAL*4, REAL*8, REAL*16, and DOUBLE PRECISION statements.



LG200025_054d

Semantics

The REAL and REAL*4 statements are equivalent. The DOUBLE PRECISION and REAL*8 statements are equivalent. The REAL*16 statement has no equivalent. The REAL*4, REAL*8, and REAL*16 statements are extensions to the ANSI 77 standard.

As an extension to the ANSI 77 standard, a length specifier can follow the item being declared. This specifier overrides the data length implied by the type statement. If the item is an array name with a dimension declarator, the length specifier precedes the dimension declarator.

If an array declarator is specified in a type statement, the declarator for that array must not appear in any other specification statement (such as DIMENSION). If only the array name is specified, then an array declarator must appear within a DIMENSION or COMMON statement.

Each symbolic name can appear in a type statement only once.

As an extension to the ANSI 77 standard, you can initialize variables or arrays in a type declaration statement by enclosing the initialization values between slashes. The second example below illustrates this method of initialization. See "DATA Statement (Nonexecutable)" for further information on initialization.

Examples	**Notes**

`REAL item`
`DIMENSION item(2,3,5)`
	`item` is a three-dimensional array containing 30 4-byte real elements.

`DOUBLE PRECISION`
`+measure/384E-04/,test/2.1/`
	`measure` is an 8-byte real variable, initialized to `.0384`. `test` is an 8-byte real variable, initialized to `2.1`.

`REAL*16 accurate(5)`	`accurate` is an array containing 5 16-byte real elements.

REAL*4 Statement (Nonexecutable)

The REAL*4 statement, which is an extension to the ANSI 77 standard, is a special case of the REAL statement. See "REAL Statement (Nonexecutable)" for details. The REAL*4 statement is equivalent to the REAL statement.

REAL*8 Statement (Nonexecutable)

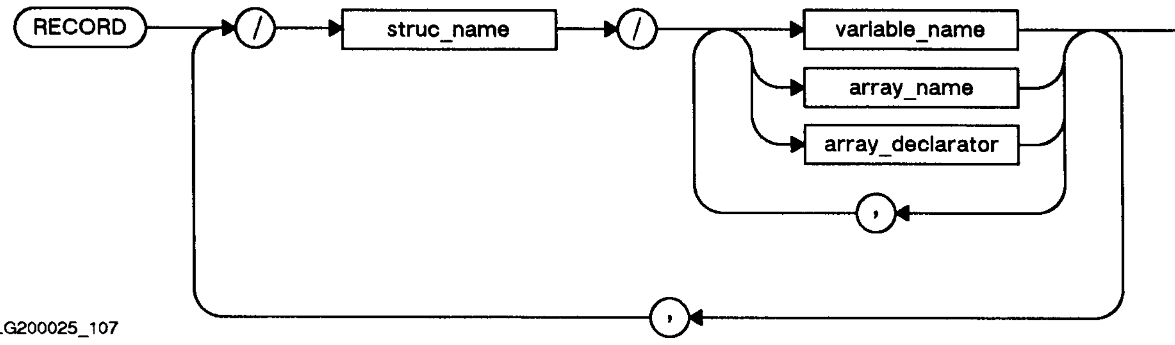
The REAL*8 statement, which is an extension to the ANSI 77 standard, is a special case of the REAL statement. See "REAL Statement (Nonexecutable)" for details. The REAL*8 statement is equivalent to the DOUBLE PRECISION statement.

REAL*16 Statement (Nonexecutable)

The REAL*16 statement, which is an extension to the ANSI 77 standard, is a special case of the REAL statement. See "REAL Statement (Nonexecutable)" for details.

RECORD Statement (Nonexecutable)

The RECORD statement declares a record variable that has the form previously declared in a STRUCTURE statement.



LG200025_107

Item	Description/Default	Restrictions
struc_name	Name of a previously declared structure.	Cannot be the name of a structure currently being declared.
variable_name	Variable name.	None.
array_name	Array name.	None.
array_declarator	Array declaration.	None.

Semantics

Record names can be used in COMMON, DIMENSION, and SAVE statements. They can not be used in DATA, EQUIVALENCE, and NAMELIST statements.

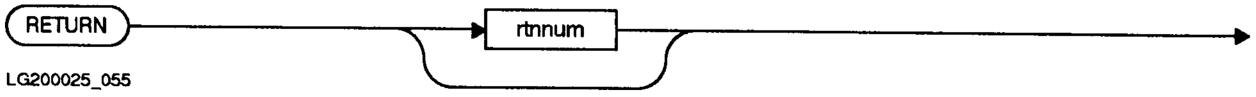
Record field values are initially undefined, but can be initialized in the structure declaration.

Refer to "STRUCTURE Statement (Nonexecutable)" for more information on structures and their use with records.

Examples	Notes
`RECORD/student/rec1,math_student(10)`	`rec1` is a record variable that has the form declared for the `student` structure. `math_student` is an array variable that has ten record elements in the form declared for the `student` structure.

RETURN Statement (Executable)

The RETURN statement transfers control from a subprogram back to the calling program unit.



LG200025_055

Item	Description/Default	Restrictions
rtnnum	Integer expression specifying the alternate return number.	See "Semantics".

Semantics

Normally, control returns from a subroutine to the calling program unit at the statement following the CALL statement. Specifying alternate return statements allows return to the calling program unit at any labeled executable statement within it.

When the RETURN statement occurs in a subroutine subprogram and no alternate return is specified, control returns to the first executable statement following the CALL statement that invoked the subroutine.

When the RETURN statement occurs in a function, control returns to the statement containing the function call. Alternate returns are not allowed in functions.

The scalar expression, *rtnnum*, may have the range of values 1 to n, where n is the number of alternate returns specified in the CALL statement. The value of *rtnnum* identifies the ordinal position of the statement label in the actual argument list of the CALL statement.

The asterisks in the SUBROUTINE statement are for documentation purposes. The number of asterisks should be the same as the number of statement labels in the CALL statement. For consistency with the ANSI standard, if *rtnnum* is a constant, its value should be less than or equal to the number of asterisks in the SUBROUTINE statement. However, if the value of *rtnnum* exceeds the number of asterisks in the SUBROUTINE statement, compilation, load, and execution are not affected. An error is generated if alternate returns are specified and no asterisks appear in the SUBROUTINE statement.

When the value of *rtnnum* is not in the range 1 to n, control returns to the statement following the CALL statement. When a variable or expression represents *rtnnum*, only one asterisk is required in the SUBROUTINE statement, although it is good programming practice to have the number of asterisks in the SUBROUTINE statement always match the number of labels in the CALL statement.

rtnnum may have any numeric data type. If *rtnnum* is not an integer, it will be converted to one. If the $HP1000 ARRAYS compiler directive is in effect, *rtnnum* may be an array name. If *rtnnum* is an array name, the first element of the array will be used.

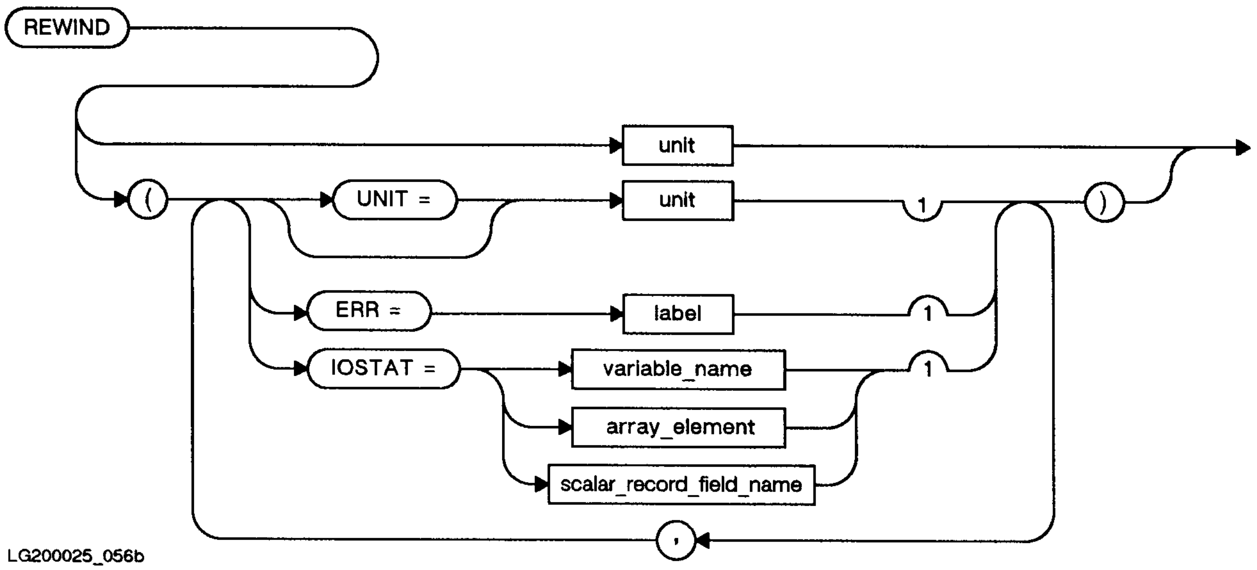
Examples	Notes

`PROGRAM main`
`   ⋮`
`CALL matrx (*10,m,*20,n,k,*30)`
`   ⋮`
`10 ... ! executable statement`
`   ⋮`
`20 ... ! executable statement`
`   ⋮`
`30 ... ! executable statement`
`   ⋮`
`END`
	The CALL statement specifies three possible return labels, plus the normal return point (the statement following the CALL).

`SUBROUTINE matrx(m,n,k,*,*,*)`
`   ⋮`
`k=2`
`   ⋮`
`RETURN k`
`END`
	The SUBROUTINE statement contains a number of asterisks equal to the number of statement labels in the CALL statement.
	
	`k` evaluates to the value 2, causing control to pass to the second alternate return label specified in the CALL statement (20). If `k` evaluates to a value outside the range 1 to 3, control returns to the statement following the CALL statement.

REWIND Statement (Executable)

The REWIND statement positions a sequential file or device at its beginning.



Item	Description/Default	Restrictions
unit	Integer expression specifying unit number of a connected file.	Must be zero or positive.
variable_name *array_element* *scalar_record_field_name*	Error code return.	Must be an integer data type.
label	Statement label.	Must be an executable statement in the same program unit.

Semantics

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for the IOSTAT error codes.

If an error occurs during execution of the REWIND statement and the ERR specifier is present, control transfers to the specified statement rather than aborting the program.

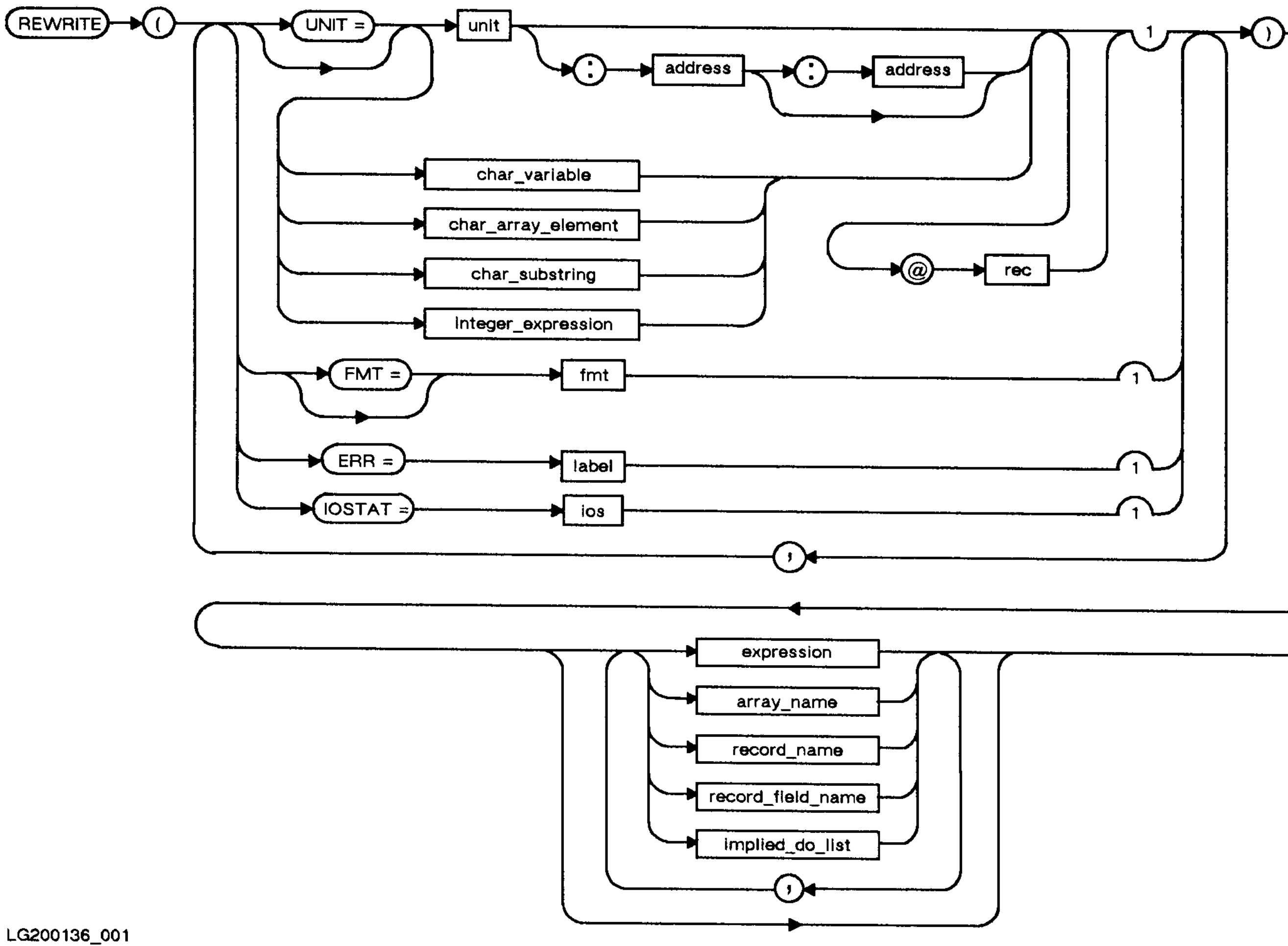
If the file or device is already positioned at its beginning, a REWIND statement has no effect.

As an extension to the ANSI 77 standard, the REWIND statement may be used with files open for direct access. It positions the file before the first record.

Examples	Notes
`REWIND 10`	The file connected to unit 10 is positioned at its beginning.
`REWIND (UNIT=5, IOSTAT=j, ERR=100)`	The file connected to unit 5 is positioned at its beginning. If an error occurs, control transfers to statement 100 and the error code is returned in `j`. If no error occurs, `j` is set to zero and control transfers to the next statement.
`REWIND (UNIT=62, IOSTAT=j)`	The file connected to unit 62 is positioned at its beginning. If an error occurs, the error code is stored in the variable `j`. If no error occurs, `j` is set to zero. In both cases, control transfers to the next statement.
`REWIND (UNIT=12, ERR=100)`	The file connected to unit 12 is positioned at its beginning. If an error occurs, control transfers to statement 100. If no error occurs, control transfers to the next statement.

REWRITE Statement (Executable)

The REWRITE statement is used to update existing records in an ISAM file. The record being updated is the most recent record read from the file by a READ statement. REWRITE unlocks the record if it is locked.



LG200136_001

Item	Description/Default	Restrictions
unit	Integer expression specifying the unit number of a file.	Must be zero or positive.
address	Integer expression.	None.
char_variable	Internal file written.	Character variable or scalar record field.
char_array_element	Internal file written.	Character array element or scalar record field.
char_substring	Internal file written.	Character substring.
integer_expression	Integer expression specifying the unit number of an internal file.	Must be zero or positive.
fmt	Format designator.	*fmt* must be as specified in a PRINT statement.
namelist_group_name	Symbolic name specifying a list of variables or arrays previously declared in a NAMELIST statement.	Cannot appear in a statement containing a format specifier.
ios	Integer variable or array element for error return.	Must be an integer type.
label	Statement label of an executable statement.	Must be the label of a statement in the same program unit.

Semantics

A REWRITE statement must contain a unit number and at most one of each of the other options.

If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. This is the unit number for the output device or file.

If the prefix `FMT=` is omitted, *fmt* must be the second item in the list and the *unit* (without a prefix) must be the first item. If *fmt* is not present and a namelist group name is not specified, the write is unformatted (binary).

record_name and *aggregate* variables can only be used in unformatted writes.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for the IOSTAT error codes.

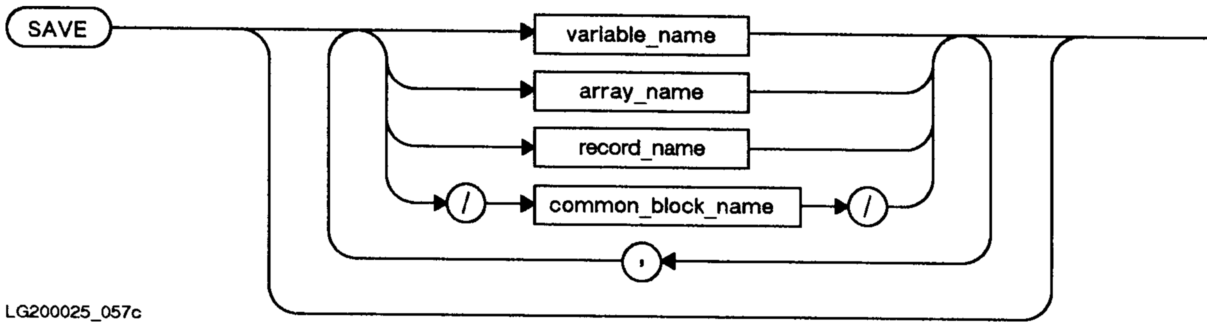
If the ERR specifier is present and an error occurs during execution of the REWRITE statement, control transfers to the specified statement rather than aborting the program.

Examples	**Notes**

`READ (10,KEY='111-22-333',ERR=555)`
`+EMPLOYEE_REC`
`EMPLYEE_REC.BONUS =`
`+EMPLOYEE_REC.SALARY * 0.1`
`REWRITE(10,ERR=556,IOSTAT=I)`
`+EMPLOYEE_REC`
	Updates a record. It reads the record, changes the fields, and then updates the record by using REWRITE.

SAVE Statement (Nonexecutable)

The SAVE statement causes the specified variables in the program unit to maintain their values after the execution of a RETURN or END statement.



LG200025_057c

Semantics

The following items must not be mentioned in a SAVE statement: formal argument names, procedure names, and names of variables in a common block.

A SAVE statement without a list of variable names or common block names declares that all allowable variables in the subprogram must be saved.

The total size of local variables in a single subroutine must be less than one gigabyte. (A gigabyte is 1,073,741,824 (2^{30}) bytes.)

When a common block name is specified, all of the variables in that common block are saved. Within an executable program, if a common block name is mentioned in a SAVE statement, it must be mentioned in a SAVE statement in each subprogram where it appears.

A SAVE statement is optional in a main program and has no effect.

SAVE statements cannot be used in procedures contained in executable libraries.

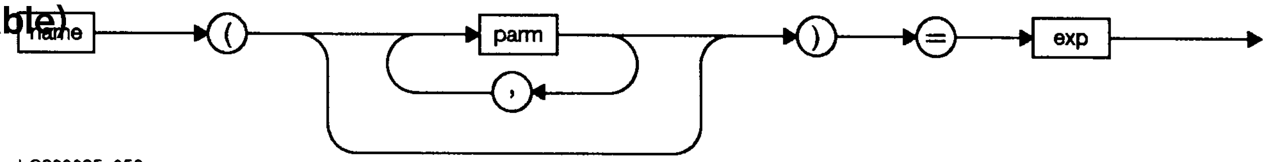
Examples	Notes

`SUBROUTINE matrix`
`:`
`SAVE a,b,c,/dot/`
`:`
`RETURN`
	The SAVE statement saves the values of `a`, `b`, and `c`, and the values of all of the variables in the common block `dot`.

`SUBROUTINE fixit`
`SAVE`
`:`
`RETURN`
	The SAVE statement saves the values of all of the variables in the subroutine `fixit`.

Statement Function Statement (Nonexecutable)

The statement function statement defines a one-statement function.



LG200025_058a

Item	Description/Default	Restrictions
name	Name of the function.	None.
parm	Formal argument.	Must be a simple variable.
exp	Arithmetic, logical, or character expression.	None.

A statement function is a program-defined, single-statement computation that applies only to the program unit in which it is defined. A statement function statement can appear only after the specification statements and before the first executable statement of the program unit.

The expression defines the actual computational procedure, which results in one value. When the statement function is referenced, the expression is evaluated using the actual arguments, and the value is assigned to the function name. The expression must be an arithmetic, logical, or character expression.

The type of a statement function is determined by using the statement function name in a type statement or by implicit typing. The type of expression in a statement function statement must be compatible with the defined type of the name of the function. For example, arithmetic expressions must be used in arithmetic statement functions, logical expressions in logical statement functions, and character expressions in character statement functions.

The arithmetic expression in an arithmetic statement function need not be the same type as the function name. For example, the expression can be type integer and the function name can be defined as type real. The expression value is converted to the statement function type at the time it is assigned to the function name.

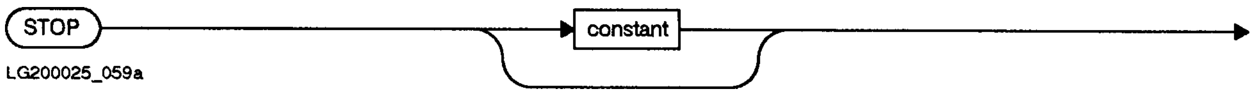
Statement functions can reference other previously defined statement functions. Statement functions cannot contain calls to themselves, nor can they contain indirect recursive calls.

The values of any formal arguments in the expression are supplied by the actual arguments when the statement function is referenced. All other variables and constants in the expression derive their values from their definitions in the program unit.

Examples	Notes
`disp(a,b,c)=a + b*c`	`disp` is a statement function with three dummy arguments, `a`, `b`, and `c`.
`tim(t1)=t1/2 + b`	`tim` is a statement function with one dummy argument, `t1`. `b` is an actual variable that is declared elsewhere in the program unit.

STOP Statement (Executable)

The STOP statement terminates program execution.



LG200025_059a

Item	Description/Default	Restrictions
constant	Integer or character constant to be displayed in the STOP message.	Cannot be a constant name or constant expression.

The STOP statement terminates program execution immediately without allowing execution to reach the END statement of the main program unit.

If *constant* is supplied, the message "`STOP` *constant_value*" is written to the standard error unit and the program terminates. If *constant* is omitted, the program terminates without a message.

Examples	Notes
`STOP 7777`	The message `STOP 7777` is written to standard error.
`STOP 'Program ended!'`	The message `STOP Program ended!` is written to standard error.
`STOP`	Nothing is written to standard error.

`READ *,a,b`
`IF (a .LT. b) STOP 56789`
`10 b = b-1`
`IF (b .EQ. a) STOP 'All done'`
`GOTO 10`
`END`
	If `a` is less than `b`, execution terminates with the message `STOP 56789`. When `b` equals `a`, execution terminates with the message `STOP All done`.

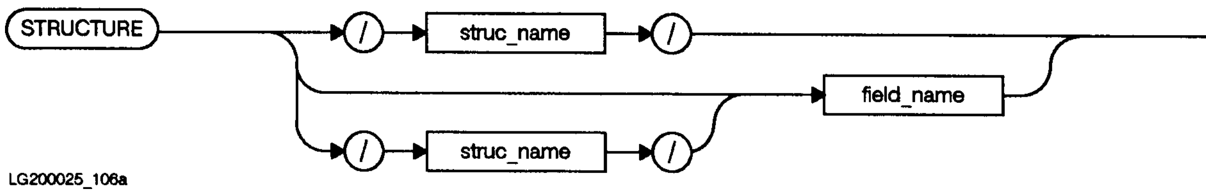
STRUCTURE Statement (Nonexecutable)

The STRUCTURE statement names and begins the declaration of a structure in a structure block. A structure is the "data type" of a record variable. It must be declared before a RECORD statement can refer to it. The END STRUCTURE statement terminates a structure block.

A structure block has the following elements:

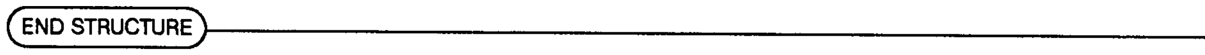
STRUCTURE statement	Begins the structure declaration.
Declaration body	Declares the fields of the structure, including their names, data types, order, and alignment. Fields are in the order of their declaration in the structure.
END STRUCTURE statement	Ends the structure declaration.

STRUCTURE Statement



LG200025_106a

END STRUCTURE Statement



LG200025_108

Item	Description/Default	Restrictions
struc_name	Name of the structure. Used in the RECORD statement to define the form of a record variable.	Required at the outermost level of the structure. Can also be used to name a substructure.
field_name	Field name for a substructure.	Required for a substructure. Not permitted at the outermost level of the structure.

Semantics

The declaration body consists of declaration statements that define symbolic names. The symbolic names are the field names of the structure. These field declarations can be any combination of the following:

Type specification statement	Any type specification statement defining variables or array declarators.
Substructure declaration	A substructure can be declared in two ways: 1. With a RECORD statement using a previously defined structure. 2. As a structure declaration block having a *field_name*. If a *struc_name* is also specified, the substructure can be referenced as a structure declaration by a subsequent RECORD statement. Recursive structure declarations are not permitted. A structure which is currently being defined can not refer to itself.
Union declaration	A union declaration block, described below, specifies two or more fields that share a common location within the structure.
Unnamed field	The special symbolic name, `%FILL`, can be used in a structure as the "name" of an empty field. `%FILL` is described below.

Structure names must be unique between structures. However, structure fields, variables, and common blocks can have the same name as a structure.

Field names must be unique at the same structure level. However, for example, a structure can contain a field named `field1` and a substructure of that structure can also contain a field named `field1`.

For convenience in declaring constants, PARAMETER statements may be placed between the statements in a structure block. However, it is important to note that a PARAMETER statement within a structure block has the same effect as if it were outside the block.

Note

A structure declaration does not allocate storage. Structure declarations only specify the form for a record.

Field Declarations

All fields declared within a structure must be explicitly typed. The IMPLICIT statement has no effect on field names within a structure declaration block.

A field that is an array must be specified as an array name with dimension declarator within the explicit type specification statement. The DIMENSION statement is not permitted within a structure. For example:

```
REAL*4 field2(10)
```

declares `field2` to be an array of 10 4-byte real values.

Dynamic, assumed size, or adjustable arrays can not be declared within a structure declaration. Also, character items with passed length or variable length can not be declared within a structure.

Unnamed Fields

Unnamed fields, substructures, and records can be declared in a structure by using the special name `%FILL` as a dummy field name. These unnamed fields can be used for alignment. They cannot be referenced or initialized. For example:

```
STRUCTURE /align/
   CHARACTER*3 shortname
   BYTE %FILL
   REAL*4 vector(10)
END STRUCTURE
```

Data Initialization

Fields within a structure can be given initial values, using the initialization rules for the type specification statements. Uninitialized fields are undefined until they are assigned values.

Unnamed fields cannot be initialized.

When a record is declared, its fields are initialized to the values specified in the structure declaration.

If more than one map block initializes the same area in a union block, the last initialization takes precedence.

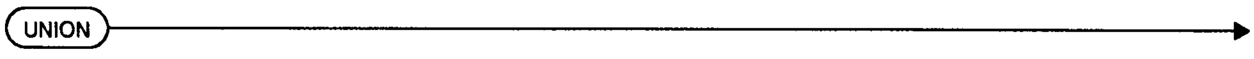
UNION Statement (Nonexecutable)

The UNION statement begins the declaration of a union block in a structure block. A union block defines a shared data section of a structure. The END UNION statement terminates a union block.

A union block has the following elements:

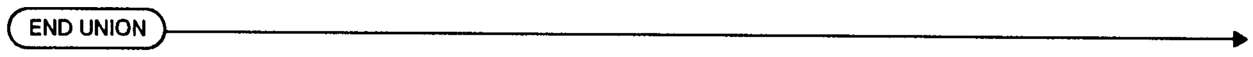
UNION statement	Begins the union declaration.
Declaration body	Two or more map declaration blocks.
END UNION statement	Ends the union declaration.

UNION Statement



LG200136_006

END UNION Statement



LG200025_109

A union declaration defines the form of a data location within a structure that is shared by different groups of data items during program execution. The data groups are defined with map declaration blocks in the union declaration. The map blocks share the same physical storage space. When one field of a map block is accessed then all the fields in that map are defined and the fields of the other maps in that union are undefined.

The overall size of a union declaration is the size of the largest map block within the union.

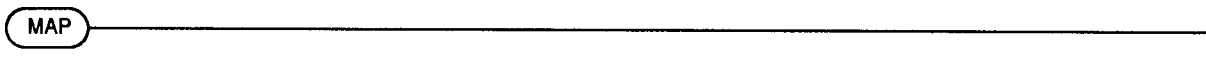
MAP Statement (Nonexecutable)

The MAP statement begins the declaration of a map block in a union declaration block. The END MAP statement terminates a map block.

A MAP block has the following elements:

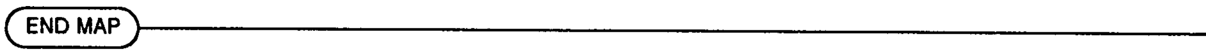
MAP statement	Begins the MAP declaration.
Declaration body	Declares the fields of the map block, including their names, data types, order, and alignment. Fields are in the order of their declaration.
END MAP statement	Ends the MAP declaration.

MAP Statement



LG200025_110

END MAP Statement



LG200025_111

A map declaration block specifies the form of the fields within a union declaration block. The rules for a map declaration body are the same as the rules for a structure declaration body.

Each map block within a union begins at the same data location in memory. Consequently, the initialization of fields within one map may affect and be affected by initializations within another map. The same data area may be initialized more than once. Only the last initialization of that area is valid. For example, in the following union block:

```
UNION
   MAP
      CHARACTER*8 a /"01234567"/
   END MAP
   MAP
      CHARACTER*4 b
      CHARACTER*4 c /"ABCD"/
   END MAP
END UNION
```

field `b` is initialized to `"0123"`, `c` to`"ABCD"`, and `a` to `"0123ABCD"`.

Examples	Notes

```
STRUCTURE /num1/
  INTEGER*4 i, j
END STRUCTURE
```

The structure **num1** defines two 4-byte integer fields, **i** and **j**, which occupy consecutive words in memory.

```
STRUCTURE /num2/
  UNION
    MAP
      INTEGER*4 i
      REAL*4 a
    END MAP
    MAP
      INTEGER*4 j
      LOGICAL*1 x(10)
    END MAP
  END UNION
END STRUCTURE

RECORD /num2/ overlay
```

The structure **num2** defines a union that contains two map blocks. The first map consists of a 4-byte integer, **i**, followed by a 4-byte real, **a**. The second map consists of a 4-byte integer, **j**, followed by an array, **x**, of 10 1-byte logical elements. Integers **i** and **j** occupy the same storage location. Real **a** occupies the same storage as the first four elements of array **x**. The entire union is 14 bytes long.

The record **overlay** has the structure of **num2**. It has the following fully qualified variable names:

- **overlay** - a record or aggregate
- **overlay.i** - a 4-byte integer
- **overlay.a** - a 4-byte real
- **overlay.j** - a 4-byte integer
- **overlay.x** - an array of 10 1-byte logicals

```
STRUCTURE /outer/
  STRUCTURE /inner/ self
   INTEGER*4 ssn
   INTEGER*2 age
   CHARACTER*18 name
  END STRUCTURE
  RECORD /inner/ spouse
  RECORD /num1/ data
END STRUCTURE

RECORD /outer/ personal

RECORD /inner/ someone
```

The structure **outer** contains a substructure declaration having both a *struc_name*, **inner**, and a *field_name*, **self**. The *struc_name* allows the substructure to be used as a structure form in the RECORD statement that defines the field **spouse** and in the separate RECORD statement that defines **someone**. The second RECORD statement in the structure refers to the structure **num1** defined above.

The record **personal** has the following fully qualified variable names:

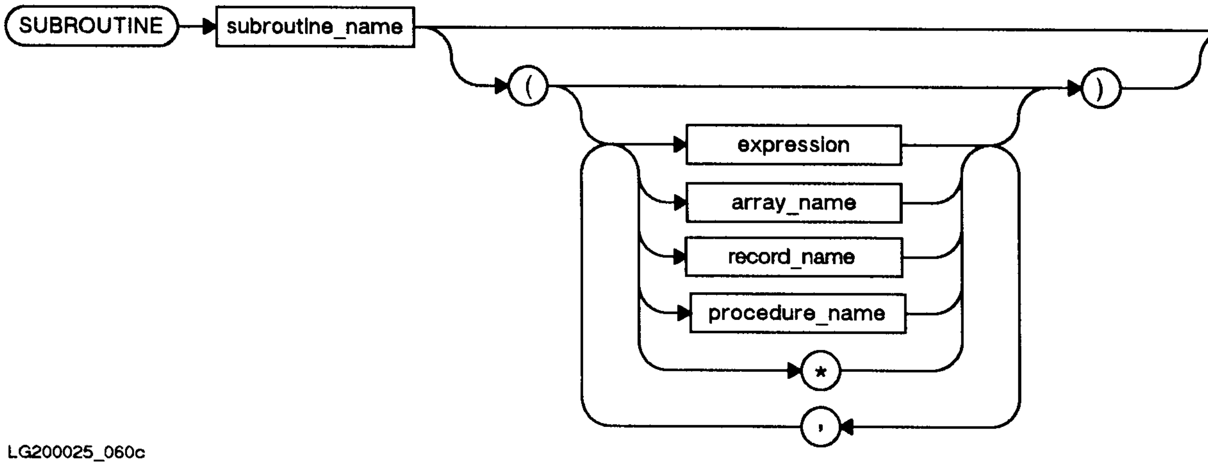
- **personal** - a record or aggregate
- **personal.self** - a record or aggregate
- **personal.self.ssn** - a 4-byte integer
- **personal.self.age** - a 2-byte integer
- **personal.self.name** - an 18-byte character
- **personal.spouse** - a record or aggregate
- **personal.spouse.ssn** - a 4-byte integer
- **personal.spouse.age** - a 2-byte integer
- **personal.spouse.name** - an 18-byte character
- **personal.data** - a record or aggregate
- **personal.data.i** - a 4-byte integer
- **personal.data.j** - a 4-byte integer

The record **someone** has the following fully qualified variable names:

- **someone** - a record or aggregate
- **someone.ssn** - a 4-byte integer
- **someone.age** - a 2-byte integer
- **someone.name** - an 18-byte character

SUBROUTINE Statement (Nonexecutable)

The SUBROUTINE statement identifies a program unit as a subroutine subprogram.



LG200025_060c

Item	Description/Default	Restrictions
*	Indicates an alternate return.	None.

Semantics

The formal arguments in a SUBROUTINE statement can be variables, array names, record names, or subprogram names. The formal arguments must be of the same type and structure as the actual arguments passed to the subroutine. In particular, the fields in actual and formal record arguments must agree in type, order, and dimension.

Asterisks in the SUBROUTINE statement can specify one or more alternate returns. Alternate returns are described in "RETURN Statement (Executable)".

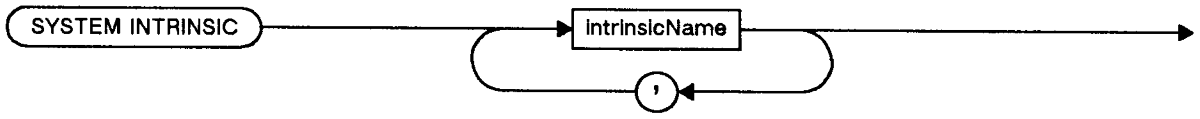
Examples	Notes
`SUBROUTINE add`	Begins a subroutine named `add` that has no formal arguments.
`SUBROUTINE sub(z,i,d,*,*,*)`	Begins a subroutine named `sub` with three arguments and three alternate return points.

SYSTEM INTRINSIC Statement (Nonexecutable)

The MPE/iX file SYSINTR.PUB.SYS contains information about the attributes of subprograms. These subprograms are usually user-callable system subprograms, such as FOPEN. All intrinsics mentioned in the MPE/iX manuals must be accessed through this facility. The information about a particular subprogram includes such items as the number and type of parameters, whether parameters are called by ANYVAR, READONLY, reference, UNCHECKABLE_ANYVAR, or value, and whether the subprogram parameters have the options DEFAULT_PARMS, EXTENSIBLE, or both. See the subsequent sections for an explanation of the preceding terms. Note that these terms relate to the way parameters are declared in HP Pascal/iX subprograms found in SYSINTR.PUB.SYS. FORTRAN reads the SYSINTR file for specially designated subprograms and generates the indicated code sequences.

You can designate that the SYSINTR file is to be searched for a particular subprogram by using the SYSTEM INTRINSIC statement.

Syntax



LG200023_045

Item	Description/Default	Restrictions
IntrinsicName	The name of the subprogram in the SYSINTR file.	If the name cannot be found in the SYSINTR file, an error message is issued.

This facility provides these advantages over the usual way of accessing external subprograms:

- Convenient access to routines written in any language is provided. For each such subprogram, the list of actual parameters does not have to be complete.

  Missing parameters (which must be specified as DEFAULT_PARMS in the intrinsic file) are indicated by commas or a right parenthesis. The occurrence of a right parenthesis before the formal parameter list is exhausted implies the rest of the parameters are missing (which means they are DEFAULT_PARMS or EXTENSIBLE, as specified in the intrinsic file).

- The VALUE, REFERENCE, ANYVAR, UNCHECKABLE_ANYVAR, or READONLY attribute of a formal parameter is recognized and the appropriate code for the actual parameter is automatically generated for the call. (An ALIAS compiler directive might otherwise be necessary to indicate how parameters should be passed or the routine might not even be callable because of the parameter type.) Parameter checking is performed at the highest level (level 3 of CHECK_ACTUAL_PARM) at compile time. * Automatic typing of SYSINTR file functions is provided. Thus, the intrinsic mechanism automatically types the function return type for you. For example, the statement

```
SYSTEM INTRINSIC FOPEN, BINARY
```

results in FOPEN being typed INTEGER*2 and BINARY being typed LOGICAL*2.

You can also specify that a Pascal intrinsic file other than SYSINTR.PUB.SYS be searched for the subprogram name. This can be done by using the SYSINTR compiler directive. See "SYSINTR Directive" in Chapter 7. For information on building Pascal intrinsic files, refer to the *HP Pascal Reference Manual*.

For more information about system intrinsics, see the SYSINTR Compiler Directive.

The SYSTEM INTRINSIC statement must appear before any executable statement in the program.

The SYSTEM INTRINSIC compiler directive functions exactly the same as the SYSTEM INTRINSIC statement, except that the directive has a global effect.

A Value Parameter

This is a parameter that is passed by value; that is, a copy of the value of the actual parameter is passed to a routine and assigned to the formal parameter of that routine. If the routine changes the value of the formal parameter, it does not change the value of the actual parameter. An actual value parameter can be a constant, an expression, a variable, or a function result.

The need for passing a FORTRAN 77/iX parameter by value arises when you have a system intrinsic that requires a certain parameter be passed by value. For example, the following program calls the system intrinsic HPCICOMMAND, which requires its fourth parameter (0) to be passed by value, and executes the MPE/iX system DATE command. For more information on HPCICOMMAND, read the *MPE/iX Intrinsic Reference Manual*.

Example

```
$STANDARD_LEVEL SYSTEM       ! Display no warnings
      PROGRAM excommand

      SYSTEM INTRINSIC HPCICOMMAND

      CHARACTER*10 command
      INTEGER*2 cmderror, parmnum
      command = "DATE"//char(13)
      CALL HPCICOMMAND(command, cmderror, parmnum, 0)
      END
```

The program `excommand` uses the system intrinsic HPCICOMMAND to execute the MPE/iX DATE command. The results from executing the program look similar to this:

```
THU, JAN 16, 1992,  8:41 AM
```

Note that the above example will be referenced in the sections "A Reference Parameter" and "The ANYVAR Parameter and UNCHECKABLE_ANYVAR Option."

A Reference Parameter

This is a parameter that is passed by reference; that is, the address of the actual parameter is passed to the routine and associated with the formal parameter. If the routine changes the value of the formal parameter, it changes the value of the actual parameter. An actual reference parameter must be a variable name.

In FORTRAN 77/iX, all variables are passed by reference except when interfacing with other programming languages and accessing FORTRAN 77/iX system intrinsics.

An example of passing parameters by reference can the seen in the section "A Value Parameter." In this example, the parameters `cmderror` and `parmnum` are passed by reference back to the calling program.

The ANYVAR Parameter and UNCHECKABLE_ANYVAR Option

When a parameter in the formal parameter list of an HP Pascal/iX procedure declaration is denoted ANYVAR, it means that a variable of any type can be passed to it as the actual parameter. The address of the actual parameter will be passed and the code in the procedure will access it as the type specified for the formal parameter in the procedure declaration. In this way, intrinsic procedures can be created with the HP Pascal/iX language that accept any type of variable as their actual parameters (assuming data alignment requirements are met).

For ANYVAR parameters, the length of the actual parameter must be passed as a "hidden parameter" in the parameter list. Hidden parameters are parameters that do not appear in formal or actual parameter lists, but are nevertheless passed to routines (they are always integers).

UNCHECKABLE_ANYVAR is an HP Pascal/iX procedure option that specifies that ANYVAR hidden parameters will not be created for a routine. This allows its parameter list to be compatible with the parameter list of a rountine written in a language other than HP Pascal/iX.

An example of an ANYVAR parameter being declared as uncheckable can be found in the section "A Value Parameter." In this example, the ANYVAR parameter `command` has been declared uncheckable by the HP Pascal/iX UNCHECKABLE_ANYVAR option and no hidden parameters are passed for the ANYVAR parameter.

An EXTENSIBLE Parameter

EXTENSIBLE is an HP Pascal/iX procedure option that identifies a procedure that has an *extensible parameter list*.

An *extensible parameter list* has a fixed number of nonextension parameters and a variable number of extension parameters. The integer *n* after the keyword EXTENSIBLE specifies that the first *n* parameters in the formal parameter list are *nonextension parameters* (*n* can be zero). Any other parameters are *extension parameters*.

A *nonextension parameter* is required. Every call to the routine must provide an actual parameter for it.

An *extension parameter* is optional. A call to the routine can omit its actual parameter from the actual parameter list. However, if the actual parameter list contains an actual parameter for the *x*th extension parameter, it must contain actual parameters for those before it.

The number of extension parameters in an extensible parameter list is flexible: you can add new ones later, and you need not recompile programs that call the routine.

Example

The program in this section uses the intrinsic HPFOPEN to open a file named `testfile` for reading and displays the file name, number and status. For more information on HPFOPEN, read the *MPE/iX Intrinsic Reference Manual*.

The MPE/iX intrinsic HPFOPEN is extensible and provides default parameters. The extensible parameters in HPFOPEN are: `status`, `itemnum1`, `item1`, `itemnum2`, and `item2`. These parameters can be omitted. HPFOPEN's nonextensible parameter is `filenum` and it cannot be omitted from the actual parameter list unless it has a default value assigned to it (see the section "A DEFAULT_PARMS Parameter"). Note that the parameters `itemnum1` and `item1` and `itemnum2` and `item2` must appear in pairs in the actual parameter list of the HPFOPEN intrinsic.

Compiling and executing the following program:

```
$STANDARD_LEVEL SYSTEM           ! Display no warnings
      PROGRAM openfile

      SYSTEM INTRINSIC HPFOPEN

      INTEGER*4 filenum, status, itemnum1, itemnum2, item2
      CHARACTER*20 item1
      PARAMETER( itemnum1 = 2, itemnum2 = 11 )

      item1 = 'testfile' ! The file name is "testfile"
```

```
item2 = 0          ! The file "testfile" has READ access

CALL HPFOPEN(filenum, status, itemnum1, item1, itemnum2, item2)

PRINT '(" File Name  : " A20)',item1
PRINT '(" File Number: " I6)',filenum
PRINT '(" File Status: " I6)',status
PRINT *,CHAR(13)
END
```

produces these results:

```
File Name  : testfile
File Number:      9
File Status:      0
```

Note that the example in this section is referenced in the section "A DEFAULT_PARMS Parameter."

A DEFAULT_PARMS Parameter

DEFAULT_PARMS is an HP Pascal/iX procedure option that specifies default values to be assigned to formal parameters when actual parameters are not passed to them. For example, the HPFOPEN intrinsic used in the program in the section "An EXTENSIBLE Parameter" could have the `status` variable omitted from the parameter list and it would still work. If you replaced the program line:

```
CALL HPFOPEN(filenum, status, itemnum1, item1, itemnum2, item2)
```

with this program line:

```
CALL HPFOPEN(filenum,, itemnum1, item1, itemnum2, item2)
```

you would get the following results:

```
File Name  : testfile
File Number:      9
File Status:      0
```

If a nonextension parameter has a default value, its actual parameter can be left out of the actual parameter list, and its default value is assigned to the formal parameter.

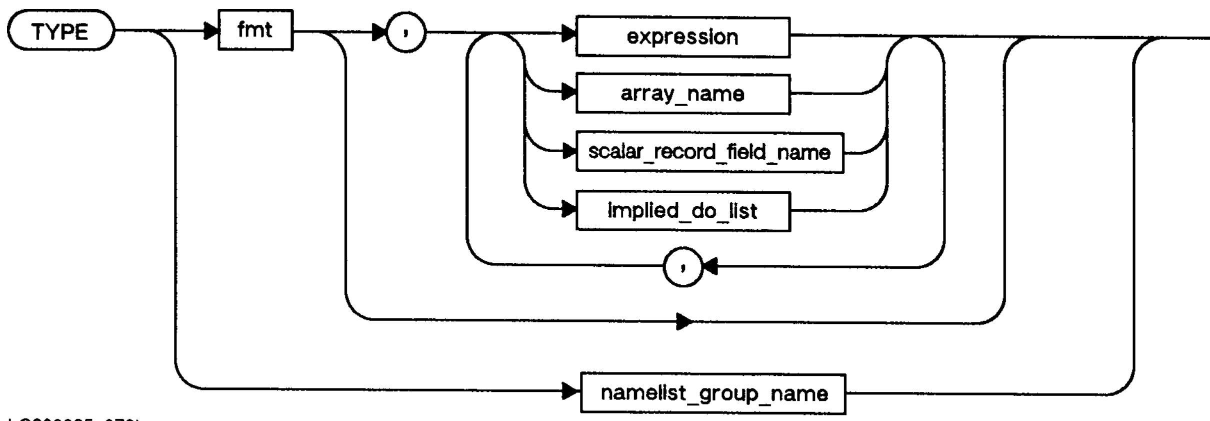
A default value must be a constant expression that is assignment compatible with its parameter. The value *nil* is the only legal default for reference, ANYVAR, function or procedure parameters.

A READONLY Parameter

READONLY is an HP Pascal/iX parameter that protects the actual parameter from modification within an MPE/iX intrinsic procedure that is called from a FORTRAN 77/iX program.

TYPE Statement (Executable)

The TYPE statement transfers data from memory to the standard output unit. (FORTRAN unit 6 is preconnected to the standard output unit.)



LG200025_076b

Semantics

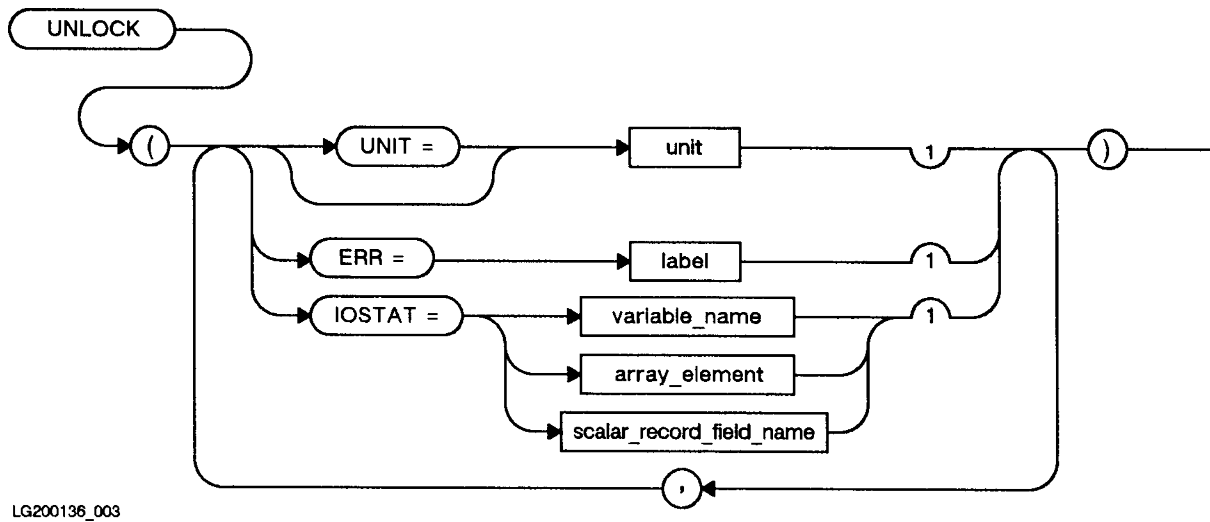
The TYPE statement, which is an extension to the ANSI 77 standard, is equivalent to the PRINT statement. See "PRINT Statement (Executable)" for further information.

UNION Statement (Nonexecutable)

The UNION statement begins a UNION statement block in a STRUCTURE statement block. For more information, refer to “STRUCTURE Statement (Nonexecutable)”.

UNLOCK Statement (Executable)

The UNLOCK statement unlocks a record that was locked by a READ on an ISAM file. If an ISAM file is shared by more than one process, you should unlock the records after each READ if they are not being updated. If the file is not shared, UNLOCK has no effect.



LG200136_003

Item	Description/Default	Restrictions
unit	Integer expression specifying unit number of a connected file.	Must be zero or positive.
variable_name *array_element* *scalar_record_field_name*	Error code return.	Must be an integer type.
label	Statement label.	Must be an executable statement in the same program unit.

Semantics

If the prefix `UNIT=` is omitted, *unit* must be the first item in the list.

If the ERR specifier is present and an error occurs during execution of the UNLOCK statement, control transfers to the specified statement rather than aborting the program.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for the IOSTAT error codes.

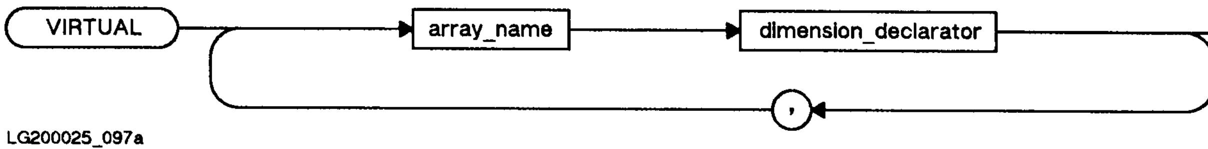
If the file is positioned at its beginning, a UNLOCK statement has no effect upon the file.

Examples	Notes
(code below)	This example generates a report. The READ statement locks the record when it reads it. The UNLOCK statement unlocks the record to allow other users to access it. If the record is locked by another user, the ERR specifier causes the READ statement to be reexecuted until the record is available.

```
      OPEN(10,file='EMP_FILE',
     +ACCESS='KEYED',STATUS='OLD',SHARED)
111 READ (10,ERR=111,END=222) EMPLOY_REC
      UNLOCK (10,ERR=112)
      PRINT 11,EMPLOYEE_REC.SSN,
     +       EMPLOYEE_REC.FIRST_NAME
     +       EMPLOYEE_REC.LAST_NAME,
     +       EMPLOYEE_REC.SALARY,
     +       EMPLOYEE_REC.REVIEW_DATE
      GOTO 111

112 PRINT *,'Can not unlock: ssn', EMPLOYEE_REC.SSN
     GOTO 111

222 PRINT *,'Employee salary review report printed'
     STOP
```

VIRTUAL Statement (Nonexecutable)

The VIRTUAL statement defines the dimensions and bounds of arrays.



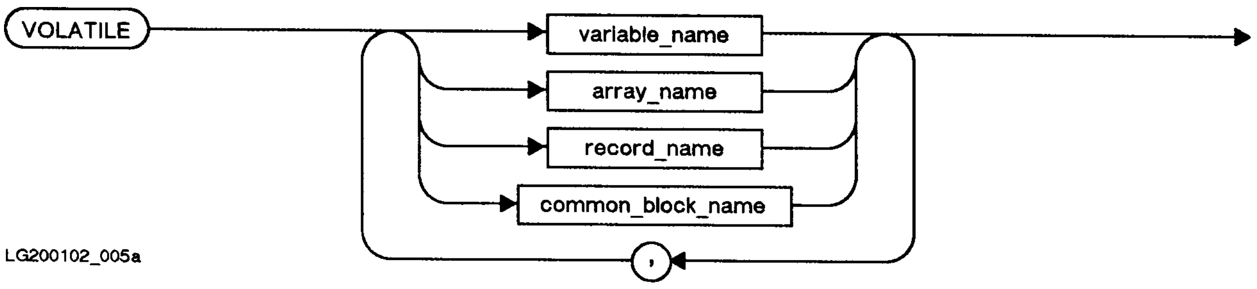
LG200025_097a

Item	Description/Default	Restrictions
array_name	Symbolic name of the array.	None.
dimension_declarator	Defines the number of dimensions and the range of each dimension.	Same as for the DIMENSION statement.

The VIRTUAL statement, which is an extension to the ANSI 77 standard, is equivalent to the DIMENSION statement. See "DIMENSION Statement (Nonexecutable)" for further information.

VOLATILE Statement (Nonexecutable)

The VOLATILE statement specifies variables, arrays, and common blocks that will not be selected for global analysis and optimization by the compiler.



Item	Description/Default	Restrictions
variable_name *array_name* *record_name* *common_block_name*	Variable, array, record, or common block that is not being optimized.	None.

Semantics

The VOLATILE statement is an extension to the ANSI 77 standard.

If an array variable name is specified, all of the elements in that array become volatile. Likewise, if a common block variable is specified, the entire common block becomes volatile. A variable that is overlapping (through EQUIVALENCE) on a volatile variable also becomes volatile.

Even though variables, arrays, records, and common block names listed in a VOLATILE statement are allocated storage, the VOLATILE statement cannot be used for declaring the size of an array or common block. When referenced, volatile variables are loaded from memory. When a volatile variable is changed, the variable is stored into memory (that is, the variable is not held in the registers).

Variables can be placed in shared memory with the SHARED_COMMON compiler directive so that they can be accessed by more than one program. Any variable placed in shared common is considered volatile. If you declare a common block as shared common, the compiler implicitly attributes volatile to that block. That is, you do *not* have to separately declare a common block as volatile and then compile and install that common block as shared.

The VOLATILE statement can also be used when variables, arrays, or common blocks are shared between the program and the exception handler. If you do not use the VOLATILE statement, the exception handler might use the incorrect value for a shared quantity because the value was kept in the registers and not updated in memory.

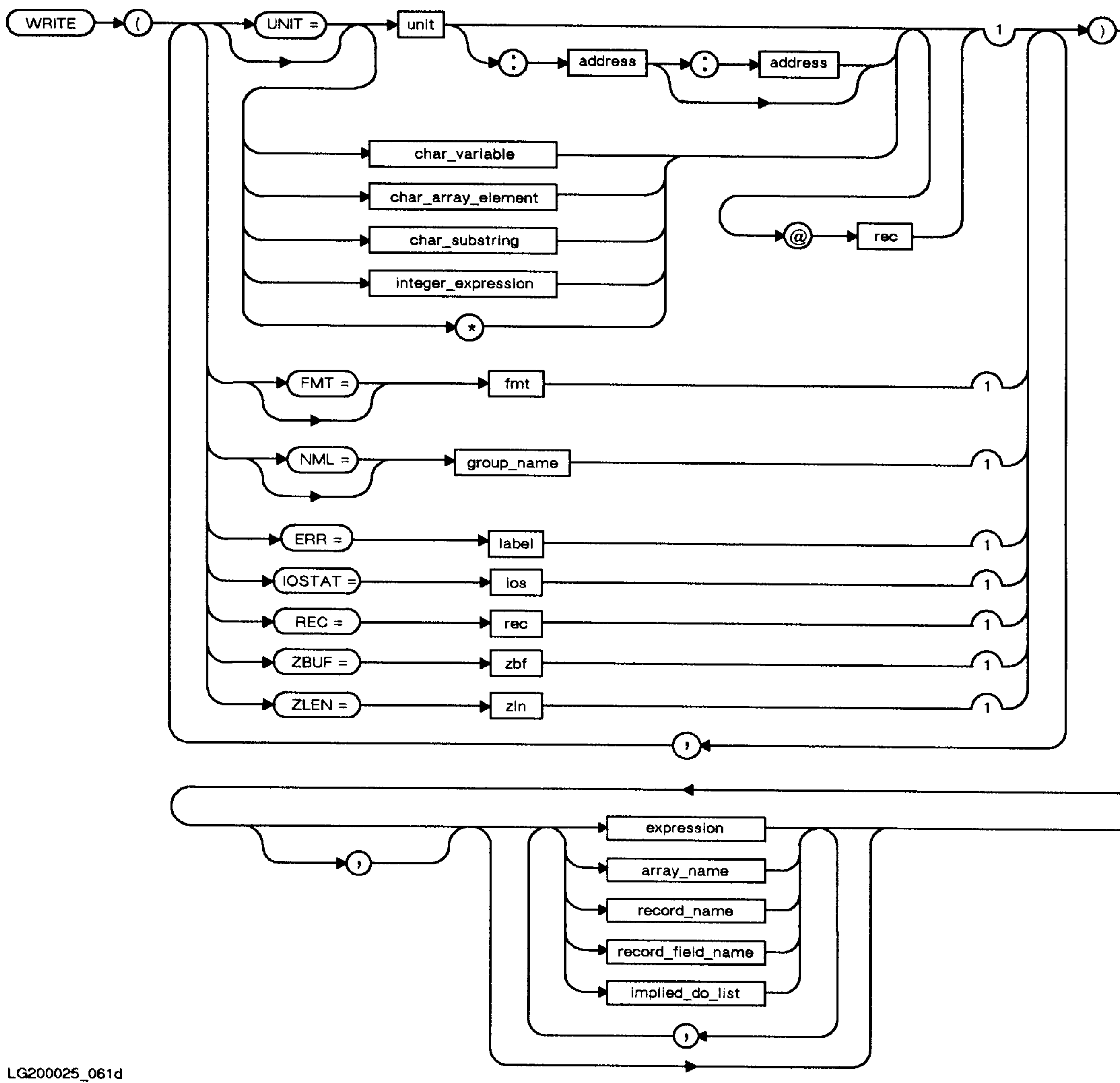
Example	Notes

`INTEGER a, b, c, d`
`INTEGER p, q, r`
`COMMON /block1/ a, b, c`
`VOLATILE /block1/, q, r`
	The common block `block1` and variables `q` and `r` are volatile and will not be selected for global analysis or optimization. An equivalent VOLATILE statement is
	`VOLATILE q, r, /block1/`

`INTEGER a, b`
`EQUIVALENCE (a,b)`
`VOLATILE a`
	The variable `b` also becomes volatile.

WRITE Statement (Executable)

The WRITE statement transfers data from memory to a file or device.



LG200025_061d

Item	Description/Default	Restrictions
unit	Integer expression specifying the unit number of a file.	Must be zero or positive.
address	Arithmetic expression of type integer.	See "Semantics".
char_variable	Internal file written.	Character variable or scalar record field.
char_array_element	Internal file written.	Character array element or scalar record field.
char_substring	Internal file written.	Character substring.
integer_expression	Integer expression specifying the unit number of an internal file.	Must be zero or positive.
*	Asterisk indicates that the standard output device (unit 6, usually a terminal) is to be used.	None.
fmt	Format designator.	*fmt* must be as specified in the PRINT statement.
namelist_group_name	Symbolic name specifying a list of variables or arrays previously declared in a NAMELIST statement.	Cannot appear in a statement containing a format specifier.
ios	Integer variable, array element, or scalar record field name for error return.	Must be an integer type.
label	Statement label of an executable statement.	Must be the label of a statement in the same program unit.
rec	Specifies the record number in a direct access file.	If *fmt* is an asterisk, a record specifier must not be present.
zbf	Variable, array name, or array element name.	Extension to ANSI 77 standard; cannot be a character type.
zln	Integer expression.	None.

Semantics

A WRITE statement must contain a unit number and at most one of each of the other options.

If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. This is the unit number for the output device or file.

If the prefix `FMT=` is omitted, *fmt* must be the second item in the list and *unit* (without a prefix) must be the first item. If *fmt* is not present and a namelist group name is not specified, the write is unformatted (binary).

record_name and *aggregate* variables can only be used in unformatted writes.

If the IOSTAT specifier is present and an error occurs, the error code is returned in the IOSTAT variable and the program is not aborted. Refer to Appendix A for the IOSTAT error codes.

You must specify a record number if a file is direct access. The record number can be specified through REC. You can also specify the record number by following the unit number with the `@` specifier.

If the ERR specifier is present and an error occurs during execution of the WRITE statement, control transfers to the specified statement rather than aborting the program.

The ZBUF and ZLEN specifiers and the *address* alternative used as a parameter of the UNIT specifier are included for compatibility with programs originally written in another version of FORTRAN. If one is used in a program, its syntax is checked but it is otherwise ignored by the compiler.

As an extension to the ANSI 77 standard, sequential WRITE operations (without the REC specifier) are allowed on files open for direct access. If the REC specifier is omitted, a WRITE statement writes the next record.

Refer to Chapter 4 for more information on the WRITE statement.

Examples	Notes
`WRITE (7,10) a,b,c`	The values of `a`, `b`, and `c` are written to the file connected to unit 7 according to FORMAT statement 10.

`ASSIGN 4 TO num`
`WRITE (UNIT=3,IOSTAT=j,`
`+ERR=5,FMT=num) z`
	The value of `z` is written to the file connected to unit 3, according to FORMAT statement 4. If an error occurs, control transfers to statement 5 and the error code is returned in `j`.

`WRITE (10) (x + y)`	The value of the expression `(x + y)` is written to the file connected to unit 10. Because *fmt* is omitted, the data is unformatted.
`WRITE (10,FMT=*) b`	The value of `b` is written to the file connected to unit 10 according to list-directed formatting.
`WRITE (2,'(I3)',REC=10) i`	The value of `i` is written to the 10th record of the direct file connected to unit 2 according to the format specification in the WRITE statement itself.
`WRITE (*)`	One record is skipped on the standard output device.

4

Input/Output

Input/output (I/O) statements allow you to enter data into a program and to transfer data between a program and a disk file, terminal, or other device. There are four types of input/output:

- Formatted input/output.
- Unformatted input/output.
- List-directed input/output.
- Namelist-directed input/output.

For each type of input/output, there are one or more input statements and corresponding output statements.

In the examples in this chapter, Δ represents a blank.

Formatted Input/Output

Formatted input/output allows you to control the use of each character of a data record. This control is specified in a FORMAT statement or in the input/output statement itself.

Formatted Input

Formatted input is specified by the following input statements. Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "READ Statement (Executable)" in Chapter 3.

Syntax

`READ` *fmt* , *list*

`READ (` *unit* , *fmt* `)` *list*

Item	Description/Default	Restrictions
fmt	Format designator.	See "Semantics".
unit	Unit number of the file.	None.
list	List of variables that specifies where the data is to be transferred.	See "Semantics".

Semantics

The format designator, *fmt*, must be one of the following:

- The statement label of a FORMAT statement.
- A variable name that has been assigned the statement label of a FORMAT statement.
- A character expression.
- A character array name that contains the representation of a format specification.
- An asterisk, indicating list-directed formatting.

list can contain implied DO loops.

For information on implied DO loops, refer to the section "DO Statement (Executable)" in Chapter 3.

The first READ statement syntax shown above transfers information from the standard input device (preconnected to unit 5). The second READ statement transfers data from a file or device.

Reading always starts at the beginning of a record. Reading stops when the list is satisfied, provided that the format specification and the record length are in agreement with the list. If the list is omitted, the file pointer is positioned at the next record without data transfer. If the format specification is longer than the list, reading stops at the first repeatable edit descriptor, colon, or right parenthesis terminating the format after the list is satisfied. If the list is longer than the format specification, reading skips to the next record, which is read using part or all of the format specification again. However, if a carriage return is entered but no value, the element is set to zero (0). This process continues until the list is satisfied. If the list is longer than the record and the format specification does not cause a skip to the next record, error 910 (Access past end of record attempted) occurs. The record is treated as if blanks were added to the end to satisfy the input list.

After the READ, the file pointer is positioned at the beginning of the next record. Each READ statement begins reading values from a fresh record of the file; any values left unread in records accessed by previous READ statements are ignored. For example, if the record contains six data elements, a READ statement such as

```
READ (4,100) i,j
```

reads only the first two elements. The remaining four elements are not read because any subsequent READ statement reads values from the next record, unless the file pointer is repositioned (with BACKSPACE or REWIND, for example) prior to the next READ.

You can use a comma (,) to terminate the input of noncharacter data or to assign the value of zero to selected values. The next field starts immediately after the comma.

You cannot use the comma to terminate the input field or to assign values when using the `A`, `R`, `M`, or `N` format descriptors, because the comma has a meaning for these descriptors.

For example, if the program

```
PROGRAM comma
INTEGER a, b, c, d, e
a = 1
b = 2
c = 3
d = 4
e = 5
READ(5,'(I2, 3I2, I2)') a, b, c, d, e
PRINT *, a, b, c, d, e
END
```

has this input (remember, `Δ` represents a space):

```
Δ9Δ8,
```

the following is printed:

```
Δ9Δ8Δ0Δ0Δ0
```

The program terminates reading values after the comma (that is, after reading the value of `b`).

Similarly, if the input is

```
Δ7Δ4,Δ3Δ2
```

the following is printed:

```
 7 4 0 3 2
```

The value zero is assigned to `c`.

Finally, if the input is

```
12,14,16
```

the following is printed:

```
Δ12Δ0Δ14Δ0Δ16
```

An array name in a list represents all the elements in the array. Values are transferred to the array elements in accordance with the standard array storage order. Refer to "Variables" in Chapter 2 for details.

Formatted Output

Formatted output is specified by the following output statements. Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "PRINT Statement (Executable)" in Chapter 3 and "WRITE Statement (Executable)" in Chapter 3.

Syntax

`PRINT` *fmt* , *list*

`WRITE (` *unit* , *fmt* `)` [,] *list*

Item	Description/Default	Restrictions
fmt	Format designator.	See "Semantics".
unit	Unit number of the file.	None.
list	List of variables or expressions that specifies the data to be transferred.	See "Semantics".

Semantics

The format designator, *fmt*, must be one of the following:

- The statement label of a FORMAT statement.
- A variable name that has been assigned the statement label of a FORMAT statement by an ASSIGN statement.
- A character expression.
- A character array name that contains the representation of a format specification.
- An asterisk, indicating list-directed formatting.

If *list* is omitted, a blank line is written (unless a literal appears in the format before a repeatable format descriptor). *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "DO Statement (Executable)" in Chapter 3.

The PRINT statement transfers information to the standard output device (preconnected to unit 6). The WRITE statement transfers information to disk files or to output devices.

The optional comma (,) in the WRITE syntax is an extension to the ANSI 77 standard.

Each WRITE statement begins writing values into a fresh record of the destination file; any space left unused in records accessed by previous WRITE statements is ignored. After the transfer is completed, the file record pointer is advanced.

Writing always starts at the beginning of a record. Writing stops when the list is satisfied, provided that the format specification and the record length agree with the list. If the format specification is longer than the list, writing stops at the first repeatable edit descriptor, colon, or right parenthesis terminating the format after the list is satisfied. If the list is longer than the format specification, writing skips to the next record, which is written using part or all of the format specification again.

This process continues until the list is satisfied. If the list is longer than the output record and the format specification does not cause a skip to the next record, the record is truncated at the record length of the file. After the write process, the file pointer is positioned at the beginning of the next record.

Array names in the list represent all the elements of the array. Array element values are transferred according to the standard array storage order. See "Variables" in Chapter 2.

Note

Record names and aggregates are not permitted in formatted I/O.

Carriage Control

The first character of each output record is always considered to be a carriage control character for devices that recognize carriage control. The standard carriage control characters are listed in Table 4-1.

Table 4-1. Carriage Control Characters

Character	Vertical Spacing Before Printing
`Δ` (blank)	One line (single spacing)
`0`	Two lines (double spacing)
`1`	To first line of next page (page eject)
`+`	No advance (overprinting)
Any other character	Device dependent

Format Specifications

A format specification is a list of format descriptors and edit descriptors, enclosed in parentheses. It is equivalent to a FORMAT statement without the *label* and `FORMAT` keyword.

Format Specification Syntax

(*descriptor_list*)

Descriptor List Syntax

$$\left\{\begin{array}{l} [\,\textit{repeat\_spec}\,]\textit{format\_descriptor} \\ [\,\textit{repeat\_spec}\,](\ \textit{descriptor\_list}\ ) \\ \textit{edit\_descriptor} \end{array}\right\}[\,[\,,\,]\ldots]$$

Item	Description/Default	Restrictions
edit_descriptor	An edit descriptor, as described in Table 4-3.	None.
format_descriptor	A format descriptor, as described in Table 4-2.	None.

repeat_spec must be an unsigned positive integer constant or a variable format descriptor whose value is positive.

The format descriptors describe how the data appears and the edit descriptors specify editing information. For example, in the following format specification:

```
(I3,3X,3F12.3)
```

the format descriptor `I3` specifies an integer number with a field width of three (the integer takes up a total of three character positions), the edit descriptor `3X` specifies that three character positions are to be skipped, and the format descriptor `3F12.3` specifies three real numbers, each with a field width of 12 and three significant digits to the right of the decimal point. A PRINT statement using this format specification could be of the form:

```
PRINT '(I3,3X,3F12.3)',i,a,b,c
```

The relationships are shown in Figure 4-1.

Format Specifiers	`I3`	`3X`	`F12.3`	`F12.3`	`F12.3`
Output Variables	`i`		`a`	`b`	`c`
Output Record	`345`	`ΔΔΔ`	`65376453.324`	`ΔΔΔ14321.265`	`Δ4765321.321`
Field Widths	3	3	12	12	12

Figure 4-1. Output Formatting

Format specifications can be supplied in FORMAT statements or written as character expressions in input/output statements.

Format Specifications in Format Statements

A format specification can be placed in a FORMAT statement that is referenced by a corresponding READ, WRITE, or PRINT statement. For the complete syntax definition, refer to "FORMAT Statement (Nonexecutable)" in Chapter 3.

FORMAT Statement Syntax

label `FORMAT` *format_specification*

Item	Description/Default	Restrictions
label	Required statement label.	None.
format_specification	Format specification, defined above.	None.

Examples	Notes

`READ (5,10) a,i,d,e`
`10 FORMAT(A2,I3,D8.2,F12.2)`
	The format descriptors `A2`, `I3`, `D8.2`, and `F12.2` in the FORMAT statement correspond with the list variables `a`, `i`, `d`, and `e` in the READ statement. List element `a` corresponds with the format descriptor `A2`, `i` with `I3`, `d` with `D8.2`, and `e` with `F12.2`.

A FORMAT statement can be used by several input/output statements. Make sure that each variable in the input/output list corresponds with its respective format descriptor in the format specification.

Format Specifications in Input/Output Statements

The format specification can be contained in the input/output statement as a character expression.

Examples	Notes
`READ (4,'(A3,3X,F10.2)') a,z`	The variables `a` and `z` are read according to the format specification `(A3,3X,F10.2)`.

`CHARACTER*6 a`
`DATA a /'(3I3) '/`
`PRINT a,i,j,k`
`WRITE (6,'(F10.2)') d`
	The three integers `i`, `j`, and `k` are printed according to the format specification `(3I3)`. The variable `d` is written as a fixed-point number according to the format specification `(F10.2)`.

Care must be taken when the format specification in an input/output statement contains an apostrophe or a quotation mark. Whenever an apostrophe (') appears within an apostrophe edit descriptor, it must be represented by two consecutive apostrophes. Similarly, if a quotation mark (") appears in a quotation mark edit descriptor, it must be represented by two consecutive quotation marks.

Each of these, in turn, must be represented by two apostrophes or quotation marks if the format is a character literal contained in an input/output statement. Notice, therefore, that four consecutive apostrophes are required in the second example below.

Examples	Notes
`WRITE (6,'(3X,''THIS IS THE END'')')`	Writes the following record: `ΔΔΔTHIS IS THE END`
`WRITE (6,'(''Isn''''t it true!'')')`	Writes the following record: `Isn't it true!`
`WRITE (6,'("Isn''t it true!")')`	Writes the same record as the previous example.

Format Descriptors

The descriptors in a format specification must be separated by a comma except before and after a slash (`/`) edit descriptor, a colon (`:`) edit descriptor, or between a scaling (`P`) edit descriptor and an immediately following `D`, `E`, `F`, or `G` edit descriptor. For example, if a slash descriptor is used to indicate a new line of output or a new record on input, the comma that would separate the descriptors is not necessary. The specification:

```
3I,F4.0,/I5,F12.6
```

is equivalent to:

```
3I,F4.0/I5,F12.6
```

Format descriptors can be preceded by a repeat specification (for example, the `3` in `3F12.4`).

Format and edit descriptors can include another set of format or edit descriptors, or both, enclosed in parentheses; this is called nesting. The nested format specification can be preceded by a repeat specification.

For example, the information shown on the input record in Figure 4-2 could be accessed with a FORMAT statement like the following:

```
10 FORMAT (I3,F7.4,3(F7.2,I3),F12.4)
```

A READ statement using the FORMAT statement could be:

```
READ 10,i,a,b,j,d,k,e,m,f
```

Format Specifiers	I3	F7.4	F7.2	I3	F7.2	I3	F7.2	I3	F12.4
Input Variables	i	a	b	j	d	k	e	m	f
Input Record	Δ26	26.4336	Δ342.26	Δ24	2373.86	439	Δ649.79	ΔΔ4	ΔΔΔ4395.4972
Field Widths	3	7	7	3	7	3	7	3	12

Figure 4-2. Input Formatting

The READ statement would read values for `i` and `a`, then repeat the nested format specification `(F7.2,I3)` three times to read values for `b` and `j`, for `d` and `k`, and for `e` and `m`, and, finally, read a value for `f`.

The format descriptors are summarized in Table 4-2; the edit descriptors, in Table 4-3. A detailed explanation of the descriptors follows the tables.

Table 4-2. Format Descriptors

Data Type	Format Descriptor	Additional Explanation
INTEGER*2, INTEGER*4	`I`[w[.m]]	None
REAL*4, REAL*8, REAL*16 COMPLEX*8, COMPLEX*16	`F`[w.d]	Fixed-point format descriptor
REAL*4, REAL*8, REAL*16 COMPLEX*8, COMPLEX*16	`D`[w.d] `E`[w.d[`E`e]]	Floating-point format descriptor
REAL*4, REAL*8, REAL*16 COMPLEX*8, COMPLEX*16	`G`[w.d[`E`e]]	Fixed- or floating-point format descriptor
On input: INTEGER*2, INTEGER*4 On output: any type	`@`[w[.m]] `K`[w[.m]] `O`[w[.m]]	Octal
On input: INTEGER*2, INTEGER*4 On output: any type	`Z`[w[.m]]	Hexadecimal
LOGICAL*1, LOGICAL*2, LOGICAL*4	`L`[w]	None
REAL*4, REAL*8, REAL*16	`M`[w.d]	Monetary format; fixed-point
REAL*4, REAL*8, REAL*16	`N`[w.d]	Numeration format; fixed-point
Character, Hollerith, or any other type (treated as character)	`A`[w]	Character or Hollerith data is left-justified in memory and external format
Character, Hollerith, or any other type (treated as character)	`R`[w]	Character or Hollerith data is right-justified in memory and external format

Table 4-3. Edit Descriptors

Edit Descriptor	Function
`BN`	Ignore blanks.
`BZ`	Treat blanks as zeros.
n`H`*string*	Hollerith literal editing
`"`*string*`"`	Literal editing.
`'`*string*`'`	Literal editing.
`NL`	Restore newline.
`NN`	No newline.
`$`	No newline; same as `NN`.
k`P`	Scale factor.
`Q`	Number of characters remaining in current input record.
`SP`	Output optional plus signs.
`SS`	Inhibit optional plus sign output.
`S`	Processor determines sign output; same as `SS`.
`T`*c*	Skip to column *c*.
`TL`*t*	Skip *t* positions to the left.
`TR`*t*	Skip *t* positions to the right.
x`X`	Skip *x* positions to the right.
`/`	Begin new record.
`:`	Terminate format if I/O list is empty.

Numeric Format Descriptors

The numeric format descriptors specify the input/output fields of integer, real, and complex data types. The following rules apply to all numeric format descriptors:

- The field width, *w*, specifies the total number of characters that a data field occupies, including any leading plus or minus sign, decimal point, or exponent.
- On input, leading blanks are not significant. Trailing and embedded blanks are treated as null characters unless the `BZ` edit descriptor is encountered or the unit was connected with `BLANK='ZERO'` specified. A field of all blanks is considered to be a 0.
- On output, the data item is right-justified in the field. If the item length is less than the field width, leading blanks are inserted in the field. If the item is longer than the field width for certain descriptors, the entire field is filled with dollar signs, as specified in the output examples of the particular descriptors.
- A complex list item is treated as two real items, and a double complex list item as two double precision items.
- If a numeric list item is used with a numeric descriptor of a different type, the value is converted as needed.

Integer Format Descriptor (I)

The I[*w*[.*m*]] format descriptor defines a field for an integer number. The corresponding input/output list item must be a numeric type. The optional *m* value specifies a minimum number of digits to be output. If *m* is not shown, a default value of one is assumed. The *m* value is ignored on input.

On input, the I*w* format descriptor causes the interpretation of the next *w* positions of the input record; the number is converted to match the type of the list element currently using the descriptor. A plus sign is optional for positive values. A decimal point must not appear in the field.

Input Examples

Descriptor	Input Field	Value Stored
I4	Δ1ΔΔ	1
I5	ΔΔΔΔΔ	0
I2	-1	−1
I4	-123	−123
I3	Δ12	12
I2	123	12

On output, the I*w* or I*w*.*m* format descriptor causes output of a numeric variable as a right-justified integer value (rounding takes place if necessary). The field width, *w*, should be one greater than the expected number of digits, to allow for a minus sign for negative values. If $m = 0$, a 0 value is output as all blanks.

Output Examples

Descriptor	Internal Value	Output Field
I4	+452.25	Δ452
I2	+6234	**
I3	−11.92	-12
I5	−52	ΔΔ−52
I3	−124	***
I10	123456.5	ΔΔΔ123457
I6.3	3	ΔΔΔ003
I3.0	0	ΔΔΔ
I3.3	−123	***

Real Format Descriptors (D, E, F, G)

The D[*w*.*d*], E[*w*.*d*[E*e*]], F[*w*.*d*], and G[*w*.*d*[E*e*]] format descriptors define fields for real or complex numbers. (Note that two descriptors must be specified for a complex value.) The input/output list item corresponding to a D, E, F, or G descriptor must be a numeric type.

On input, all of these format descriptors work identically.

The input field for these descriptors consists of an optional plus or minus sign followed by a string of digits that may contain a decimal point. If the decimal point is omitted in the input string, the number of digits equal to *d* from the right of the string are interpreted to be to the right of the decimal point. If a decimal point appears in the input string and conflicts with the format descriptor, the decimal point in the input string takes precedence. This basic form can be followed by an exponent in one of the following forms:

- A signed integer constant.
- An E followed by an integer constant.
- A D followed by an integer constant.

All three exponent forms are processed the same way. The following are examples of valid input fields:

Input Examples

Descriptor	Input Field	Value Stored
F6.5	4.51E4	45100.0
G4.2	51-3	.00051
E8.3	7.1ΔEΔ5	710000.
D9.4	ΔΔΔ45E+35	$.0045 \times 10^{35}$
BZ,F7.1	54E34Δ	-5.4 x 10^{340} (overflow error)
F12.10	34	3.4×10^{-9}

The BZ edit descriptor equates blanks to zeros. It is described in "Blank Interpretation Edit Descriptors (BN, BZ)".

The appearance of the output field depends on whether the format descriptor specifies a fixed- or floating-point format.

Floating-Point Format Descriptors (D, E)

The D[*w* . *d*] and E[*w* . *d*[E*e*]] format descriptors define a floating-point field on output for real and complex values. The value is rounded to *d* digits. The exponent part consists of *e* digits. If E*e* is omitted, the exponent occupies two positions. The field width, *w*, should follow the general rule:

w >= d + 7

or, if E*e* is used,

w >= d + e + 5

to provide positions for a leading blank, the sign of the value, the decimal point, *d* digits, the letter E, the sign of the exponent, and the exponent.

Output Examples

Descriptor	Internal Value	Output Field
`D10.3`	+12.342	`ΔΔ.123E+02`
`E10.3E3`	−12.3454	`-.123E+002`
`E12.4`	+12.340	`ΔΔΔ.1234E+02`
`D12.4`	−.00456532	`ΔΔ-.4565E−02`
`D10.10`	99.99913	`**********`
`E11.5`	+999.997	`Δ.10000E+04`
`E10.3E4`	$.624 \times 10^{-30}$	`.624E-0030`

Fixed-Point Format Descriptor (F)

The `F`[*w*`.`*d*] format descriptor defines a fixed-point field on output for real and complex values. The value is rounded to *d* digits to the right of the decimal point. The field width, *w*, should be four greater than the expected length of the number, to provide positions for a leading blank, the sign, the decimal point, and a roll-over digit for rounding if needed.

Output Examples

Descriptor	Internal Value	Output Field
`F5.2`	+10.567	`10.57`
`F3.1`	−254.2	`***`
`F6.3`	+5.66791432	`Δ5.668`
`F8.2`	+999.997	`Δ1000.00`
`F8.2`	−999.998	`-1000.00`
`F7.2`	−999.997	`*******`
`F4.1`	23	`23.0`

Fixed- or Floating-Point Format Descriptor (G)

The `G`[*w*`.`*d*[`E`*e*]] format descriptor defines a fixed- or floating-point field, as needed, on output for real and complex values. The `G` format descriptor is interpreted as an `F`*w*`.`*d* descriptor for fixed-point form or as an `E`*w*`.`*d*[`E`*e*] descriptor for floating-point form according to the magnitude of the data. If the magnitude is less than 0.1 or greater than or equal to 10***d* (after rounding to *d* digits), the `E`*w*`.`*d*`[E`*e*`]` format descriptor is used; otherwise the `F`*w*`.`*d* format descriptor is used. When `F`*w*`.`*d* is used, trailing blanks are included in the field where the exponent would have been. The field occupies *w* positions; the fractional part consists of *d* digits, and the exponent part consists of *e* digits. If `E`*e* is omitted, the exponent occupies two positions. The field width, *w*, should follow the general rule for floating-point descriptors:

w `>=` *d* `+ 7`

or, if `E`*e* is used,

w `>=` *d* `+` *e* `+ 5`

to provide for a leading blank, the sign of the value, *d* digits, the decimal point, and, if needed, the letter `E`, the sign of the exponent, and the exponent.

Descriptor	Internal Value	Interpreted As	Output Field
`G10.3`	+1234	E10.3	`ΔΔ.123E+04`
`G10.3`	−1234	E10.3	`Δ-.123E+04`
`G12.4`	+12345	E12.4	`ΔΔΔ.1235E+05`
`G12.4`	+9999	F8.0,4X	`ΔΔΔ9999.ΔΔΔΔ`
`G12.4`	−999	F8.1,4X	`ΔΔ-999.0ΔΔΔΔ`
`G7.1`	+.09	E7.1	`Δ.9E-01`
`G5.1`	−.09	E5.1	`*****`
`G12.1`	+9999	E12.2	`ΔΔΔΔΔ.1E+05`
`G8.2`	+999	E8.2	`Δ.10E+04`
`G7.2`	−999	E7.2	`*******`
`G8.2`	0	E8.2	`Δ00E+00`

Character Format Descriptors (A, R)

The A[*w*] and R[*w*] format descriptors define fields for character and noncharacter data. The size of the list variable (byte length) determines the maximum effective value for *w*. If *w* is not specified, the size of the field is equal to the size of the input/output variable.

As an extension to the ANSI 77 standard, any data type can be used with the A and R descriptors.

Using the A and R format descriptors for input and output, *w* can be equal to, less than, or greater than the specified byte size of the input or output variable. If *w* is equal to the length of the variable, the character data field is the same as the variable. If *w* is less than or greater than the length of the variable, there are eight possibilities for the character data field, summarized in Table 4-4.

The R format descriptor is an extension to the ANSI 77 standard.

Table 4-4. Contents of Character Data Fields

Input Descriptor	**Length of Input Variable**	**Result**
A[*w*]	$w < len$	Left-justified in variable, followed by blanks.
	$w > len$	Taken from right part of field.
R[*w*]	$w < len$	Right-justified in variable, preceded by binary zeros.
	$w > len$	Taken from right part of field.
Output Descriptor	**Length of Output Variable**	**Result**
A[*w*]	$w < len$	Taken from left part of variable.
	$w > len$	Right-justified in variable, preceded by blanks.
R[*w*]	$w < len$	Taken from right part of variable.
	$w > len$	Right-justified in variable, preceded by blanks.

In the following examples, Δ or Δ represents a blank and ⊔ represents a byte of binary zeros.

Input Examples

Descriptor	Input Field	Internal Length	Value Stored
A3	XYZ	3	XYZ
R3	XYZ	4	⊔XYZ
A5	ABCΔΔ	10	ABCΔΔΔΔΔΔΔ
R9	RIGHTMOST	4	MOST
A5	CHAIR	5	CHAIR
R8	CHAIRΔΔΔ	8	CHAIRΔΔΔ
A4	ABCD	2	CD

Output Examples

Descriptor	Internal Value	Internal Length	Output Field
A6	ABCDEF	6	ABCDEF
R4	ABCDEFGH	4	EFGH
A4	ABCDE	5	ABCD
A8	STATUS	6	STATUSΔΔ
A4	NEXT	4	NEXT
R8	STATUS	6	ΔΔSTATUS

Numeric Data with Character Format Descriptors

The `A` and `R` character format descriptors can be used with integer and real data types.

If you specify the NOSTANDARD compiler directive, the data is output in reverse order, starting at the right and progressing to the left.

Example

```
      PROGRAM demo
!  Output numeric data with character format using an external write.
      INTEGER*4 i4
      i4 = 4Habcd
      WRITE(6,100) i4
100   FORMAT(a)
      STOP
      END
```

If you specify the NOSTANDARD compiler directive, `dcba` is output. If it is not specified, `abcd` is output.

For more information refer to “NOSTANDARD Directive” in Chapter 7.

Logical Format Descriptor (L)

The `L`[*w*] format descriptor defines a field for logical data. The input/output list item corresponding to a `L` descriptor must be a logical type.

On input, the field width is scanned for optional blanks followed by an optional decimal point, followed by a `T` for true or an `F` for false. The first nonblank character in the input field (excluding the optional decimal point) determines the value to be stored in the declared logical variable. If the first nonblank character is not a `T` or an `F`, an error occurs.

Input Examples

Descriptor	Input Field	Value Stored
L5	ΔΔΔTΔ	.TRUE.
L2	F1	.FALSE.
L4	ΔxΔT	Error
L5	ΔRTΔ	Error
L7	TFALSEΔ	.TRUE.
L7	.FALSE.	.FALSE.

On output, a `T` or an `F` is right-justified in the output field depending on whether the value of the list item is true or false.

Output Examples

Descriptor	Internal Value	Output Fields
L5	.FALSE.	ΔΔΔΔF
L4	.TRUE.	ΔΔΔT
L1	.TRUE.	T
L2	.FALSE.	ΔF

The logical value true or false is determined by the least significant bit of the most significant byte of the internal representation.

Octal Format Descriptors (@, K, O)

As an extension to the ANSI 77 standard, the `@`[*w*[.*m*]], `K`[*w*[.*m*]], and `O`[*w*[.*m*]] format descriptors define a field for octal data. These descriptors provide conversion between an external octal number and its internal representation. On output, list elements may be of any type, though character variables are output only as the octal equivalent of their ASCII representation (no length descriptor). Variables to receive octal input must be integer types.

On input, the presence of too many digits for the integer variable (or list element) to receive produces undefined results. Legal octal digits are 0 through 7. Plus and minus signs are not permitted on input nor printed on output. If any nonoctal digit appears, an error occurs.

On output, if *w* is greater than the number of converted octal digits (including blanks between words but excluding leading zeros), the octal digits are right-justified in the output field. Blanks are inserted at the boundaries of machine words in internal representation. If *w* is less than the number of converted octal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output without a sign, their high-order bits are nonzero and cause the field to be filled with asterisks if *w* is less than the number of octal digits in the entire output value.

The optional *m* value specifies a minimum number of digits to be output, forcing leading zeros as necessary up to the first non-zero digit. The *m* is ignored on input.

Input Examples

Descriptor	Input Field (Octal)	Value Stored (Octal)
`@6`	`123456`	123456
`2K4`	`00360005`	000036 and 000005
`O5`	`12934`	Error
`@6`	`17777B`	Error
`K9`	`-37000000`	Error

Output Examples

Descriptor	Internal Value (Decimal)	Output Field (Octal)
`K6`	99	`ΔΔΔ143`
`O2`	99	`**`
`@8`	−1 (INTEGER*4)	`********`
`@6`	32767	`Δ77777`

Hexadecimal Format Descriptor (Z)

As an extension to the ANSI 77 standard, the `Z`[*w*[.*m*]] format descriptor defines a field for hexadecimal data. This descriptor provides conversion between an external hexadecimal number and its internal representation. On output, list elements may be of any type, though character variables are output only as the hexadecimal equivalent of their ASCII representation (without a length descriptor). Variables to receive hexadecimal input must be an integer type.

On input, the presence of too many digits for the integer variable (or list element) to receive produces undefined results. Legal hexadecimal digits are 0 through 9 and A through F. Plus and minus signs are not permitted on input, nor printed on output. If any nonhexadecimal digit appears, an error occurs.

On output, if *w* is greater than the number of converted hexadecimal digits (including blanks between words but excluding leading zeros), the hexadecimal digits are right-justified in the output field. Blanks are inserted at the boundaries of machine words in internal representation. If *w* is less than the number of converted hexadecimal digits, the field is filled with asterisks. This primarily affects the output of negative values. Because negative values are output without a sign, their high-order bits are nonzero and cause the field to be filled with asterisks if *w* is less than the number of hexadecimal digits in the entire output value.

The optional *m* value specifies a minimum number of digits to be output, forcing leading zeros as necessary up to the first non-zero digit. The *m* is ignored on input.

Input Examples

Descriptor	Input Field (Hexadecimal)	Value Stored (Hexadecimal)
`Z6`	`123ABC`	123ABC
`2Z4`	`009F0005`	00009F and 000005
`Z5`	`12G34`	Error
`Z6`	`Z111111`	Error
`Z9`	`37000000`	Error

Output Examples

Descriptor	Internal Value (Decimal)	Output Field (Hexadecimal)
Z6	99	ΔΔΔΔ63
Z2	299	**
Z7	−1 (INTEGER*4)	*******
Z6	32767	ΔΔ7FFF

Variable Format Descriptor (<expression>)

Variable format descriptors, having the form <*expression*>, allow the values of integer variables, integer constant names, and character constants to be used in format specifications. Integer variable format descriptors may be used wherever an integer may appear, except they cannot be used to specify the number of characters in a Hollerith descriptor. Character-constant variable format descriptors may be used anywhere in the format specification between the opening and closing parentheses. The following is an example of a variable format descriptor:

```
I<isize>
```

In this example, the `I` format descriptor performs an integer data transfer with a field width equal to the value of `isize` when the format is used.

Variables may be INTEGER*2 or INTEGER*4. The value of a variable format descriptor must be of a valid magnitude for its use in the format; otherwise an error occurs.

Variable format descriptors are only permitted in FORMAT statements and in constant format specifications used in I/O statements. They are not allowed in run-time formats (that is, those that are prepared in arrays or character expressions at run-time).

If a variable is used, its value is reevaluated each time it is encountered in the normal format scan. If the value of a variable used in a descriptor changes during execution of the I/O statement, the new value is used the next time the format item containing the descriptor is processed.

Example

The following program illustrates the use of variable format descriptors:

```
      PROGRAM varfmt1
      INTEGER n
      PARAMETER (n = 3)
      REAL x(n,n)

C  Data for two-dimensional array x(n,n):
      DATA x / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /

C  Print out the constants 1 through 3 in variable width fields.
      DO 10 j = 1,3
        PRINT 100,j
  100   FORMAT (1x, I<j>)
   10 CONTINUE

C  Print out the lower diagonal elements of matrix x.
      DO 20 i = 1,n
        PRINT 101, (x(i,k), k = 1,I)
  101   FORMAT (1x, <I>F5.1)
```

```
 20 CONTINUE
    END
```

Output (`Δ` is a blank):

```
Δ1                  Format specification (1x,I<j>) became (1x,I1)
ΔΔ2                 Format specification (1x,I<j>) became (1x,I2)
ΔΔΔ3                Format specification (1x,I<j>) became (1x,I3)
ΔΔΔ1.0              Format specification (1x,<I>F5.1) became
                    (1x,1F5.1)
ΔΔΔ2.0ΔΔ5.0         Format specification (1x,<I>F5.1) became
                    (1x,2F5.1)
ΔΔΔ3.0ΔΔ6.0ΔΔ9.0    Format specification (1x,<I>F5.1) became
                    (1x,3F5.1)
```

Monetary Format Descriptor (M)

The M[*w*.*d*] format descriptor defines a field for a real number without an exponent (fixed-point) written in monetary form.

On output, the `M` format descriptor causes output of a numeric value in ASCII character fixed-point form, right-justified with commas and a dollar sign. The least significant digit (position *d*) is rounded. If needed, a leading minus sign is printed before the dollar sign.

In addition to the number of numeric digits, the field width *w* must allow for the number of commas expected plus four characters to hold the sign, the dollar sign, the decimal point, and a rollover digit (if necessary). If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required, the output value of the entire field is filled with asterisks.

On input, the `M` format descriptor causes interpretation of the next *w* positions of the input record as a real number without an exponent. The field width is expected (but not required) to have a dollar sign and commas embedded in the data as described for `M` output (the dollar sign and commas are ignored). If commas are used, the usage must be consistent; that is, commas must occur every three digits of the nonfractional part of the input value. The number is converted to an internal representation value for the variable (list element) currently using the format descriptor.

Input Examples

Descriptor	Input Field	Value Stored
`M10.3`	`ΔΔΔ$12.340`	12.340
`M10.3`	`ΔΔ$12.3402`	12.3402
`M13.3`	`ΔΔΔΔ80175.397`	80175.397
`M12.2`	`-$80,175.397`	−80175.397
`M12.2`	`ΔΔΔ99999.996`	99999.996

Output Examples

Descriptor	Internal Value	Output Field
`M10.3`	+12.3402	`ΔΔΔ$12.340`
`M10.3`	−12.3404	`ΔΔ-$12.340`
`M13.3`	+80175.3965	`ΔΔ$80,175.397`
`M12.2`	+99999.996	`Δ$100,000.00`
`M12.2`	−99999.996	`-$100,000.00`
`M11.2`	−99999.995	`***********`

Numeration Format Descriptor (N)

The N[*w*.*d*] field descriptor defines a field for a real number without an exponent (fixed-point) written in numeration form (that is, with commas, which are then ignored, in the input field).

On output, the N field descriptor causes output of a numeric value in ASCII character fixed-point form, right-justified with commas. The least significant digit is rounded. If needed, a leading minus sign is printed before the most significant digit.

In addition to the number of numeric digits, the field width *w* must allow for the number of commas expected, plus three characters to hold the sign, the decimal point, and a rollover digit (if necessary). If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the output value, the entire field is filled with asterisks.

On input, the N field descriptor causes interpretation of the next *w* positions of the input record as a real number without an exponent. The field width is expected (but not required) to have commas embedded in the data as described for N output (the commas are ignored). If commas are used, the usage must be consistent; that is, commas must occur every three digits of the dollar part of the input value. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

Input Examples

Descriptor	Input Field	Value Stored
N10.3	ΔΔΔΔ12.340	12.340
N10.3	ΔΔΔ12.3402	12.3402
N13.3	ΔΔΔΔ80175.397	80175.397
N12.2	ΔΔΔ80175.397	80175.397
N12.2	ΔΔ99,999.996	99999.996
N13.2	ΔΔ−10,000.317	−10000.317

Output Examples

Descriptor	Internal Value	Output Field
N10.3	+12.3402	ΔΔΔΔ12.340
N10.3	−12.3404	ΔΔΔ−12.340
N13.3	+80175.3965	ΔΔΔ80,175.397
N12.2	−80175.396	ΔΔ−80,175.40
N12.2	+99999.996	ΔΔ100,000.00
N10.2	−99999.995	**********

Edit Descriptors

Edit descriptors specify editing between numeric, Hollerith, and logical fields on input and output records. There are 19 edit descriptors.

- `BN`, `BZ`, and `Q` apply only to input.
- `NL`, `NN`, `$`, `S`, `SP`, and `SS` apply only to output.
- `H`, `"`*string*`"`, `'`*string*`'`, `P`, `T`, `TL`, `TR`, `X`, `/`, and `:` apply to both input and output.

Blank Interpretation Edit Descriptors (BN, BZ)

The `BN` and `BZ` edit descriptors interpret embedded and trailing blanks in numeric input fields. At the beginning of input statement execution, blank characters are ignored. (An exception to this rule occurs when the unit is connected with `BLANK='ZERO'` specified in the OPEN statement.) Note that `BN` and `BZ` override the BLANK specifier for the current READ statement. If a `BZ` edit descriptor is encountered in the format specification, trailing and embedded blanks in succeeding numeric fields are treated as zeros. The `BZ` edit descriptor remains in effect until a `BN` edit descriptor or the end of the format specification is encountered. If `BN` is specified or defaulted, all embedded blanks are removed and the input number is right-justified within the field width. The `BN` and `BZ` edit descriptors affect the `D`, `E`, `F`, `G`, `I`, `@`, `K`, `O`, and `Z` format descriptors during the execution of an input statement. They have no effect during execution of an output statement. They have no effect at any time on the `A`, `L`, `M`, `N`, and `R` format descriptors.

Input Examples

Descriptor	Input Field	BN Editing	BZ Editing
`I4`	`1Δ2Δ`	12	1020
`F6.2`	`Δ4Δ.Δ2`	4.2	40.02
`E7.1`	`5Δ.ΔE1Δ`	$5.\times10^{1}$	50.0×10^{10}
`I2`	`ΔΔ`	0	0
`E5.0`	`3E4ΔΔ`	$3.\times10^{4}$	$3.\times10^{400}$ (overflow)

End-of-Line Edit Descriptors (NL, NN, $)

As extensions to the ANSI 77 standard, FORTRAN 77 has edit descriptors to control the action taken at the end of an output line or record. These actions include controlling the cursor position on a terminal after a write and performing multiple writes on the same output line.

The `NL`, `NN`, and `$` end-of-line edit descriptors (also known as prompt edit descriptors) control the carriage-return/line-feed (newline) characters normally appended to a sequential output record.

The `NN` and `$` edit descriptors suppress the move to the next line before the write operation to the terminal, giving the appearance of appending the result of the write operation (containing the `NN` or `$` descriptor) to the current line. These descriptors perform multiple write operations to one apparent line of a carriage control output device.

The NL edit descriptor causes a newline character to be appended after the write operation containing the NL descriptor. The NL edit descriptor advances to the next line after each write containing the NL descriptor. NN (equivalent to $) is the default in HP FORTRAN 77/iX.

These descriptors apply to sequential output only. They are ignored by READ statements and direct access WRITE statements.

The `$` edit descriptor is the same as `NN` and is included for compatibility with other versions of FORTRAN. The prompt edit descriptors are an extension to the ANSI 77 standard.

The following program shows how to use the `NL` and `NN` descriptors to produce 50-column records. Unless they are used together, the program does not produce the desired effect, which, in this case, is to produce 10 fields per line. The FORMAT statement labeled 200 produces the desired effect, while statements 300, 400, and 500 do not. The program is followed by its output.

Example

```
          PROGRAM test
          n=23
          WRITE(6,100) (i, i=1,7)
          WRITE(6,200) (i, i=1,n)
          WRITE(6,*)
          WRITE(6,*)
          WRITE(6,100) (i, i=1,7)
          WRITE(6,300) (i, i=1,n)
          WRITE(6,*)
          WRITE(6,*)
          WRITE(6,100) (i, i=1,7)
          WRITE(6,400) (i, i=1,n)
          WRITE(6,*)
          WRITE(6,*)
          WRITE(6,100) (i, i=1,7)
          WRITE(6,500) (i, i=1,n)
      100 FORMAT (7I5)
      200 FORMAT (NN, 3I5, :, /, NL, (10I5))
      300 FORMAT (3I5, :, /, NL, (10I5))
      400 FORMAT (3I5, :, /, (10I5))
      500 FORMAT (NN, 3I5, :, /, (10I5))
          END
```

Output

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


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


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


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

Plus Sign Edit Descriptors (S, SP, SS)

The S, SP, and SS edit descriptors control printing of optional plus signs in numeric output. A formatted output statement does not normally print optional plus signs. However, an SP edit descriptor in the format descriptor forces optional plus signs to print in any subsequent numeric output. The S and SS descriptors inhibit printing of optional plus signs in subsequent numeric output.

Literal Edit Descriptors ('string', "string", H)

The literal edit descriptors are used to write a character constant to the output record or to skip columns in an input record. The '*string*' (apostrophe) and "*string*" (quotation mark) descriptors have the form of a character constant. The length of an apostrophe or quotation mark descriptor is the number of characters between the delimiters. As with character constants, two consecutive apostrophes in an apostrophe descriptor or two consecutive quotation marks in a quotation mark descriptor count as one character. The quotation mark edit descriptor is an extension to the ANSI 77 standard.

The *n*H*string* edit descriptor has the form of a Hollerith constant; *n* is the number of characters in *string*, including any blanks. The length of an H descriptor is *n*.

On output, the character constant is written.

On input, a literal edit descriptor behaves like a right tab (TR) of the same length as *string*. The input characters are skipped. The use of literal edit descriptors on input is an extension to the ANSI 77 standard.

Output Examples

Descriptor	Field Width	Output Field
'BEGIN DATA INPUT'	16	BEGIN DATA INPUT
"DAVID'S TURN"	12	DAVID'S TURN
"THE ENDΔΔΔ"	10	THE ENDΔΔΔ
'ΔΔSPACEΔΔ'	10	ΔΔSPACEΔΔ
"$*[/\%<;,#!"	11	$*[/\%<;,#!
'Aren''t'	5	Aren't
""""	1	"
'"'	1	"
7H$ΔHelp!	7	$ΔHelp!

Input Bytes Remaining Edit Descriptor (Q)

As an extension to the ANSI 77 standard, the `Q` edit descriptor returns the number of bytes remaining on the current input record. The value is returned to the next item in the input list, which must be an integer variable. This is useful when the exact contents of the input record are to be read, avoiding the blank padding of further variables read.

This descriptor applies to input only. It is ignored in output statements.

Position Edit Descriptor (X)

The *x*`X` edit descriptor skips *x* positions of an input/output record. *x* must be a positive integer.

The `X` descriptor is identical to the `TR` descriptor.

On input, the `X` edit descriptor causes the next *x* positions of the input record to be skipped.

Input Examples

Descriptors	Input Record	Values Stored
`F6.2,3X,I2`	`673Δ21END45`	673.21, 45
`1X,I2,A3`	`$6ΔEND`	6, END

On output, the `X` edit descriptor causes *x* positions of the output record to be filled with blanks. If the positions were already defined, they are left unchanged. This can happen when the `T` or `TL` edit descriptor is used.

Output Examples

Descriptors	Internal Values	Output Fields
`F8.2,2X,I3`	5.87,436	`ΔΔΔΔ5.87ΔΔ436`
`F4.2,3X,"TOTAL"`	32.4	`32.4ΔΔΔTOTAL`

Tab Edit Descriptors (T, TL, TR)

The tab edit descriptors position the cursor on the input or output record. The `T`*c* edit descriptor specifies an absolute column number (*c*), while the `TL`*t* and `TR`*t* descriptors specify a number of column positions to skip the left (`TL`) or right (`TR`) of the current cursor position. The `TR` descriptor is identical to the `X` descriptor. If the `T` or `TL` descriptor causes subsequent format descriptors to overwrite previous fields, the last character written for a particular column position in the output record is the character output for that position. On input, characters can be reread, possibly under a different editing format.

Input Example

Descriptors	Input Record	Stored Values
`A4,T1,F4.0`	`1234`	'1234', 1234.0

Output Examples

Descriptors	Internal Values	Output Record
`T5,F3.1`	1.0	`ΔΔΔΔ1.0`
`F3.1,TR4,F3.2`	1.0, .11	`1.0ΔΔΔΔ.11`
`T10,F3.1,TL12,F3.2`	1.0, .11	`.11ΔΔΔΔΔΔ1.0`

Record Terminator Edit Descriptor (/)

The / edit descriptor terminates processing of the current record and begins processing of a new record (such as a new line on a line printer or a terminal). The / edit descriptor has the same result for both input and output: it terminates the current record and begins a new one. For example, on output, a new line is printed; on input, a new line is read.

Colon Edit Descriptor (:)

If there are no more items in the input/output list, the colon edit descriptor (`:`) terminates format control (just as if the final right parenthesis in the format specification had been reached). If more items remain in the list, the colon edit descriptor has no effect.

Output Examples

Descriptors	Internal Values	Output Record
`(10('value=',I2))`	1, 2	`value= 1 value= 2 value=`
`(10(:,'value=',I2))`	1, 2	`value= 1 value= 2`

In the first example, the descriptor `'value='` is repeated an extra time because format control is not terminated until the descriptor I2 is reached and not satisfied. In the second example, the : descriptor terminates format control before the string `value=` can be output a third time.

Scale Factor Edit Descriptor (P)

The scale factor edit descriptor, *k*P (*k* is the scale value), is a descriptor that modifies the output of the D[*w*.*d*], E[*w*.*d*], and G[*w*.*d*] (interpreted as E[*w*.*d*]) format descriptors and the fixed-point output of the F[*w*.*d*] format descriptor. The scale factor also modifies the fixed-point inputs to the D[*w*.*d*], E[*w*.*d*], F[*w*.*d*], and G[*w*.*d*] format descriptors. A scale factor has no effect on the output of the G[*w*.*d*] (interpreted as F[*w*.*d*]) descriptor or on floating-point input.

For example, if a number of data items are stored without decimal points but are supposed to be interpreted as containing an implied decimal point two positions from the right, using a scale factor of −2 causes the items to be printed with the decimal point. Thus, with the format descriptor F7.2, the value 123 is printed 123.00, and with the format descriptor -2PF7.2, it is printed 1.23.

When a format specification is interpreted, the scale factor is set to 0. Each time a scale factor descriptor is encountered in a format specification, a new value is set. This scale value remains in effect for all subsequent affected format descriptors or until use of the format specification ends.

Examples	Notes
(E10.3,F12.4,I9)	No scale factor change. The default value, zero, remains in effect.
(E10.3,2PF12.4,I9)	Scale factor is zero for E10.3, changes to 2 for F12.4, has no effect on I9.

On input, the scale factor affects fixed-field (no exponent) input to the D[*w*.*d*], E[*w*.*d*], F[*w*.*d*], and G[*w*.*d*] format descriptors. The external value is multiplied by 10 raised to the $(-k)$th power, as illustrated below.

Input Examples

Descriptors	Input Field	Value Stored
E10.4	ΔΔ123.9678	123.9678
2PD10.4	ΔΔ123.9678	1.239678
−2PG11.5	ΔΔ123.96785	12396.785
−2PE12.5	123967.85E02	123967.85E02 (Note)

Note: If the input includes an exponent, the scale factor has no effect.

On output, the scale factor affects D[*w*.*d*], E[*w*.*d*], F[*w*.*d*], and G[*w*.*d*] (interpreted as E[*w*.*d*]) format descriptors only. The scale factor has no effect on the G[*w*.*d*] (interpreted as F[*w*.*d*]) field descriptor.

For `E`[*w*`.`*d*], `D`[*w*`.`*d*], and `G`[*w*`.`*d*] (interpreted as `E`[*w*`.`*d*]) format descriptors, the scale factor has the effect of shifting the decimal point of the output number to the right *k* places while reducing the exponent by *k* (the value of the mantissa remains the same). The number of significant digits printed is equal to ($d + k$).

Output Examples for E, D, and G

Descriptors	Stored Value	Output Field
`E12.4`	12.345678	`ΔΔΔ.1235E+02`
`3PE12.4`	12.345678	`ΔΔΔ123.5E−01`
`-3PD12.4`	12.345678	`ΔΔΔ.0001D+05`
`1PG11.3`	1234	`ΔΔ1.234E+03`

For the `F`[*w*`.`*d*] format descriptor, the internal value is multiplied by 10 raised to the ($+k$)th power, as illustrated below.

Output Examples for F

Descriptors	Input Field	Value Stored
`F11.3`	`1234.500`	1234.500
`-2PF11.3`	`1234.500678`	12.345
`2PF11.3`	`1234.500678`	123450.068

The scale factor need not immediately precede its format descriptor. For example, the format specification:

```
(3P,I2,F4.1,E5.2)
```

is equivalent to:

```
(I2,3P,F4.1,E5.2)
```

If the scale factor does not immediately precede a `D`[*w*`.`*d*], `E`[*w*`.`*d*], `F`[*w*`.`*d*], or `G`[*w*`.`*d*] format descriptor, it should be separated from other descriptors by commas or slashes. If the scale factor immediately precedes a `D`, `E`, `F`, or `G` format descriptor, the comma or slash descriptor is optional.

For example, the format specification:

```
(I2,3PF4.1,E5.2)
```

is equivalent to:

```
(I2,3P,F4.1,E5.2)
```

The scale factor affects all `D`, `E`, `F`, and `G` descriptors until either the end of the format specification or another scale factor is encountered.

Repeat Specification

The repeat specification is a positive integer written to the left of the format descriptor it controls. If a scale factor is needed also, it is written to the left of the repeat specification.

The repeat specification allows one format descriptor to be used for several list elements. It can also be used for nested format specifications; thus edit descriptors can be repeated by enclosing them in parentheses as shown above.

Examples	Notes
`(3F10.5)`	Equivalent to `(F10.5,F10.5,F10.5)`
`(2I3,2(3X,A5))`	Equivalent to `(I3,I3,3X,A5,3X,A5)`
`(L2,2(F2.0,2PE4.1),I5)`	Equivalent to `(L2,F2.0,2PE4.1,F2.0,E4.1,I5)`
`(2P3G10.4)`	Equivalent to `(2PG10.4,G10.4,G10.4)`

Nesting of Format Specifications

The group of format and edit descriptors in a format specification can include one or more other groups enclosed in parentheses (called groups at nested level *n*). Each group at nested level 1 can include one or more other groups at nested level 2; those at level 2 can include groups at nested level 3, and so forth.

Examples	Notes
`(E9.3,I6,(2X,I4))`	One group at nested level 1.
`(L2,A3/(E10.3,2(A2,L4)))`	One group at nested level 1 and one at level 2.
`(A,(3X,(I2,(A3)),I3),A)`	One group at nested level 1, one at level 2, and one at level 3.

Processing a Format Specification

A formatted input/output statement references each element in a series of list elements, and the corresponding format specification is scanned to find a format descriptor for each list element. As long as a list element and field descriptor pair occurs, normal execution continues.

If a program does not provide a one-to-one match between list elements and format descriptors, execution continues only until a format descriptor, an outer right parentheses, or a colon is encountered and there are no list items left. If there are fewer format descriptors than list elements, the following three steps are performed:

1. The current record terminates.
2. A new record begins.
3. Format control returns to the repeat specification for the rightmost specification group at nested level 1. If there is no group at level 1, control returns to the first descriptor in the format specification.

Examples	**Notes**
`(I5,2(3X,I2,(I4)))`	Control returns to `2(3X,I2,(I4))`
`(F4.1,I2)`	Control returns to `(F4.1,I2)`
`(A3,(3X,I2),4X,I4)`	Control returns to `(3X,I2),4X,I4`

When part or all of a format specification is repeated, the current scale factor is not changed until another scale factor is encountered. Repetition also has no effect on the `BN` and `BZ` edit descriptors.

Unformatted Input/Output

Unformatted input/output allows you to transfer data in internal representation (binary). Each unformatted input/output statement transfers exactly one record. Unformatted input/output to devices is done in binary mode.

Unformatted Input

Unformatted input is specified by the following input statement. Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "READ Statement (Executable)" in Chapter 3.

Syntax

```
READ ( unit [...]) list
```

Item	Description/Default	Restrictions
unit	Unit number of the file.	None.
list	List of variables that specifies where the data is to be transferred.	None.

If *list* is omitted, the file is moved to the next record without data transfer. The list can contain implied DO loops. For syntax and detailed information on implied DO loops, refer to "DO Statement (Executable)" in Chapter 3.

With unformatted input, the format specifier (FMT) cannot be present in the WRITE statement.

Because only one record is read when an unformatted READ statement is executed, the number of list elements must be less than or equal to the number of values in the record. A complex item requires two real values.

The type of each input value should agree with the type of the corresponding list item. A complex value in the input record, however, can correspond to two real list items, or two real values can correspond to one complex list item.

The data is transferred exactly as it is written; thus, no precision is lost.

Unformatted Output

Unformatted output is specified by the following output statement. Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "WRITE Statement (Executable)" in Chapter 3.

Syntax

```
WRITE ( unit ) [,]list
```

Item	Description/Default	Restrictions
unit	Unit number of the file.	None.
list	List of variables or expressions that specifies the data to be transferred.	None.

The *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, refer to "DO Statement (Executable)" in Chapter 3. If *list* is omitted, an empty record is written. If *list* contains a function reference, that function must not contain any READ or WRITE statements.

With unformatted output, the format specifier FMT cannot be present in the WRITE statement.

The output list must not specify more values than can fit into one record. If the specified values do not fill the record, the remainder of the record is undefined.

List-Directed Input/Output

List-directed input/output allows you to transfer data without specifying its exact format. The format of the data is determined by the data itself.

List-Directed Input

List-directed input is specified by the following input statements: Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to “READ Statement (Executable)” in Chapter 3.

Syntax

`READ * ,` *list*

`READ (` *unit* `, * [...])` *list*

Item	Description/Default	Restrictions
unit	Unit number of the internal or external file.	None.
list	List of variables that specifies where the data is to be transferred.	None.

If *list* is omitted, the file is positioned at the next record without data transfer. The list can contain implied DO loops. For syntax and detailed information on implied DO loops, refer to “DO Statement (Executable)” in Chapter 3.

The first READ statement syntax shown above transfers information from the standard input device. Unit 5 is preconnected to the standard input device. The second READ statement transfers data from a disk file or device.

List-directed input from an internal file is an extension to the ANSI 77 standard.

Data for list-directed input consists of values separated by one or more blanks, or by a slash or comma preceded or followed by any number of blanks. An end-of-record also acts as a separator except within a character constant. Leading blanks in the first record read are not considered part of a value separator unless followed by a slash or comma. Input data can also take either of the forms:

*r***c*
*r**

where:

r	is an unsigned, nonzero integer constant.
c	is a constant.

The *r***c* form means *r* repetitions of the constant *c*, and the *r** form means *r* repetitions of null values. Neither form can contain embedded blanks, except where permitted in the constant *c*.

Reading always starts at the beginning of a new record. As many records as required to satisfy the list are read unless a slash in the input record is encountered.

Embedded blanks in input values are not allowed (they are always interpreted as value separators).

The forms of values in the input record are described in Table 4-5. See "Data Types" in Chapter 2 for more details.

Table 4-5. List-Directed Input Format

Data Type	**Input Record Format**
INTEGER*2 INTEGER*4 LOGICAL*1	Same form as integer constants. **Note:** LOGICAL*1 (BYTE) requires integer input, not logical input.
REAL*4 REAL*8 REAL*16	Any valid form for real constants. In addition, the exponent can be indicated by a signed integer constant (the `D`, `E`, or `Q` can be omitted), and the decimal point can be omitted for those values with no fractional part.
COMPLEX*8 COMPLEX*16	Any valid form for complex constants. Each of the numbers can be preceded or followed by blanks or the end of a record.
LOGICAL*2 LOGICAL*4	A field of characters in which the first nonblank character (excluding an optional leading decimal point) must be a `T` for true or an `F` for false. **Note:** LOGICAL*1 (BYTE) requires integer input, not logical input.
CHARACTER	Same form as character constants. Character constants can be continued from one record to the next. The end-of-record does not cause a blank or any other character to become part of the constant. If the length of the character constant is greater than or equal to the length, *len*, of the list item, only the leftmost *len* characters of the constant are transferred. If the length of the constant is less than *len*, the constant is left-justified in the list item with trailing blanks.

The data in the input record is converted to that of the list item, following the type conversion rules given in Table 3-3.

Example

The statement:

```
READ *,s,t,x,y,z
```

and the input record:

```
ΔΔ'TOTAL'ΔΔ(42Δ,Δ1),TRUEΔΔ362ΔΔΔ563.63D6
```

cause the following assignments to take place, assuming the variable is of the specified type:

Variable	**Data Type**	**Value Assigned**
s	CHARACTER*5	'TOTAL'
t	COMPLEX*8	(42.,1.)
x	LOGICAL*4	.TRUE.
y	REAL*4	362.
z	REAL*8	563.63×10^6

A null value can be specified in place of a constant when you do not want the value of the corresponding list item to change. If the value is defined, it retains its value; if the value is undefined, it remains undefined. A null value is indicated by two successive value separators (two commas separated by any number of blanks) or by a comma before the first input value on a line.

Example

The statement:

```
READ *,x,y,z
```

and the input record:

```
Δ,5.12Δ,ΔΔ
```

cause the following assignments to take place:

Variable	**Data Type**	**Value Assigned**
x	REAL*4	Retains previous value.
y	REAL*4	5.12
z	REAL*4	Retains previous value.

Encountering an end-of-line (end-of-record) in the input record causes the read to be continued on the next record until the input list items are satisfied. If a slash (/) is encountered, the read terminates and the remaining items in the input list are unchanged.

An end-of-record is treated like a blank. An end-of-record is not itself data and is not placed in a character item when a character constant is continued on another line. (That is, character constants can be continued.)

List-Directed Output

List-directed output is specified by the following output statements: Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "PRINT Statement (Executable)" in Chapter 3 and "WRITE Statement (Executable)" in Chapter 3.

Syntax

PRINT * , *list*

WRITE (*unit* , * [...]) *list*

Item	Description/Default	Restrictions
unit	Unit number of the internal or external file.	None.
list	List of variables or expressions that specifies the data to be transferred.	See "Semantics".

Semantics

If *list* contains a function reference, that function must not contain any READ or WRITE statements. The list can contain implied DO loops. For syntax and detailed information on implied DO loops, refer to "DO Statement (Executable)" in Chapter 3.

List-directed output to an internal file is an extension to the ANSI 77 standard.

The PRINT statement transfers information to the standard output unit. The WRITE statement transfers information to external files or devices. Unit number 6 is preconnected to the standard output file.

The forms of values in a list-directed output record are described in Table 4-6. See "Data Types" in Chapter 2 for more details.

Table 4-6. List-Directed Output Format

Data Type	Output Record Format
INTEGER*2 INTEGER*4 LOGICAL*1	Output as integer constants. **Note:** LOGICAL*1 (BYTE) produces integer output, not logical output.
REAL*4 REAL*8 REAL*16	Output with or without an exponent, depending on the magnitude of the value.
COMPLEX*8 COMPLEX*16	Output as two REAL*4 or REAL*8 values, separated by commas and enclosed in parentheses.
LOGICAL*2 LOGICAL*4	Output as a `T` for the value true and an `F` for the value false. **Note:** LOGICAL*1 (BYTE) produces integer output, not logical output.
CHARACTER	A character value is not delimited by apostrophes or quotation marks, and each apostrophe or quotation mark within the value is written as one character.

Every value is preceded by exactly one blank, except character values. Trailing zeros after a decimal point are omitted. A blank character is also inserted at the beginning of each record to provide carriage control when the file is printed.

Sample Program Data

Internal Values	Data Type
`a` = 11.15	REAL*4
`b` = .11145D−05	REAL*8
`c` = (10 , 3.0)	COMPLEX*8
`d` = (1.582D−03 , 4.9851)	COMPLEX*16
`e` = .TRUE.	LOGICAL*2
`f` = .FALSE.	LOGICAL*4
`i` = 11250	INTEGER*4
`j` = −32799	INTEGER*4
`n` = 'PROGRAM NAME'	CHARACTER*15
`p` = 'test.out'	CHARACTER*8
`r` = 32Q−4300	REAL*16

Sample Output from Sample Program Data

Output Statement	Output Record
`PRINT *,a,i`	`Δ11.15Δ11250` (Note)
`WRITE(6,*)c`	`Δ(10.,3.)`
`WRITE(6,*)j,e`	`Δ-32799ΔT`
`PRINT *,b`	`Δ1.1145E-6` (Note)
`WRITE(6,*)d`	`Δ(1.582E-3,4.9851)`
`WRITE(6,*)n,p`	`PROGRAMΔNAMEΔΔΔtest.out`

Note: Output to the standard output unit. The first output character (not shown) is converted to provide single-spacing carriage control.

The total length of each list-directed output record to an external file is 72 bytes or less, including carriage control. Items that would overflow the 72-byte record if added to the current record cause the current record to be written out and a new record started. The item that caused the overflow begins the new record. Character strings longer than 71 characters are broken into as many records as necessary, with each record given a leading blank for carriage control.

List-directed output to an internal file uses the item length of the internal file as the record length to determine where output items must be broken. Slashes, as value separators, and null values are not output by list-directed formatting.

Namelist-Directed Input/Output

Namelist-directed input/output statements are similar in function to formatted statements. However, they use different mechanisms to control the translation of data. Formatted I/O statements use explicit format specifiers; namelist-directed I/O statements use data types.

Namelist Specifier

The namelist specifier is an argument used in an I/O statement to specify that namelist-directed I/O is being used and to identify the group name of the entities that may be modified on input or written on output.

Syntax

[`NML =`] *group_name*

Item	Description/Default	Restrictions
group_name	Name of a list of variables or array names previously defined in a NAMELIST statement.	None.

If the prefix `NML=` is omitted, the unit number must be the first item in the list and the group name must be the second. A namelist specifier cannot occur in a statement that contains a format specifier.

Namelist-Directed Input

The namelist-directed READ statement reads external records sequentially until it finds the specified group_name. Using the data types of the entities in the corresponding NAMELIST statement, it then translates the data from external to internal form and assigns the translated data to the specified namelist entities.

External File Syntax

$*group_name entity1* = *value1* [, *entity2* = *value2*] [, . . .]
$ [END]

Example

```
NAMELIST /FOO/I,J,K,/BOO/L,M,K
CHARACTER*10 K
INTEGER I,J,L,M
  :
OPEN(8,...)
READ(UNIT=8,NML=BOO)
```

A sample data file for `FOO` follows. Notice that the comma which follows the string to initial `K` is optional.

```
$BOO L=10, M=45,
   K='hellohello'
$
$FOO I=23
       K='hello',
       J=11
$END
```

Namelist-directed input data can contain a multiplier to repeat a value. Input data can take either of the following forms:

*r***c*
*r**

where *r* is an unsigned, nonzero integer constant and *c* is a constant. The *r***c* form produces *r* repetitions of the constant *c*. The *r** form produces *r* repetitions of null values. Neither form can contain embedded blanks, except where permitted in constant *c*.

The namelist-directed READ statement changes the values of only those namelist entities that appear in the input data. Similarly, when character substrings and array elements are specified, the values of only the specified variable substrings and array elements are changed. When a list of values is assigned to an array name, the first value in that list is assigned to the first element of the array, the second value to the second element, and so on. The consecutive commas indicate that the value of the array element in that position remains unchanged.

When a list of values is assigned to an array *element*, the assignment begins with the specified array element rather than with the first element of the array.

You can input a question mark (?) to a namelist READ statement to print the current value of all the items on the namelist. Input is allowed from either a terminal or file and output is written to unit 6. Unit 6 must be connected to a terminal for the question mark to generate output. If unit 6 is not connected to a terminal, the question mark is ignored.

The following is a sample namelist program and its output using the query feature:

```
      PROGRAM query

      INTEGER lu, iu
      PARAMETER (lu = 6)
      PARAMETER (iu = 5)

      CHARACTER*1 i1
      CHARACTER*10 j1
      CHARACTER*255 k1
      DATA i1 /' '/
      DATA j1 /'          '/
      NAMELIST/n/i1,j1,k1
      DO i = 1,255
        k1(i:i+1) = ' '
      END DO

      WRITE(lu,25) 'Enter ? to see current namelist values'
      WRITE(lu,25) 'then enter the values i1 and j1 as shown below:'
      WRITE(lu,25) '$N'
      WRITE(lu,25) 'i1=''I'' j1=''CAN'''
      WRITE(lu,25) '$END'
      READ(iu,nml=n)

      WRITE(lu,25) '----------------------------------------------'
      WRITE(lu,25) 'Enter ? to see i1 and j1 initialized'
      WRITE(lu,25) 'then enter the value k1 as shown below:'
      WRITE(lu,25) '$N'
      WRITE(lu,25) 'k1=''ANSWER'''
      WRITE(lu,25) '$END'
      READ(iu,nml=n)

      WRITE(lu,25) '----------------------------------------------'
      WRITE(lu,25) 'Enter ? to verify i1, j1 and k1 initialized'
      WRITE(lu,25) 'then enter the following:'
      WRITE(lu,25) '$N'
      WRITE(lu,25) '$END'
      READ(iu,nml=n)

      WRITE(lu,25) '----------------------------------------------'
      WRITE(lu,25) 'Enter ? again to verify multiple question marks'
      WRITE(lu,25) 'then enter the following:'
      WRITE(lu,25) '$N'
      WRITE(lu,25) '$END'
      READ(iu,nml=n)

      STOP
25    FORMAT(a)
      END
```

The output is shown below. User input is underlined.

```
Enter ? to see current namelist values
then enter the values i1 and j1 as shown below:
$N
i1='I' j1='CAN'
$END
?
 $N
 I1     =' '
 J1     ='          '
 K1     ='



                                                  '
 $END
$N
i1='I' j1='CAN'
$END
-----------------------------------------------
Enter ? to see i1 and j1 initialized
then enter the value k1 as shown below:
$N
k1='ANSWER'
$END
?
 $N
 I1     ='I'
 J1     ='CAN       '
 K1     ='



                                                  '
 $END
$N
k1='ANSWER'
$END
```

```
-----------------------------------------------
Enter ? to verify i1, j1 and k1 initialized
then enter the following:
$N
$END
?
 $N
 I1      ='I'
 J1      ='CAN        '
 K1      ='ANSWER



                                                         '
 $END
$N
$END
-----------------------------------------------
Enter ? again to verify multiple question marks
then enter the following:
$N
$END
?
 $N
 I1      ='I'
 J1      ='CAN        '
 K1      ='ANSWER



                                                         '
 $END
$N
$END
```

Namelist-Directed Output

The namelist-directed WRITE statement is a sequential write that translates internal storage to external records according to the specified namelist. Only the brief forms of the relevant syntax elements are shown here. For the complete syntax, refer to "READ Statement (Executable)" in Chapter 3.

Syntax

WRITE (*unit* , [NML =]*group_name* [...])

Namelist-directed output is suitable for use as namelist-directed input.

Namelist Output File Syntax

$*group_name*
entity = *value*
⋮
$END

Each entity is begun on a separate line.

Example

```
CHARACTER*5 BLA(2)
INTEGER HA,SOO
LOGICAL KOG
NAMELIST /BLANK/HA,SOO,KOG,BLA

BLA(1) = 'hello'
BLA(2) = 'HELLO'
KOG = .FALSE.
HA = 123
SOO = 17
    ⋮
WRITE(xxx,NML=BLANK)
    ⋮
END
```

Output:

```
$BLANK
HA = 123
SOO = 17
KOG = F
BLA = 'hello', 'HELLO'
$END
```

5 File Handling

This chapter describes the OPEN statement and the procedures used for MPE/iX file handling.

The OPEN Statement

Files are always referenced in FORTRAN 77 programs using unit numbers. Under MPE/iX, FORTRAN 77 preconnects two units at the beginning of every program. Unit five is connected to $STDIN by the formal file designator FTN05 and unit six is connected to $STDLIST by the formal file designator FTN06. If a file equation is present for either FTN05 or FTN06, the temporary and permanent file domains are searched before a new file is created. An OPEN statement is not required to perform I/O with units five and six. Units five and six can be reassigned using the OPEN statement at any time.

The OPEN statement connects MPE/iX files to FORTRAN unit numbers. If present, the FILE clause in the OPEN statement must specify a legal MPE/iX file name that is connected to the specified unit. If the status clause on the OPEN statement specifies SCRATCH, FORTRAN 77 creates an MPE/iX nameless file and deletes it when the file is closed. An OPEN status of UNKNOWN causes FORTRAN to open or create the file indicated by the FILE clause, or if no FILE clause is present, to create an MPE/iX nameless file. All files created by FORTRAN are created as MPE/iX "NEW" and are saved unless the DELETE option appears in the CLOSE statement (except SCRATCH files, which are always deleted). FORTRAN 77 creates files of type BINARY if the FORM clause specifies UNFORMATTED, otherwise it creates files of type ASCII. The RECL parameter in the OPEN statement always specifies byte length, not word length as is sometimes used in MPE/iX.

The FNUM Procedure

Occasionally you might want to use MPE/iX intrinsics to perform I/O directly. You can intermix FORTRAN I/O and intrinsic I/O to the same file by using the FNUM procedure.

The FNUM procedure can be called from an HP FORTRAN 77 program as follows:

```
i = FNUM(unit)
```

Item	Description/Default	Restrictions
`unit`	Positive integer (INTEGER*2 or INTEGER*4) that specifies the file table entry for which the MPE/iX system file is to be extracted.	Must be nonnegative.

Refer to the *HP FORTRAN 77/iX Programmer's Guide* for an example of using the FNUM function.

The FSET Procedure

The FSET procedure changes the MPE/iX operating system file number assigned to a given FORTRAN 77 logical unit number.

The FSET procedure is called from an HP FORTRAN 77 program as follows:

```
CALL FSET(unit, newfile, oldfile)
```

Item	Description/Default	Restrictions
`unit`	Positive integer constant or variable (INTEGER*2 or INTEGER*4) that specifies the file table entry for which the change is to be made.	Must be nonnegative.
`newfile`	Positive integer constant or variable (INTEGER*2 or INTEGER*4) that specifies the new MPE/iX file number to be assigned to `unit`.	None.
`oldfile`	Integer variable to which the procedure returns the old value of the file number that was assigned to `unit`.	None.

Arguments to FSET may be INTEGER*2 or INTEGER*4, but all the arguments in a given call must match in size.

Refer to the *HP FORTRAN 77/iX Programmer's Guide* for an example of using the FSET procedure.

The UNITCONTROL Procedure

The UNITCONTROL procedure enables a FORTRAN 77 program to request several actions for any FORTRAN logical unit.

The UNITCONTROL procedure is called from an HP FORTRAN 77 program as follows:

```
CALL UNITCONTROL(unit,opt)
```

Item	Description/Default	Restrictions
`unit`	Positive integer (INTEGER*2 or INTEGER*4) that speccifies the unit number of the file to be used.	Must be nonnegative.
`opt`	Integer (INTEGER*2 or INTEGER*4) that specifies the option (see table 5-1).	None.

The options for the UNITCONTROL intrinsic are listed in table 5-1.

Table 5-1. UNITCONTROL Options

Option	Description
-1	Rewind (but don't close file).
0	Backspace.
1	Write an EOF mark.
2	Skip backward to a tape mark.
3	Skip forward to a tape mark.
4	Unload the tape and close the file.
5	Leave the tape loaded and close the file.
6	Convert the file to prespacing.
7	Convert the file to postpacing.
8	Close the file.

Refer to the *HP FORTRAN 77/iX Programmer's Guide* for an example of using the UNITCONTROL intrinsic.

Automatically Opening Files

For compatibility with FORTRAN 66/V, the FORTRAN 77 I/O library automatically opens units 1 through 99 (excluding 5 and 6) to the formal file designators FTN01 throught FTN99, respectively. OPEN statements are not required for these files, though a :FILE equation is usually required. If no :FILE equation is used, and the first executed I/O statement to that unit is not READ or WRITE, the file will be opened with direct unformatted attributes.

If the first I/O to a unit is READ or WRITE, the format of the file is based on the attributes of the READ/WRITE specified by the user. If the READ is a sequential formatted READ, the file will be opened with sequential formatted attributes.

```
      PROGRAM main
C     Write to a unit without explicitly opening it.
      WRITE(50,*) 'hello world'
      STOP
      END
```

In the above program, the formal file `FTN50` is opened with sequential formatted attributes because no explicit OPEN of unit 50 is encountered prior to the WRITE and no :FILE equation is present for `FTN50`.

For more information on the OPEN statement, see chapter 3, “FORTRAN Statements”.

6

Compiling and Running HP FORTRAN 77/iX Programs

Before a FORTRAN 77 program can be executed, the following must occur:

1. The FORTRAN 77 compiler translates the source code into an object file.
2. The HP Link Editor/iX (LINKEDIT.PUB.SYS) links one or more object files into a program file.
3. The MPE/iX operating system loads and executes the program file.

You can advance through each of these steps independently, controlling each process along the way. In particular, you can use the MPE/iX commands FTNXL, LINK, and RUN for steps 1, 2, and 3, respectively.

Alternatively, you can combine steps with a single MPE/iX command. The MPE/iX command FTNXLLK performs steps 1 and 2 and the command FTNXLGO performs steps 1, 2, and 3.

This chapter discusses the MPE/iX commands FTNXL, FTNXLLK, and FTNXLGO in detail and also explains how you can invoke the FORTRAN 77 compiler with the RUN command. Refer to the *HP Link Editor/iX Reference Manual* for details on the LINK command.

The FTNXL Command

The MPE/iX command FTNiX invokes the FORTRAN 77 compiler and causes the compiler to process the specified source program and generate object code to an object file.

Syntax

$$\texttt{FTNXL}\ \left[\,\textit{textfile}\,\right]\left[\texttt{,}\ \left[\,\textit{objectfile}\,\right]\left[\texttt{,}\ \left[\,\textit{listfile}\,\right]\right]\right]$$
$$\left[\left\{\begin{array}{l}\texttt{; INFO=}\\ \texttt{,}\end{array}\right\}\texttt{"}\ \textit{text}\ \texttt{"}\right]$$

Item	Description/Default	Restrictions
textfile	The name of the input file that the FORTRAN 77 compiler will read; the default is $STDIN.	Must be an ASCII file or a system defined file name such as $STDIN.
objectfile	The name of the relocatable object file or relocatable library file on which the compiler will write the object code; the default is $OLDPASS.	Must be a binary file or a system defined file name such as $OLDPASS; must have code type NMOBJ or NMRL.
listfile	The name of the file on which the compiler will write the program listing; the default is $STDLIST.	Must be an ASCII file or a system defined file name such as $STDLIST.
text	A specification of initial compiler directives.	Compiler directives must be separated by semi-colons or commas; "**$**" will be inserted at the beginning of the string automatically.

Description

If *textfile* is omitted, the default file is $STDIN, which is the current input device. In a session, this device is the terminal, allowing you to enter source code interactively. For interactive mode, a special prompt (>) appears on the screen. Indicate the end of the source code by entering a colon (:) immediately after the prompt. If *listfile* is $STDLIST, the listing is echoed back to the terminal. If *listfile* is $NULL or a file other than $STDLIST, the listing is not echoed back to the terminal, but is directed to $NULL or to the specified file.

If *objectfile* is omitted, the file $OLDPASS is the default. If $OLDPASS does not exist, the system uses $NEWPASS and renames it to $OLDPASS at the end of the compile. You can create a new object file in one of three ways:

- By specifying a nonexistent object file in the FTNXL command. This creates a permanent object file of the correct type.

- By saving a default $OLDPASS object file with the SAVE command.
- By building a new file of NMOBJ or NMRL type with the BUILD command. The *filecode* parameter must be NMOBJ or NMRL, as in the following commands:

  ```
  :BUILD MYOBJ; DEV=DISC; CODE=NMOBJ
  :BUILD MYRL; DEV=DISC; CODE=NMRL
  ```

If the object file is of type NMRL, any existing module with an entry point duplicating one in the current compilation unit will be replaced. See the RLFILE and RLINIT compiler directives in Chapter 7 for additional information.

If *listfile* is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job.

The *text* field of the INFO parameter permits you to specify the compiler directives that initially take effect. FORTRAN 77 places a dollar sign (`$`) in front of the *text* field and places the string before the first line of source code in the text file. For example,

```
FTNXL myfile ;INFO="SHORT;HP3000_16"
```

The FTNXLLK Command

The MPE/iX command FTNXLLK compiles a FORTRAN 77 program into an object file and then links this object file into a specified program file.

Syntax

$$\texttt{FTNXLLK}\ \left[\,\textit{textfile}\,\right]\left[\,\texttt{,}\ \left[\,\textit{progfile}\,\right]\left[\,\texttt{,}\ \left[\,\textit{listfile}\,\right]\right]\right]$$
$$\left[\begin{Bmatrix}\texttt{; INFO=}\\ \texttt{,}\end{Bmatrix}\texttt{"}\ \textit{text}\ \texttt{"}\right]$$

Item	Description/Default	Restrictions
textfile	The name of the input file that the FORTRAN 77 compiler will read; the default is $STDIN.	Must be an ASCII file or a system defined file name such as $STDIN..
progfile	The name of the file that will contain the program after the compile and link is complete. The default is $OLDPASS.	If the specified file exists, the file must have the file code NMPRG. If the file does not exist, a permanent program file is created.
listfile	The name of the file on which the compiler will write the program listing; the default is $STDLIST.	Must be an ASCII file or a system defined file name such as $STDLIST.
text	A specification of initial compiler directives.	None.

Description

If *textfile* is omitted, the default file is $STDIN, which is the current input device. In a session, this device is the terminal, allowing you to enter source code interactively. For interactive mode, a special prompt (>) appears on the screen. Indicate the end of the source code by entering a colon (:) immediately after the prompt. If *listfile* is $STDLIST, the listing is echoed back to the terminal. If *listfile* is $NULL or a file other than $STDLIST, the listing is not echoed back to the terminal, but is directed to $NULL or to the specified file.

You can create a new program file in one of three ways:

- By specifying a nonexistent program file in the FTNXLLK command. This creates a permanent file of the correct type.
- By saving a default $OLDPASS program file with the SAVE command.
- By building a new program file of NMPRG type with the BUILD command. The *filecode* parameter must be NMPRG, as in the following command:

```
:BUILD MYPRG; CODE = NMPRG
```

If the object file is of type NMRL, any existing module with an entry point duplicating one in the current compilation unit will be replaced. See the RLFILE and RLINIT compiler directives in Chapter 7 for additional information.

If you specify an existing program file, the system reuses this file. An error occurs if this file is too small or if the *filecode* parameter is not NMPRG.

If *listfile* is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job.

The *text* field of the INFO parameter permits you to specify the compiler directives that initially take effect. FORTRAN 77 places a dollar sign (**$**) in front of the *text* field and places the string before the first line of source code in the text file.

The FTNXLGO Command

The MPE/iX command FTNXLGO compiles, prepares, and executes a FORTRAN 77 program. After successful completion of FTNXLGO, the program file is the temporary file $OLDPASS, which you can save using the MPE/iX SAVE command.

Syntax

$$\texttt{FTNXLGO}\ \left[\,\textit{textfile}\,\right]\left[\,\texttt{,}\ \left[\,\textit{listfile}\,\right]\right]\ \left[\left\{\begin{matrix}\texttt{; INFO=}\\ \texttt{,}\end{matrix}\right\}\texttt{"}\ \textit{text}\ \texttt{"}\right]$$

Item	Description/Default	Restrictions
textfile	The name of the input file that the FORTRAN 77 compiler will read; the default is $STDIN.	Must be an ASCII file or a system defined file name such as $STDIN.
listfile	The name of the file on which the compiler will write the program listing; the default is $STDLIST.	Must be an ASCII file or a system defined file name such as $STDLIST.
text	A specification of initial compiler directives.	None.

Description

If *textfile* is omitted, the default file is $STDIN, which is the current input device. In a session, this device is the terminal allowing you to enter source code interactively. For interactive mode, a special prompt (>) appears on the screen. Indicate the end of the source code by entering a colon (:) immediately after the prompt. If *listfile* is $STDLIST, the listing is echoed back to the terminal. If *listfile* is $NULL or a file other than $STDLIST, the listing is not echoed back to the terminal, but is directed to $NULL or to the specified file.

If *listfile* is omitted, the system assigns the file $STDLIST as the default file. Typically, this is the terminal in a session or the printer in a batch job.

The *text* field of the INFO parameter permits you to specify the compiler directives that initially take effect. FORTRAN 77 places a dollar sign ($) in front of the *text* field and places the string before the first line of source code in the text file.

If $OLDPASS exists, the object file is $OLDPASS. Otherwise, $NEWPASS is used. If $OLDPASS exists and is of type NMRL, the file is appended to and module replacement occurs if there are any duplicate entry points. Using the RLFILE or RLINIT directives when compiling with the default object file causes $OLDPASS to be of type NMRL. See the RLFILE and RLINIT compiler directives Chapter 7 for additional information.

Running the Compiler

The FORTRAN 77 compiler is a program file named FTNCOMP in the PUB group of the SYS account. To execute FTNCOMP, use the MPE/iX command RUN or simply enter the file name.

The default source, object file, and listing files for the compiler are $STDIN, $NEWPASS, and $STDLIST, respectively. To override these default values, you must:

1. Equate the nondefault file with its formal designator using an MPE/iX FILE command.
2. Select an appropriate value for the PARM parameter of the RUN command. This value indicates which files are not defaulted.

The FORTRAN 77 compiler recognizes the formal file designators listed in Table 6-1.

Table 6-1. Formal File Designators

Formal Designator	File
FTNTEXT	Source file
FTNOBJ	Object file
FTNLIST	Listing file

The PARM parameter of the RUN command indicates which files have appeared in the file equations. The compiler opens these files instead of the default files. For the FORTRAN 77 compiler, the PARM parameter accepts an integer value in the range 0 to 7, as shown in Table 6-2.

Table 6-2. Values for the PARM Parameter

Value	Files Present in the FILE Command
0	None
1	Source
2	Listing
3	Listing, source
4	Object
5	Object, source
6	Object, listing
7	Object, listing, source

The low order three bits of the PARM field represent these three files:

Bit 29	**Bit 30**	**Bit 31**
object	listing	source

If the PARM parameter sets a bit for the text file and no FTNTEXT file equation exists, an attempt is made to use a permanent file named FTNTEXT. If the permanent file FTNTEXT does not exist, an error is generated.

If the PARM parameter sets a bit for either the listing file or the object file and no file equation exists for FTNLIST or FTNOBJ, the compiler creates a permanent file with the name FTNLIST or FTNOBJ to which the appropriate output is directed. On the other hand, if a file equation exists and the bit is not set in the PARM value, the compiler uses the default file.

Setting PARM to 0 is equivalent to the FTNXL command with no parameters.

The RUN command also has an optional INFO parameter. FORTRAN 77 places a dollar sign ($) in front of the *text* field and places the string before the first line of source code in the text file. Thus, as with the FTNXL, FTNXLLK, and FTNXLGO commands, you can use the INFO parameter of the RUN command to specify initial compiler directives.

Example

```
:FILE FTNTEXT=SOURCEX
:FILE FTNOBJ=SOURCEO
:RUN FTNCOMP.PUB.SYS; PARM=5; INFO="TABLES"
```

The commands above runs the compiler (FTNCOMP.PUB.SYS), reads the source from SOURCEX, and outputs the object file into SOURCEO. By default, the listing is output to the terminal (STDLIST) and the TABLES directive is placed before the first line of source.

Passing Run Command Parameters

A maximum of two input parameters from a program's RUN command can be passed to a program. One parameter must be a CHARACTER*(*) data type and the other an INTEGER*2 or INTEGER*4 type. For example, if you want to pass two parameters to the program named `test`, where one parameter is a character string and other is an integer, you need these statements:

```
PROGRAM test(p1,p2)
CHARACTER*(*) p1
INTEGER p2
```

In the program's RUN command, the character parameter is identified with the INFO string and the INTEGER parameter is identified with the PARM word. Data is passed to the program `test` with the following RUN command:

```
:RUN test; INFO="infile";PARM=3
```

Listing Format

The following is an example of the compiler listing format with symbol table generation specified. On the left side of the compiler listing, there are three columns of numbers:

- Column one lists the statement numbers
- Column two lists the line numbers
- Column three lists the nesting level numbers

```
PAGE    1  HEWLETT-PACKARD   HP31501A.00.01
   HP FORTRAN 77  (C) HEWLETT-PACKARD CO. 1987   THURS, JAN  1, 1987, 12:01 PM


    0     1     $TABLES
    1     2           PROGRAM fibonacci
    1     3
    2     4           INTEGER*4 count, fibs, i, j, k
    2     5
    3     6     100   WRITE(6,10)
    4     7           READ(5,*) fibs
    5     8           IF (fibs .LT. 1) THEN
    6     9  1           WRITE(6,20)
    7    10  1           GOTO 100
    8    11  1        ENDIF
    9    12           i = 0
   10    13           j = 1
   11    14           k = 1
   12    15           count = 1
   13    16           DO WHILE (count .LE. fibs)
   14    17  1           WRITE(6,30) count, k
   15    18  1           i = j
   16    19  1           j = k
   17    20  1           k = i + j
   18    21  1           count = count + 1
   19    22  1        END DO
   20    23     10    FORMAT (' How many Fibonacci numbers would you like?')
   21    24     20    FORMAT (' Sorry, number must be greater than zero.')
   22    25     30    FORMAT (' Fibonacci number ',I4,' is ',I4)
   23    26           END
```

```
Name              Class             Type              Offset     Location
----              -----             ----              ------     --------

COUNT             Variable          Integer*4         SP -72     Local
fibonacci         Program
FIBS              Variable          Integer*4         SP -68     Local
I                 Variable          Integer*4         SP -64     Local
J                 Variable          Integer*4         SP -60     Local
K                 Variable          Integer*4         SP -56     Local
10                Stmt Label        Format                         20
20                Stmt Label        Format                         21
30                Stmt Label        Format                         22
100               Stmt Label        Executable                      3


  NUMBER OF ERRORS =     0   NUMBER OF WARNINGS =    0
  PROCESSOR TIME   0: 0: 4   ELAPSED TIME      0: 0:45
  NUMBER OF LINES =     26
```

7

Compiler Directives

Compiler directives are commands within the source program that indicate to the compiler exactly how it (or a program it is compiling) is to function.

A compiler directive must begin with a $ in column 1. A compiler directive can be continued by using a backslash (\) to terminate the line you want to continue. Each continuation line must begin with a $. A compiler directive cannot occur within a continued FORTRAN statement. More than one directive can be placed on a line by separating the directives with commas or semicolons, except when indicated.

Compiler options specified in the file take precedence over compiler options specified on the command line.

In general, compiler directives specified in your program will affect all files that are included (using the INCLUDE statement) in that file. Compiler directives that appear before an executable statement have a global effect on all routines within that file.

If ON or OFF is applicable but not specified, ON is assumed. For example, the LIST directive can be specified as any of the following, with the first two equivalent:

```
$LIST ON
$LIST
$LIST OFF
```

The keywords OPTION and CONTROL can be used, but are not necessary.

They are included for backward compatibility. Thus, the following three are equivalent to the preceding:

```
$OPTION LIST ON
$OPTION LIST
$OPTION LIST OFF
```

as are these three:

```
$CONTROL LIST ON
$CONTROL LIST
$CONTROL LIST OFF
```

In contrast to other statements, blanks are significant within compiler directives.

The directive name or the words ON or OFF can be written in any combination of uppercase and lowercase letters.

If any directive other than those listed in this manual is used, it produces the warning message

```
Warning:  Compiler option identifier expected (724)
```

In this manual, “program head” is any of the following statements: PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA.

Effects of the Directives

Table 7-1 lists the default condition if a compiler directive is omitted. The default condition remains in effect until specifically changed.

Table 7-1. Default State of the Compiler Directives

Compiler Directive	Default State
ANSI	OFF
CHECK_ACTUAL_PARM	Level 3
CHECK_FORMAL_PARM	Level 3
CHECK_OVERFLOW	INTEGER
CODE	ON
CODE_OFFSETS	OFF
CONTINUATIONS	19 lines
COPYRIGHT	NONE
DEBUG	OFF
ELSE	NONE
ENDIF	NONE
EXTERNAL_ALIAS	NONE
FTN3000_66	OFF
HP3000_16	OFF
IF	NONE
INCLUDE	NONE
INIT	OFF
LINES	56
LIST	ON
LIST_CODE	OFF
LITERAL_ALIAS	OFF
LOCALITY	NONE
LONG	LONG
LOWERCASE	ON
NLS	OFF
NOSTANDARD	OFF

Continued on next page

Table 7-1.
Default State of the Compiler Directives (continued)

Compiler Directive	Default State
ONETRIP	OFF
OPTIMIZE	OFF
PAGE	NONE
PAGEWIDTH	80
RANGE	OFF
RLFILE	NONE
RLINIT	NONE
SAVE_LOCALS	OFF
SET	NONE
SHORT	(LONG)
STANDARD_LEVEL	HP
SUBTITLE	NONE
SYMDEBUG	OFF
SYSINTR	SYSINTR.PUB.SYS
SYSTEM INTRINSIC	NONE
TABLES	OFF
TITLE	NONE
UPPERCASE	OFF
VERSION	NONE
WARNINGS	ON
XREF	OFF

The following directives have special effects:

- ALIAS, if placed before the PROGRAM statement, has a global effect (that is, it affects all procedures or function calls throughout the program). If placed after the PROGRAM statement, the directive is in effect only in the current program unit (that is, it only has a local effect).
- COPYRIGHT must be issued separately for each program unit; that is, it only has a local effect.
- INCLUDE is invoked once for each file to be included.
- PAGE takes effect (once for each occurrence) at the place it is issued.

- SYSTEM INTRINSIC, if placed before the PROGRAM statement, has a global effect (that is, it declares system intrinsics throughout the program). If placed before the first nondirective statement of a program unit, it takes effect thereafter. This directive works exactly like the SYSTEM INTRINSIC statement, except with a global effect.
- VERSION must be issued separately for each program unit; that is, it only has a local effect.

In this chapter, the term "program head" is any of the following statements: PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA.

ALIAS Directive

The ALIAS directive specifies that a subroutine, function, entry, or common block name has an external name different from its internal name and, optionally, that a subroutine or function has a nonstandard calling sequence or parameter passing mechanism.

Specifying the name of the language automatically generates the appropriate type of parameter for that language. Thus, specifying Pascal causes all parameters, including character, to be passed by reference; specifying C causes all noncharacter parameters less than or equal to 64 bits to be passed by value and all others by reference. The ALIAS directive applies to subroutines, entries, and functions used externally; the directive does not apply to the program unit being defined.

The ALIAS directive can function in two modes: local and global.

Syntax

```
                                                     ┌      ┐
                   ┌   ┌ 'external_name' ┐ ┐   │Pascal│
$ALIAS name        │ = ┤                 ├ │   │C     │
                   └   └ "external_name" ┘ ┘   │COBOL │
                                                     └      ┘
┌    ┌ %VAL   ┐         ┐
│ (  ┤ %REF   ├ [,...]) │
└    └ %DESCR ┘         ┘
```

name	is a named common block name if it is enclosed in slashes, or else a subroutine, function, or entry name.
external_name	is a string that can include special characters. When *external_name* is specified, the compiler changes the external name of the subroutine or common block to the specified string. *external_name* must be delimited by single quotation marks and is downshifted by default, unless an UPPERCASE or LITERAL_ALIAS directive specifies otherwise.

Default None.

Location A global ALIAS directive must appear before the program head of the program unit (or before the first statement of a default program, that is, one with no PROGRAM statement).

A local ALIAS directive must appear within the boundaries of a particular program unit; that is it must appear after the program head of the program unit (PROGRAM, SUBROUTINE, or FUNCTION statement), if any, and before any DATA, statement function, or executable statement.

Toggling/ Duration

A global ALIAS directive applies to all program units subsequent to its appearance.

A local ALIAS directive applies only to the particular program unit and is not defined for later program units.

Attempts to redefine ALIAS names generate a warning message.

Additional Information

Note The compiler always changes *external_name* to lower case, no matter how it is entered. If you want to pass this parameter in mixed case, use the LITERAL_ALIAS directive; if you want to pass it in upper case, use the UPPERCASE directive.

The language option enables the compiler to correctly pass parameters to other languages. If you make no specification, the compiler assumes you are calling an HP FORTRAN 77 subprogram.

If `%VAL`, `%REF`, or `%DESCR` appears in the ALIAS directive, it represents the parameter passing information for the given subroutine or function, which must be external. There are three parameter passing mechanisms:

`%VAL`	passing by value.
`%REF`	passing by reference (the default for noncharacter data).
`%DESCR`	passing by descriptor (the default for character data and for all procedures).

Using alternative parameter passing information allows a FORTRAN program or subprogram to call a procedure or function written in another language, such as Pascal or C. This includes system intrinsic functions (which are described in the appropriate reference manual), as well as user-written routines. (*Passing* a datum *by reference* is equivalent in C to *passing a pointer* to that datum.)

Using the language identification automatically takes care of character parameter passing for Pascal, and *all* parameter passing for Pascal if all are VAR parameters. Using the language identification automatically takes care of *all* parameter passing for C, provided C's standard conventions are followed. In C, all arrays, character variables greater than one byte, and other items that are greater than eight bytes are passed by reference. Other items (besides arrays and characters longer than one byte) that are less than or equal to eight bytes are passed by value in C.

To pass a FORTRAN CHARACTER variable by reference instead of by descriptor (in other words, to pass a pointer to the variable itself instead of passing a descriptor), do the following:

```
$ALIAS rout="extname" (%ref)
        .
        .
        .
CHARACTER*10 name
        .
        .
        .
CALL rout(name)
```

Examples

```
$ALIAS fun = '.CDBL'
$ALIAS /blk/ = '$TIME'
$ALIAS execle_2args = 'execle'(%REF,%REF,%REF,%VAL,%REF)
$ALIAS copy_time = 'sprintf' C
```

The ALIAS directive can also be used as another method of accessing some MPE/iX and library routines. However, because it is not a reliable method for accessing these, it is recommended that the interface routines be used instead whenever possible.

An example of using the ALIAS directive is shown below:

```
       PROGRAM print_time
*
$ALIAS get_time = 'time' C (%ref)
$ALIAS format_time = 'ctime' (%ref)
$ALIAS copy_time = 'sprintf' C
       INTEGER format_time
*
       CHARACTER*26 buf
       INTEGER char_ptr
       REAL*8 tmbuf
*
* get time since Jan 1, 1970 in numerical form
*
       CALL get_time (tmbuf)
*
* convert numeric to ASCII string
*
       char_ptr = format_time (tmbuf)
*
* now put C string into a FORTRAN string
*
       CALL copy_time (buf, '%s'//char(0), char_ptr)
*
       WRITE (6,*) buf
       END
```

ALIGNMENT Directive

The ALIGNMENT directive aligns COMPLEX*8 data on 4 or 8 byte boundaries as specified by the user.

Syntax

$$\texttt{\$ALIGNMENT COMPLEX\_8} \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

4	Aligns all COMPLEX*8 data on 32 bit boundaries.
8	Aligns all COMPLEX*8 data on 64 bit boundaries.
Default	64 bit boundaries.
Location	This directive may appear before any program or subprogram unit, but may not appear among executable statements within a program or subprogram unit.
Toggling/ Duration	When multiple $ALIGNMENT directives are found in a program, the last occurrence of the directive will be in effect until reset by a later occurrence of the directive.

Example

```
      PROGRAM DemonstrateAlign
      CALL initialize
      CALL modify8
      CALL test8
      CALL initialize
      CALL modify4
      CALL test4
      END

      SUBROUTINE initialize     ! initialized common block to 0.
          COMMON /block/ intarray
          INTEGER*4 intarray(4)
          DO 10 i=1,4
           intarray(i) = 0
   10     CONTINUE
      END

$ALIGNMENT COMPLEX_8 8
      SUBROUTINE modify8
      COMMON / block/ i4, c8
      INTEGER*4 i4
      COMPLEX*8 c8
      I4 = '7FFFFFFF'x
      c8 = '7FFFFFFF7FFFFFFF'x
      END
```

```
        SUBROUTINE test8
        COMMON /block/ intarray
            INTEGER*4 intarray(4)
            IF (intarray(2) .EQ. 0) THEN
              PRINT*, "The data was 8 byte aligned this time."
            ELSE
              PRINT*, "The data was 4 byte aligned this time."
            ENDIF
        END

    $ALIGNMENT COMPLEX_8 4
        SUBROUTINE modify4
            COMMON /block/ i4, c8
            INTEGER*4 i4
            COMPLEX*8 c8
            i4 = '7FFFFFFF'x
            c8 = '7FFFFFFF7FFFFFFF'x
        END

        SUBROUTINE test4
        COMMON /block/ intarray
            INTEGER*4 intarray(4)
            IF (intarray(2) .EQ. 0 THEN
              PRINT*, "The data was 8 byte aligned this time."
            ELSE IF (intarray(4) .EQ. 0) THEN
              PRINT*, "The data was 4 byte aligned this time."
            ENDIF
        END
```

Output:

The data was 8 byte aligned this time. The data was 4 byte aligned this time.

ANSI Directive

The ANSI directive turns the generation of listing information on or off in compliance with the ANSI 77 standard.

ANSI tells the compiler to include warning messages in the list file when features of FORTRAN (other than compiler directives) that are not a part of the ANSI 77 standard are used.

Syntax

$$\texttt{\$ANSI} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; warnings are not included in the list file.
Location	The ANSI directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Applies to all program units subsequent to its appearance. May be toggled.

ASSEMBLY Directive

The ASSEMBLY directive turns on or off the generation of an assembly listing. It is equivalent to the LIST_CODE directive.

The listing is written to a separate, temporary file named *FTNASSM*.

Syntax

$$\texttt{\$ASSEMBLY} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; no assembly listing is generated.
Location	The ASSEMBLY directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

CHECK_ACTUAL_PARM Directive

This directive specifies the level of checking the HP Link Editor/iX performs when a program calls a subroutine or function.

Syntax

$$\texttt{\$CHECK\_ACTUAL\_PARM} \left\{ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} \right\}$$

The level 0, 1, 2, or 3 determines the amount of information placed in the object file. The HP Link Editor/iX uses this information to indicate the level of checking on the parameters of the subroutine or function; the levels are listed in Table 7-2.

Table 7-2. Levels of Checking

Level	Description
0	No checking.
1	Check the function type.
2	Check the function type and the number of subroutine or function parameters.
3	Check the function type, the number of subroutine or function parameters, and the type of each parameter.

Default	Level 3. If the subroutine or function has a lower checking level (as found in the subroutine or function's CHECK_FORMAL_PARM directive, if specified), the HP Link Editor/iX ignores the level indicated by the CHECK_ACTUAL_PARM directive and uses the lower level. The compiler generates no parameter checking information for subroutines or functions declared SYSTEM INTRINSIC.
Location	The CHECK_ACTUAL_PARM directive can appear anywhere in the source code.
Toggling/ Duration	This directive remains in effect until the next occurrence of CHECK_ACTUAL_PARM.

CHECK_FORMAL_PARM Directive

This directive specifies the level of checking the HP Link Editor/iX performs when a subroutine or function is called.

Syntax

$$\text{\$CHECK\_FORMAL\_PARM}\left\{\begin{array}{c}0\\1\\2\\3\end{array}\right\}$$

The level 0, 1, 2, or 3 determines the amount of information placed in the object file. The HP Link Editor/iX uses this information to check the formal parameters of the declared procedure or function against the actual parameters in the calling program, subroutine, or function. The possible levels are the same as for the CHECK_ACTUAL_PARM directive and are listed in Table 7-2.

Default Level 3.

If the checking level of the subroutine or function call is lower, the HP Link Editor/iX ignores the checking level specified by the CHECK_FORMAL_PARM directive and uses the lower value.

Location This directive must appear before any nondirective statement in the program unit, including the program head.

Toggling/ Duration This directive remains in effect until the next occurrence of CHECK_FORMAL_PARM. Cannot be toggled after the appearance of nondirective statements in a program unit.

CHECK_OVERFLOW Directive

The CHECK_OVERFLOW directive generates code that traps when an overflow occurs in integer arithmetic.

You can use CHECK_OVERFLOW with the ON statement to make your program branch to a trap subroutine. If this directive is not used, the ON statement will have no effect on integer overflow errors.

Syntax

$$\texttt{\$CHECK\_OVERFLOW} \left\{ \begin{array}{l} \texttt{INTEGER\_2} \\ \texttt{INTEGER\_4} \\ \texttt{INTEGER} \end{array} \right\} \left[\begin{array}{l} \texttt{ON} \\ \texttt{OFF} \end{array} \right]$$

`INTEGER_2`	turns on code generation to catch INTEGER*2 overflows.
`INTEGER_4`	turns on code generation to catch INTEGER*4 overflows.
`INTEGER`	turns on code generation to catch both INTEGER*2 and INTEGER*4 overflows.

Default INTEGER; trap code for overflows in integer arithmetic is not generated.

For MPE V compatibility, the default is the same as if INTEGER has been specified. Therefore, to increase the performance of your program, specify $CHECK_OVERFLOW INTEGER OFF to suppress the additional code generated to perform the checking.

Location This directive can appear anywhere in your program.

Toggling/ Duration Applies to all routines subsequent to its appearance. May be toggled.

Impact on Performance The overhead for turning on INTEGER*2 checking is significant. Several extra instructions are generated for each INTEGER*2 operation. The overhead associated with INTEGER*4 checking is insignificant.

CODE Directive

The CODE directive turns on or off the generation of object code.

Syntax

$$\texttt{\$CODE}\begin{bmatrix}\texttt{ON}\\ \texttt{OFF}\end{bmatrix}$$

ON	The compiler generates object code.
OFF	The compiler checks syntax only.

Default	On; object code is generated.
Location	The CODE directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

CODE_OFFSETS Directive

The CODE_OFFSETS directive turns on or off the generation of the machine code offsets of each FORTRAN statement.

Syntax

$$\texttt{\$CODE\_OFFSETS} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; no machine code offsets are generated for individual FORTRAN statements.
Location	The CODE_OFFSETS directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

Additional Information

The offsets are listed by statement number at the end of the listing.

You cannot specify `$CODE_OFFSETS ON` if you are using optimization directives (such as `$OPTIMIZE ON`).

CONTINUATIONS Directive

The CONTINUATIONS directive defines the maximum number of continuation lines allowed in a source program statement.

Syntax

```
   $CONTINUATIONS n
```

n	is an integer from 0 to 99.
Default	19 is the maximum number of continuation lines allowed.
Location	The CONTINUATIONS directive can appear anywhere in the source file. It must be the only directive on the line.
Toggling/ Duration	The CONTINUATIONS directive applies to all source file lines subsequent to its appearance. May be reset.

COPYRIGHT Directive

The COPYRIGHT directive places a nonexecutable literal string in the binary object file. The string may be read by tools that display the object code generated.

Syntax

$$\texttt{\$COPYRIGHT} \left\{ \begin{array}{l} \text{'}\textit{copyright\_name}\text{'} \\ \text{"}\textit{copyright\_name}\text{"} \end{array} \right\} \left[\texttt{DATE} \left\{ \begin{array}{l} \text{'}\textit{copyright\_date}\text{'} \\ \text{"}\textit{copyright\_date}\text{"} \end{array} \right\} \right]$$

copyright_name	is a string that specifies a name that becomes part of the notice. Characters beyond 72 are truncated.
copyright_date	is a string that specifies the date or dates that become part of the notice. Characters beyond 72 are truncated. If *copyright_date* is omitted, the current year is used. If the `DATE` suboption is omitted, the current year is used. Although the date can be omitted, we encourage you to specify a year.

Default None; no notice is included in the binary object file.

Location The COPYRIGHT directive must appear before any nondirective statements in the program. It must precede an executable program unit (one beginning with a PROGRAM, FUNCTION, or SUBROUTINE statement). The copyright message is placed in the object and executable files. If an executable program unit does not follow the COPYRIGHT directive, the directive is ignored.

The notice also appears in the executable file.

Toggling/ Duration Applies to entire program. May not be toggled.

Additional Information

The text of the notice is:

`(C) Copyright` *copyright_date* `by` *copyright_name*`. All rights reserved. No part of this program may be photocopied, reproduced, or transmitted without the prior consent of` *copyright_name*`.`

CROSSREF Directive

The CROSSREF directive produces a cross reference listing of a program unit. It is equivalent to the XREF directive.

Syntax

```
$CROSSREF [ON ]
          [OFF]
```

Default	Off. No cross reference listing is generated.
Location	Must appear before any nondirective statements in the program unit. It must precede an executable program unit (for example PROGRAM, FUNCTION, or SUBROUTINE.)
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

Example

The following is a sample program using the CROSSREF directive.

```
$CROSSREF
      INTEGER FUNCTION icp(op)
      COMMON /chars/ iblnk,ibkslsh,iequal,irparen,ilparen,
     1 icomma,iperiod,iplus,iminus,isemi,idollar,ileta,iletz
C     ...
C     ... returns incoming priority of op
      INTEGER op
      INTEGER legal(34)
      data legal /2h( ,2h+ ,2h- ,2h* ,2h/ ,2h^ ,2h-1,2h-2,
     1            2h-3,2h-4,2h-5,2h-6,2h-7,2h-8,2h-9,2h10,
     2            2h11,2h12,2h13,2h14,2h15,2h16,2h17,2h18,
     3            2h19,2h20,2h21,2h22,2h23,2h24,2h25,2h26,
     4            2h27,2h) /
      icp=-1
      DO 20 i=1,34
      IF (op .EQ. legal(i)) GO TO 30
   20 CONTINUE
C     ...
C     ... illegal op to icp
      CALL eror(7)
      RETURN
C     ...
      30 GO TO (80,40,40,50,50,60,70,70,70,70,70,70,70,70,70,
     1       70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,
     2       70,70,70,80),i
      40 icp=1
      RETURN
      50 icp=2
      RETURN
   60 icp=4
```

```
31                RETURN
```

```
     32            70 icp=5
     33               RETURN
     34            80 icp=100
     35               RETURN
     36               END
```

The following is the cross reference listing for the above program. The default line width is 80 columns. It can be changed using the PAGEWIDTH directive.

```
SYMBOL    TYPE    FILE       LINE
------    ----    ----       ----
CHARS/    (COMMN) crossref.f $3
EROR      (PROC ) crossref.f #20
I         (VAR  ) crossref.f !15   16   25
IBKSLSH   (VAR  ) crossref.f %3
IBLNK     (VAR  ) crossref.f %3
ICOMMA    (VAR  ) crossref.f %4
ICP       (PROC ) crossref.f *2   !14  !26  !28  !30  !32  !34
IDOLLAR   (VAR  ) crossref.f %4
IEQUAL    (VAR  ) crossref.f %3
ILETA     (VAR  ) crossref.f %4
ILETZ     (VAR  ) crossref.f %4
ILPAREN   (VAR  ) crossref.f %3
IMINUS    (VAR  ) crossref.f %4
IPERIOD   (VAR  ) crossref.f %4
IPLUS     (VAR  ) crossref.f %4
IRPAREN   (VAR  ) crossref.f %3
ISEMI     (VAR  ) crossref.f %4
LEGAL     (VAR  ) crossref.f *8   @9    16
OP        (ARGMT) crossref.f %2   *7    16

   NUMBER OF ERRORS =     0   NUMBER OF WARNINGS =    0
```

`SYMBOL`	The symbol name.
`TYPE`	The class of the symbol: ■ ARGMT - argument passed to a procedure or function. ■ COMMN - name of a common block. ■ CONST - named constant in a PARAMETER statement. ■ NMLST - NAMELIST variable. ■ PROC - internal or external procedure or function name. ■ VAR - variables.
`FILE`	The file where the symbol is found.
`LINE`	One or more rows of line numbers that indicate where the symbol is found. Each line has a suffix that indicates the status of the symbol at time of access. ■ blank - symbol is being referenced. ■ `!` - symbol is being modified. ■ `*` - symbol is being defined and declared. ■ `%` - symbol is being declared. ■ `^` - symbol is being declared, defined, and modified. ■ `#` - symbol is being called. ■ `$` - symbol is declared, defined, and used. ■ `@` - symbol is declared and modified.

DEBUG Directive

The DEBUG directive enables the processing of debug lines (those with a `D` or `d` in column 1) as statement lines instead of comment lines.

Syntax

$$\texttt{\$DEBUG}\begin{bmatrix}\texttt{ON}\\ \texttt{OFF}\end{bmatrix}$$

Default	Off; lines containing a `D` in column 1 are treated as comment lines by the compiler.
Location	The DEBUG directive can appear anywhere in a program unit.
Toggling/ Duration	May be toggled. DEBUG ON remains in effect until DEBUG OFF or the end of the program is encountered.

Examples

```
     .
     .
     .
D      PRINT *, 'This line won''t print.'
$DEBUG
D      PRINT *, 'This line WILL print.'
$DEBUG OFF
D      PRINT *, 'Just another comment line.'
     .
     .
     .
```

ELSE Directive

The ELSE directive is used with the IF directive. The ELSE directive semantically parallels the FORTRAN ELSE statement.

Syntax

```
$ELSE
```

Default	None.
Location	The ELSE directive must be the only directive that appears on the line. The source following the $ELSE line is compiled only if the expression, which is part of an IF directive, has a value of *false*.

ENDIF Directive

The ENDIF directive terminates the IF directive. Each IF directive requires an ENDIF, and vice-versa.

Syntax

```
$ENDIF
```

Default	None.
Location	The ENDIF directive must be the only directive that appears on the line. It may appear after the last FORTRAN END statement.

EXTERNAL_ALIAS Directive

The EXTERNAL_ALIAS directive allows the user to specify a new external name for all occurrences of a given procedure or function name.

Syntax

$$\texttt{\$EXTERNAL\_ALIAS}\ \mathit{name}\ \texttt{=} \left\{ \begin{array}{l} \texttt{'}\mathit{new\_external\_name}\texttt{'} \\ \texttt{"}\mathit{new\_external\_name}\texttt{"} \end{array} \right\}$$

name	is a name appearing in the source code.
new_external_name	is the name to which all references to *name* are to be changed. The *new_external_name* is a string which can include special characters. It must be delimited by apostrophes (') or quotation marks ("). Only the first 15 characters are significant.

Default	None; procedures and functions retain their current names.
Location	The EXTERNAL_ALIAS directive must appear before any executable statements in the program unit.
Toggling/ Duration	Cannot be toggled using another EXTERNAL_ALIAS directive after the appearance of executable statements in the program unit. However, the LOWERCASE directive could be used as a toggle.

Additional Information

The EXTERNAL_ALIAS directive differs from the ALIAS directive in the following ways:

- The EXTERNAL_ALIAS directive changes both actual and formal uses of the name (that is, calls and entry points), whereas the ALIAS directive changes only actual references (calls).
- Only procedures or function calls can be specified; common blocks cannot be specified.
- No parameter passing or language information can be specified; use the ALIAS directive instead.

FTN3000_66 Directive

The FTN3000_66 directive allows you to specify FORTRAN 66/V features for compatibility with FORTRAN 77.

Syntax

```
            ┌LOGICALS┐ ┌   ┐
$FTN3000_66 │IO      │ │ON │
            │CHARS   │ │OFF│
            └        ┘ └   ┘
```

Default	Off.
Location	The FTN3000_66 directive can appear anywhere in a program unit.
Toggling/ Duration	You cannot turn off FTN3000_66 before an entry point if FTN3000_66 has been turned on for the primary entry point because multiple entries into a routine must have the same character passing method as the primary entry for that routine.

The following example shows an illegal use of FTN3000_66:

```
$FTN3000_66 CHARS ON
      SUBROUTINE routine1(a,b)
      CHARACTER*10 a,b,c
      WRITE(6,*) a
$FTN3000_66 CHARS OFF           ! ILLEGAL
      ENTRY ent1(b,c)
      WRITE(6,*) b,c
      END
```

FTN3000_66 CHARS

When FORTRAN 77 passes character items as arguments to a subroutine or function, the item is passed by descriptor. That is, the item is passed with a byte address and a length by value. Other languages, such as HP Pascal/iX and HP COBOL II/iX, pass character items by reference without the length. If the FTN3000_66 CHARS directive is turned on in HP FORTRAN 77/iX, character items are passed by reference without the length by value. This makes character passing compatible between other languages and FORTRAN 77.

When the FTN3000_66 directive is turned on, CHARACTER*(*) types are only permitted in the main program.

FTN3000_66 IO

The FTN3000_66 IO directive creates compatibility with FORTRAN 66/V files. This compatibility is achieved by reading and writing CHARACTER items in unformatted files in two-byte blocks and by performing multiple physical I/O operations for I/O lists that request them.

When reading and writing CHARACTER items in unformatted files in two-byte blocks, each odd-length item is padded with a trailing blank. For example, a CHARACTER*5 item is followed by a blank before the next data item so that the following item will begin on an even 16-bit boundary. The default does not leave any space between data items in unformatted files.

The following is an example of a program using the FTN3000_66 IO directive to write CHARACTER items in an unformatted file in two-byte blocks:

Example

```
$FTN3000_66 IO
      CHARACTER CH*5, CH2
      OPEN (1, FILE='MYFILE', STATUS='NEW', ACCESS='DIRECT',
     >     FORM='UNFORMATTED', RECL=8)
      CH = "AZWXY"
      CH2 = "B"
      WRITE(1, REC=1)CH,CH2
      STOP
      END
```

The following command displays the binary output of the above program in hexadecimal:

```
:fcopy from=MYFILE; to=; hex
MYFILE RECORD 0 (%0, #0)
```

Output:

```
      A Z  W X  Y _  B _
0000: 415A 5758 5920 4220
```

The underscore represents the blank inserted by the FTN3000_66 IO option to pad the first item (CH) to an even (16-bit) word boundary.

When performing multiple physical I/O operations for I/O lists that request them, the I/O library performs unformatted sequential READs and WRITEs of length longer than the file's record length to (or from) as many records as necessary. These records are used in sequence until the entire list of elements has been transferred. The default does not allow the I/O list items' combined length to exceed the record length.

Note

If the storage required exceeds the size of the record, transfer continues into the next record. This usually leaves part of that next record unused.

The following example shows the I/O library performing unformatted sequential READs and WRITEs of length longer than the file's record length to (and from) as many records as necessary.

Example

```
$FTN3000_66 IO
      PROGRAM sofx1c
      INTEGER*4 iarr(256)
      INTEGER*4 i4(18), i5, i6, i7
      OPEN (12,file='rsofx1c', FORM='UNFORMATTED', ACCESS='DIRECT',
     >         RECL=74)  ! create file with 18.5 32-bit words long
      CLOSE (12, status='KEEP')
      OPEN (12,FILE='rsofx1c', FORM='UNFORMATTED', ACCESS='SEQUENTIAL')
      WRITE (12) iarr
      REWIND 12
      READ (12) i4, i5, i6, i7  !read multiple records in pieces
      REWIND 12
      READ  (12) iarr  ! full array read
      END
```

The following file description shows that the WRITE above has written 14 physical records to hold the data in the array `iarr`. Twelve bytes at the end of the last physical record are unused:

```
 :listf rsofx1c,2


 FILENAME  CODE  ------------LOGICAL RECORD-----------  ----SPACE----
                   SIZE  TYP       EOF      LIMIT R/B  SECTORS #X MX

 RSOFX1C            37W  FB         14       4095   3       43  1 32''
```

FTN3000_66 LOGICALS

FTN3000_66 LOGICALS causes the FORTRAN 77 compiler to generate and use the same internal representations for logical data as in FORTRAN 66/V. These specific internal representations are also used by other non-HP machines. When FTN3000_66 LOGICALS is turned on, LOGICAL*2 is represented by two whole bytes and LOGICAL*4 is represented by four whole bytes. With FTN3000_66 logicals, -1 represents the value .TRUE. and zero represents the value .FALSE..

In addition to allowing compatibility of logical data, FTN3000_66 LOGICALS allows logical data to be mixed with numeric types and be treated as integers when they appear in a numeric context.

HP1000 Directive

The HP1000 directive specifies options for compatibility with FORTRAN 7X, which is the version of FORTRAN 77 on the HP 1000 computer.

Syntax

```
         ┌ ARRAYS      ┐
         │ ALIGNMENT   │ ┌ ON  ┐
$HP1000  │ STRING_MOVE │ └ OFF ┘
         └ DO_LOOP     ┘
```

Default	Off. Specify OFF or do not use the directive if your program does not rely on specific data layout or the ripple effect of the STRING_MOVE option and if it has no string array initializations in DATA statements. If used with no other parameters, the ON option turns on all the options.
Location	The HP1000 directive must appear before any nondirective statements in a program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

ARRAYS Option

The ARRAYS option causes arrays to be handled as they are in FORTRAN 7X. Using this option, you can reference an array element without specifying all of the subscripts. The option also modifies the way that character strings are assigned to integer arrays in DATA statements. Normally HP FORTRAN 77/iX requires that all subscripts be specified when accessing an array element in a program. If the ARRAYS option is turned on, you can omit one or more of the subscript specifiers from the list. The first element number for that dimension is assumed for omitted specifiers.

For example, for the real array `RARRAY(5,5,5)`:

```
R = RARRAY(4)
```

is equivalent to:

```
R = RARRAY(4,1,1)
```

and for the array `IARRAY(10,-5:5)`:

```
IARRAY(1) = 10
```

is equivalent to:

```
IARRAY(1,-5) = 10
```

In HP FORTRAN 77/iX, when you initialize an integer array with character strings, you must specify a separate string for each array element. For example:

```
INTEGER iarray(5)
DATA iarray/'abcd','efgh','ijkl','mnop','qrst'/
```

With the ARRAYS option, you can initialize more than one array element with a single string. For example, the following allows you to initialize the entire preceding array:

```
INTEGER iarray(5)
DATA iarray/'abcdefghijklmnopqrst'/
```

If the string is not long enough to initialize the entire array, as many elements as possible are initialized with the first string. If there is another string, it is used starting with the next element. If there are not enough strings, the remainder of the array is blank-filled. For example:

```
INTEGER iarray(10)
DATA iarray/'abcdef','ijkl','mnopqrstuvwxyz'/
```

results in:

```
iarray(1)= 'abcd'
iarray(2)= 'efΔΔ'
iarray(3)= 'ijkl'
iarray(4)= 'mnop'
iarray(5)= 'qrst'
iarray(6)= 'uvwx'
iarray(7)= 'yzΔΔ'
iarray(8)= 'ΔΔΔΔ'
iarray(9)= 'ΔΔΔΔ'
iarray(10)='ΔΔΔΔ'
```

(where each Δ represents a blank).

ALIGNMENT Option

The ALIGNMENT option of the HP1000 directive aligns data on 16-bit boundaries, rather than on the HP FORTRAN 77/iX boundary formats shown in the following table:

Alignment	**HP1000 Alignment Format**	**HP FORTRAN 77/HP-UX Alignment Format**
8-bit	Character	Character
16-bit	LOGICAL*2, LOGICAL*4, INTEGER*2, INTEGER*4, REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16	INTEGER*2, LOGICAL*2
32-bit		LOGICAL*4, INTEGER*4, REAL*4, COMPLEX*8
64-bit		REAL*8, REAL*16, COMPLEX*16

Use this option when you need the data layout specified by either EQUIVALENCE or COMMON statements to be exactly the same as in a FORTRAN 77 (FORTRAN 7X) program, such as when calling database intrinsics. Using the ALIGNMENT option may cause degradation in run-time performance.

Exercise caution when changing the directive between program units of a single program. In particular, mixing a specification for HP1000 alignment and native alignment can produce unpredictable results.

STRING_MOVE Option

The STRING_MOVE option causes assignments of character variables to be done byte-by-byte, which creates a ripple effect when the source and target are overlapped. For example,

```
CHARACTER*12 a
a(1:1) = '*'
a(2:12) = a(1:11)
```

results in `a` having the value:

```
'**ΔΔΔΔΔΔΔΔ
ΔΔ'
```

if STRING_MOVE is off.

If STRING_MOVE is on, the result in `a` is:

```
'************'
```

DO_LOOP Option

The DO_LOOP option causes the compiler to allow the DO loop control index to be modified within the range of the loop, as for FORTRAN 7X. Modification of the DO loop index does not affect the number of times the loop executes because the index value is established when the loop is entered. For example, the loop in the following program executes five times, even though the value of `I` is modified within the loop:

Example

```
$HP1000 DO_LOOP ON
      PROGRAM showdo
      DO i = 1,5          ! This loop will execute five times.
         PRINT *, i
         i = i + 25
      END DO
      END
```

Output:

```
1
27
53
79
105
```

HP3000_16 Directive

The HP3000_16 directive:

- allows MPE/iX programs to correctly access MPE V data files that contain floating point data
- aligns noncharacter data on 16-bit boundaries and character data on 8-bit boundaries, as on MPE V
- performs character moves the same as on MPE V when there is an overlapping character substring that should ripple across

Syntax

```
             [ ON          ]
             [ OFF         ]
  $HP3000_16 [ ALIGNMENT   ]
             [ REALS       ]
             [ STRING_MOVE ]
```

Item	Description	Restrictions
`ON`	Turns on ALIGNMENT, REALS, and STRING_MOVE options.	None.
`OFF`	Turns off ALIGNMENT, REALS, and STRING_MOVE options.	For accessing IEEE floating point data with no assumptions of MPE V data layout and rippling overlapping character substrings.
`ALIGNMENT`	Aligns noncharacter data on 16-bit boundaries.	Not for accessing MPE V floating point numbers or rippling overlapping character substrings.
`REALS`	For accessing MPE V floating point numbers.	Makes no assumptions of MPE V data alignment and rippling overlapping character substrings.
`STRING_MOVE`	Performs a byte-by-byte move of a character substring when a character substring is assigned to another character substring and the substrings overlap.	Makes no assumptions of MPE V data alignment or the format of the floating point data.

MPE V and MPE/iX have different data alignments, as summarized in Table 7-3.

Table 7-3. Data Alignment on MPE V and MPE/iX

Alignment	**MPE V**	**MPE/iX**
8-bit	CHARACTER	CHARACTER
16-bit	COMPLEX*8, COMPLEX*16 INTEGER*2, INTEGER*4 LOGICAL*2, LOGICAL*4 REAL*4, REAL*8, REAL*16	INTEGER*2 LOGICAL*2
32-bit		COMPLEX*8 INTEGER*4 LOGICAL*4 REAL*4
64-bit		COMPLEX*16 REAL*8, REAL*16

Default Off. Specify `OFF` or do not use the directive when your program uses IEEE floating point data (as opposed to MPE V floating point data), when your program does not rely on MPE V data layout, and when the ripple effect of overlapping character substrings is not necessary.

Location The HP3000_16 directive must appear before any nondirective statement in a program unit.

Toggling/ Duration Exercise caution when changing the directive between program units of a single program because mixing MPE V and MPE/iX real numbers gives incorrect results and using COMMON and EQUIVALENCE variables might produce unpredictable results if you mix MPE V and MPE/iX alignment.

Impact on Performance Use the ON option only when your program accesses MPE V floating point data, when you need data aligned as on MPE V, and when you are assigning a character substring to another character substring and the substrings overlap by one character. Of all the options, the `ON` option causes the greatest degradation in program performance.

The STRING_MOVE option has a performance degradation on all character substring moves that overlap because the compiler cannot generate code for a fast move.

Additional Information

The `ON` option turns on the `ALIGNMENT`, `REALS`, and `STRING_MOVE` options.

The `ALIGNMENT` option aligns data on 16-bit boundaries, rather than on the MPE/iX boundaries shown in Table 7-3. Use this option only if you require the preceding condition and if your program does not access files containing MPE V floating point numbers or rippling overlapping character substrings is not necessary. If your program accesses MPE V floating point numbers, or requires rippling overlapping character substrings, use the `REALS` or `STRING_MOVE` option. Use this option when you need to rely on EQUIVALENCE and COMMON specifications for MPE data layout, as when calling database intrinsics.

The `REALS` option reads, writes, and executes floating point numbers in MPE V representation, rather than using IEEE floating point standards (as used on MPE/iX). Use this option only if you require the preceding condition and if your program makes no assumptions on data alignment or rippling overlapping character substrings. If you require 16-bit data alignment, use the `ON` or `ALIGNMENT` option.

The `STRING_MOVE` option performs a byte-by-byte move whenever a character substring is assigned to another character substring and the substrings overlap. Use this option if your program expects the characters to be rippled across.

Example

```
   .
   .
   CHARACTER*10 ch
   .
   .
   .
   ch = "*         "
   ch[2:10] = ch[1:9]
```

After the assignment on MPE V, `ch` contains the following:

```
   **********
```

On MPE/iX without using the STRING_MOVE option, `ch` contains the following:

```
   **ΔΔΔΔΔΔΔΔ
```

where `Δ` represents a blank.

Refer to the *HP FORTRAN 77/iX Programmer's Guide* for more details on this directive.

IF Directive

The IF directive conditionally compiles blocks of source code.

Syntax

`$IF (`*condition_list*`)`

condition_list	is a logical expression for conditional compilation.
Default	None.
Location	The IF directive can appear anywhere in the source code. It must be the only directive that appears on the line.
Toggling/ Duration	The IF directive remains in effect until terminated by an ENDIF directive.

Additional Information

The condition list is interpreted as having identifiers for operands and .NOT., .AND., and .OR. for operators. The operator precedence ranking is from .NOT.,highest, to .OR., lowest; this precedence can be overridden with parentheses.

Directives used with IF are ELSE, ENDIF, and SET. The IF directive has an optional ELSE block, and a required ENDIF that delimits it. An identifier is given a value with the SET compiler directive.

The semantics of conditional compilation closely parallel those of the IF statement. If the expression evaluates to true, the text between the $IF and the next ENDIF or next ELSE is compiled. If the expression evaluates to false, that text is treated as a comment.

IF directives can be nested to 16 levels. If a user nests further than 16 levels, an error message is issued and the code within the illegal $IF block is not compiled.

There can be, at most, one ELSE corresponding to each IF.

The identifiers in SET and IF compiler directives are in no way related to FORTRAN variables in the source text. That is, if the same variable name is used both as an identifier for one of these directives and elsewhere within a program, the one has no effect upon the other.

Examples

```
$SET (DEBUG=.TRUE.,TOGGLE=.FALSE.)
$SET (SYSTEM1=.TRUE.)
     .
     .
     .
$IF (DEBUG .AND. TOGGLE)
$IF (SYSTEM1)

$IF (.NOT. (DEBUG .AND. TOGGLE))

$IF (.TRUE.)
     .
     .
     .

$ELSE
     .
     .
     .
$ENDIF
$ENDIF
$ENDIF
$ENDIF

$IF (SYSTEM1 .OR. TOGGLE)
     .
     .
     .
$ENDIF
     .
     .
     .
```

INCLUDE Directive

The INCLUDE directive includes the contents of a file at the current position in the source.

Syntax

$$\texttt{\$INCLUDE} \left\{ \begin{array}{l} \text{'}\textit{filename}\text{'} \\ \text{"}\textit{filename}\text{"} \end{array} \right\}$$

filename	names a file whose contents are to be included at the current position in the source.
Default	None.
Location	The INCLUDE directive can appear anywhere within a program unit.
	The INCLUDE directive can also appear as a statement starting in column seven, with no initial "$". See "INCLUDE Statement (Nonexecutable)" in Chapter 3 for a complete description.

Additional Information

The *filename* can be fully qualified by group and account names and a lockword (a password associated with an individual file). Uppercase and lowercase letters are equivalent.

The compiler reads the designated file until it encounters an EOF marker. Then, the compiler resumes processing from the source line after the INCLUDE directive. The compiler ignores any additional directive listed on the same line as the INCLUDE directive.

Included code can itself contain additional INCLUDE directives, to a maximum nesting level of eight.

Line numbering within the listing of an included file begins with one. Each line in the included file has a plus sign to the left of the line number. When the included file listing ends, the include level decreases appropriately and the previous line numbering resumes.

Example

```
$INCLUDE 'globf77'
```

INIT Directive

The INIT directive turns on or off the generation of the code that initializes all variables for the program unit it immediately precedes.

Syntax

$$\text{\$INIT} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; no code to initialize variables for the next program unit is generated.
Location	The INIT directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	The INIT directive applies only to the program unit that immediately follows it.

Additional Information

Arithmetic variables are initialized to zero, logical variables to false, and character variables to all null characters.

Using this directive can decrease the portability of a FORTRAN 77 program.

LINES Directive

The LINES directive sets the number of lines per page in the listing file to the given integer.

Syntax

```
$LINES number
```

number	is an integer greater than or equal to 5, and less than 32768.
Default	56 lines per page.
Location	The LINES directive can appear anywhere within the program unit.
Toggling/ Duration	Applies until the next LINES directive or until the end of the program.

LIST Directive

The LIST directive turns on or off inclusion of the source program in the listing file, starting with the line after the one containing the option.

Syntax

$$\texttt{\$LIST}\begin{bmatrix}\texttt{ON}\\ \texttt{OFF}\end{bmatrix}$$

ON	The source program is included in the listing program.
OFF	Only diagnostics go into the list file. Other list directives such as CODE_OFFSETS and TABLES have no effect until LIST is turned ON again.

Default	On; the source program is included in the listing file.
Location	The LIST directive can appear anywhere within the program unit.
Toggling/ Duration	Applies until the next LIST directive, if any, changes it.

LIST_CODE Directive

The LIST_CODE directive turns on or off the generation of an assembly listing. It is equivalent to the ASSEMBLY directive.

Syntax

$$\texttt{\$LIST\_CODE} \begin{bmatrix}\texttt{ON}\\\texttt{OFF}\end{bmatrix}$$

Default	Off; no assembly listing is generated.
Location	The LIST_CODE directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Once it has been turned on for a source file, it cannot be turned off.

Additional Information

The listing generated is written to a temporary file named *FTNASSM*.

LITERAL_ALIAS Directive

The LITERAL_ALIAS directive determines whether external names appearing in the ALIAS directive and the EXTERNAL_ALIAS directive are to have their case shifted or left as is.

Syntax

$$\texttt{\$LITERAL\_ALIAS} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

`ON`	Any external names in an ALIAS directive or an EXTERNAL_ALIAS directive are processed just as they appear; that is, they are neither upshifted or downshifted. This permits mixed-case external names.
`OFF`	External names are either upshifted or downshifted, depending on the setting of the UPPERCASE directive.

Default Off; external names are either upshifted or downshifted, depending on the setting of the UPPERCASE directive.

Location The LITERAL_ALIAS directive must appear before any nondirective statements in the program unit.

Toggling/ Duration Cannot be toggled after the appearance of nondirective statements in a program unit.

LOCALITY Directive

The LOCALITY directive groups the generated code from a program unit together in the same general memory area as other program units having the same LOCALITY name.

Syntax

$$\texttt{\$LOCALITY} \left\{ \begin{array}{l} \text{'}\textit{name}\text{'} \\ \text{"}\textit{name}\text{"} \end{array} \right\}$$

Default	None; procedures that call each other are not necessarily grouped in the same general memory area.
Location	The LOCALITY directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.
Impact on Performance	This directive can improve run-time performance because grouping procedures that frequently call each other makes memory access more efficient.

LONG Directive

The LONG directive sets the default size for integer and logical data types and constants to four bytes. The INTEGER and LOGICAL type names are set equivalent to INTEGER*4 and LOGICAL*4, respectively.

Syntax

```
$LONG [INTEGERS]
```

`INTEGERS`	Optional; this word has no effect.
Default	LONG (4 bytes) if neither the LONG nor the SHORT directive is given.
Location	The LONG directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Cannot be changed after the appearance of nondirective statements in a program unit.

LOWERCASE Directive

The LOWERCASE directive turns on or off shifting to lower case of all FORTRAN external names. This directive does not affect the external name of the ALIAS and EXTERNAL_ALIAS directives or intrinsic names if "$LITERAL_ALIAS ON" has been specified.

See "ALIAS Directive" and "LITERAL"'ALIAS Directive".

Specifying `$UPPERCASE ON` (or `$UPPERCASE`) is equivalent to specifying `$LOWERCASE OFF`.
Specifying `$UPPERCASE OFF` is equivalent to specifying `$LOWERCASE ON` (or `$LOWERCASE`).

Syntax

$$\texttt{\$LOWERCASE} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	On; all FORTRAN external names are shifted to lowercase.
Location	The LOWERCASE directive can appear anywhere within the program unit.
Toggling/ Duration	Can be toggled with either another LOWERCASE directive or with an EXTERNAL_ALIAS directive. (Note that the EXTERNAL_ALIAS directive cannot be used after the appearance of nondirective statements in a program unit.) If the LOWERCASE directive is toggled within a program unit, the point of declaration (or the point of first use if implicitly declared) determines the case of external names.

MIXED_FORMATS Directive

The MIXED_FORMATS directive allows you to cause a numeric format descriptor of a type different from the numeric list item to override the data type of the list item.

Syntax

$$\texttt{\$MIXED\_FORMATS} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off.
Location	The MIXED_FORMATS directive can appear anywhere in your program.
Toggling/ Duration	Remains in effect until another occurrence of the MIXED_FORMATS directive changes it.

Additional Information

Note

The MIXED_FORMATS directive is not recommended for general use. It is an extension to the ANSI 77 standard, and programs using it are not portable to systems without mixed formatting capability. If list items and actual data types do not match, a floating-point exception could occur.

When MIXED_FORMATS is on, the type of the numeric format descriptor in input or output overrides the type of the list item. However, no type conversion is done on the list item. The example below illustrates the effect of the MIXED_FORMATS directive.

Example

```
      PROGRAM mixfmts
$MIXED_FORMATS ON
      INTEGER*4 i4
      REAL*4 r4
      CHARACTER*12 std, mixed
      EQUIVALENCE (i4,r4)

C Integer type coercion
      i4 = -12
  100 FORMAT (I12)
      WRITE (mixed,100) r4
      WRITE (std,100) i4
      IF (mixed .NE. std) STOP 'Failed 1'

C Real type coercion
      r4 = 123.456
  101 FORMAT (E12.3)
      WRITE (std,101) r4
      WRITE (mixed,101) i4
      IF (mixed .NE. std) STOP 'Failed 2'
      STOP 'Passed.'
```

```
                    END
```

NLS Directive

The NLS (Native Language Support) directive supports special processing to handle foreign language text and data.

Syntax

$$\$NLS \begin{bmatrix}\texttt{LITERALS}\\\texttt{COMPARE}\end{bmatrix}\begin{bmatrix}\texttt{ON}\\\texttt{OFF}\end{bmatrix}$$

`ON`	turns on both `LITERALS` and `COMPARE`.
`OFF`	turns off all NLS processing. By default, the Native-Computer character set and collating sequence is used.
`LITERALS`	enables the handling of native language characters in strings and comments during compilation of a source program. Run-time native language I/O is also enabled.
`COMPARE`	enables all operators that deal with string comparisons (LGE, LGT, LLE, LLT) to compare string variables and string constants using the collating sequence corresponding to the specified NLDATALANG JCW. Run-time native language I/O is also enabled.

Default	Off; the Native-Computer character set and collating sequence is used.
Location	The NLS directive must appear before any nondirective statements in a program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.
Impact on Performance	Using the LITERALS option decreases compile time performance.

Additional Information

Note

`$NLS LITERALS` replaces the NLS_SOURCE compiler directive. Any occurrences of `$NLS_SOURCE` in FORTRAN source programs should be replaced with `$NLS LITERALS`.

To use the NLS directive, NLUSERLANG and NLDATALANG must be set. NLDATALANG determines the language used for string comparisons and scanning FORTRAN source programs. NLUSERLANG determines the language used to output compiler error messages. For example, to set NLDATALANG and NLUSERLANG, specify the following:

```
:SETJCW NLDATALANG 221
:SETJCW NLUSERLANG 0
```

In the example above, 221 is the JCW value for Japanese and zero is the JCW value of Native-Computer (the default value). Refer to the *Native Language Programmer's Guide* for a complete list of JCW values.

Examples

Following is a FORTRAN source file called `test`:

```
$NLS ON
      PROGRAM testnls
      CHARACTER*10 st1,st2
      st1 = 'coin'
      st2 = 'change'
      IF (LLT(st1,st2)) PRINT *,'This is the Spanish language.'
      IF (LGT(st1,st2)) PRINT *,'This is the English language.'
      STOP
      END
```

Following are examples of setting the MPE JCWs for test, followed by the output from the program for each setting:

`:SETJCW NLUSERLANG 0`	*Tells the compiler to print compile-time messages using the default message catalog.*
`:SETJCW NLDATALANG 12`	*Tells the compiler to do lexical comparisons in Spanish and to expect Spanish characters in the source file.*

```
:FTNiXLK TEST
:SAVE $OLDPASS,NLSPROG
:NLSPROG
```

Output:

```
This is the Spanish language.
```

```
:SETJCW NLDATALANG 0
:NLSPROG
```

Output:

```
This is the English language.
```

Note

The Spanish alphabet has both the letters "c" and "ch". Because "c" comes before "ch" in the Spanish alphabet, `coin` is considered to be lexically less than `change`. In English, `change` is considered to be lexically less than `coin`.

NLS_SOURCE Directive

The NLS_SOURCE directive was replaced by NLS LITERALS. We recommend that you replace all instances of NLS_SOURCE with NLS LITERALS.

NOSTANDARD Directive

The NOSTANDARD directive specifies options for compatibility with industry standard non-HP FORTRAN 77 programs.

Syntax

```
              ┌ CHARS      ┐
              │ LOGICALS   │
              │ IO         │ ┌ ON  ┐
$NOSTANDARD   │ SYSTEM     │ └ OFF ┘
              │ INTRINSICS │
              └ OPEN       ┘
```

Default	Off. If NOSTANDARD is specified without options, all options are `ON`.
Location	The LOGICALS, SYSTEM, INTRINSICS, and OPEN options must appear before any nondirective statements in the program unit, including the program head. However, the CHARS and IO options are allowed anywhere in the program unit.
Toggling/ Duration	CHARS and IO apply until another NOSTANDARD directive changes them.

CHARS Option

By default, the compiler passes character items by descriptor. That is, the address of the item is passed by reference immediately followed by the length of the item passed by value. The CHARS option causes the length parameter to be passed at the end of the parameter list by value. This option is provided for migrating programs that have character passing incompatibilities with HP FORTRAN 77. The NOSTANDARD CHARS directive is allowed anywhere in the program unit.

LOGICALS Option

The LOGICALS option causes the compiler to treat logicals as two whole bytes for LOGICAL*2 and four whole bytes for LOGICAL*4. The value .TRUE. is represented by `-1` and the value .FALSE. is represented by zero. If this option is not specified, by default HP FORTRAN 77 uses only one byte to store the logical .TRUE. or .FALSE. value, even if LOGICAL*2 or LOGICAL*4 is specified.

Note

You can specifiy LOGICAL or LOGICALS for this option.

IO Option

When using character format descriptors A[*w*] or R[*w*] with integer and real data types, the IO option causes data to be output in reverse order, starting at the right and progressing left. For more information, see "Character Format Descriptors (A, R)" in Chapter 4. The NOSTANDARD IO option is allowed anywhere in the program unit.

SYSTEM Option

Several intrinsic functions are available through the NOSTANDARD directive. They include:

DATE	Returns a string in the form `dd-mm-yy`, such as 15-09-88.
IDATE	Returns 3 integer values representing the current month, day, and year.
EXIT	Terminates the program as if a STOP statement without an argument has been encountered.
RAN	A random number generator of the multiplicative congruential type that returns a floating-point number in the range between 0.0 and 1.0 exclusively.
SECNDS	Returns the number of seconds elapsed since midnight minus the number of seconds passed as an argument.
TIME	Returns a string in the form `hh:mm:ss`, such as 22:10:30.

Note

Functions RAN and SECNDS cannot be used with the $HP3000_16 ON directive. This directive causes the floating-point format to be classic HP 3000 instead of IEEE, and will not be recognized by these functions. The compiler attempts to find a compatibility mode routine for these which does not exist.

For information on these intrinsics, see "FORTRAN Intrinsic Functions and Subroutines" in Appendix B.

INTRINSICS Option

This directive allows the 9000 Series 800 to return an INTEGER*2 when $SHORT is enabled, but an INTEGER*4 when $SHORT is not enabled (like on the 9000 Series 300 and other vendors' FORTRAN compilers).

- INT
- IFIX
- IDINT
- IQINT
- IDNINT
- IQNINT
- MAX1
- MIN1
- ZEXT

OPEN Option

The OPEN option allows multiple OPENs of the same file with different unit numbers. By default, multiple OPENs of the same file cause a run time error.

```
$NOSTANDARD OPEN ON
        program main

C       connect DataFile to unit 10
        OPEN(10,FILE='DataFile')
C       connect DataFile to unit 20 for reading only
C       Note:  without NOSTANDARD OPEN ON, this would
C              cause an error at runtime
        OPEN(20,FILE='DataFile',READONLY)

        STOP
        END
```

ONETRIP Directive

The ONETRIP directive turns on or off the requirement that the body of each DO loop (other than DO WHILE loops) is executed at least once, in compliance with the previous ANSI 66 standard.

Syntax

```
          ┌ ON  ┐
$ONETRIP  │     │
          └ OFF ┘
```

Default	Off; individual DO loop bodies are not required to execute at least once.
Location	The ONETRIP directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

OPTIMIZE Directive

The OPTIMIZE directive sets up optimizer options that can improve performance.

Syntax

```
              ┌ LEVEL1                                          ┐
              │ LEVEL2                                          │
              │ LEVEL2_MIN                                      │
              │ LEVEL2_MAX                                      │
              │ ASSUME_NO_EXTERNAL_PARMS                        │
  $OPTIMIZE   │ ASSUME_NO_FLOATING_INVARIANT                    │
              │ ASSUME_NO_PARAMETER_OVERLAPS                    │
              │ ASSUME_NO_SHARED_COMMON_PARMS                   │
              │ ASSUME_NO_SIDE_EFFECTS                          │
              │ ASSUME_PARM_TYPES_MATCHED                       │
              └ LOOP_UNROLL[COPIES=n SIZE=n STATISTICS]         ┘
  ┌ ON  ┐
  └ OFF ┘
```

`ON`	Alone, specifies level 2 optimization. With a preceding option, sets that option on.
`OFF`	Alone, specifies level 0 optimization. This is the default. With a preceding option, sets that option off.
`LEVEL1`	Specifies level 1 optimization.
`LEVEL2`	Specifies level 2 optimization, with the following `ASSUME` settings:

ASSUME_NO_EXTERNAL_PARMS	ON
ASSUME_NO_FLOATING_INVARIANT	ON
ASSUME_NO_PARAMETER_OVERLAPS	ON
ASSUME_NO_SHARED_COMMON_PARMS	ON
ASSUME_NO_SIDE EFFECTS	OFF
ASSUME_PARM_TYPES_MATCHED	ON
LOOP_UNROLL	ON

`LEVEL2_MIN`	Specifies level 2 optimization with all the `ASSUME` settings `OFF`.
`LEVEL2_MAX`	Specifies level 2 optimization with all the `ASSUME` settings `ON`.

`ASSUME_NO_EXTERNAL_PARMS`	Assumes that none of the parameters passed to the current procedure are from an external space, that is, different from the user's own data space. Parameters can come from another space if they come from operating system space or if they are in a space shared by other users.
`ASSUME_NO_FLOATING_INVARIANT`	Assumes that no floating invariant operations are executed conditionally with loops.
`ASSUME_NO_PARAMETER_OVERLAPS`	Assumes that no actual parameters passed to a procedure overlap each other.
`ASSUME_NO_SHARED_COMMON_PARMS`	This directive should be `ON` when all of the following are true: ■ The parameter passed to the current procedure is part of a common block used by that procedure. ■ The parameter is named differently than the variable name it has in the common block. ■ The parameter is reassigned with the same value within the procedure.
`ASSUME_NO_SIDE_EFFECTS`	Assumes that the current procedure changes only local variables. It does not change any variables in COMMON, nor does it change parameters.
`ASSUME_PARM_TYPES_MATCHED`	Assumes that all of the actual parameters passed were the type expected by this subroutine.
`LOOP_UNROLL`	Unrolls DO loops having 60 or less operations four times. For further details, see "Loop Unrolling" in this chapter. The default is ON.

There are five levels of optimization:

Level 0 Does no optimizing. This is obtained by specifying `$OPTIMIZE OFF`.

Level 1	Optimizes only within each basic block. This is obtained by specifying `$OPTIMIZE LEVEL1 ON`.
Level 2 minimum	Optimizes within each procedure with no assumptions on interactions of procedures. That is, the compiler assumes nothing, making this the most conservative level 2 optimization. This level is obtained by specifying `$OPTIMIZE LEVEL2_MIN ON` within each procedure.
Level 2 normal	Optimizes within each procedure with normal assumptions on interactions of procedures set as described earlier. In general, these settings are appropriate for most FORTRAN programs. This level is obtained by specifying `$OPTIMIZE LEVEL2 ON`, `$OPTIMIZE ON` or just `$OPTIMIZE` within each procedure.
Level 2 maximum	Optimizes within each procedure with all assumptions on interactions of procedures set to OFF. This is obtained by specifying `$OPTIMIZE LEVEL2_MAX ON` within each procedure.

A basic block is a set of instructions to be executed in sequence, with one entrance, the first instruction, and one exit, the last; the block contains no branches.

Parameters can come from another space if they come from the operating system or if they are in a space shared by other users.

The following options are meaningful only when the compiler is performing level 2 optimization, that is, only if the option `ON`, `LEVEL2`, `LEVEL2_MIN`, or `LEVEL2_MAX` has been specified:

```
ASSUME_NO_PARAMETER_OVERLAPS
ASSUME_NO_SIDE_EFFECTS
ASSUME_PARM_TYPES_MATCHED
ASSUME_NO_EXTERNAL_PARMS
ASSUME_NO_SHARED_COMMON_PARMS
ASSUME_NO_FLOATING_INVARIANT
LOOP_UNROLL
```

Default Off.

Location The following OPTIMIZE options must appear before any nondirective statements in the program unit:

```
OFF
ON
LEVEL1
LEVEL2
LEVEL2_MIN
LEVEL2_MAX
ASSUME_NO_PARAMETER_OVERLAPS
ASSUME_NO_EXTERNAL_PARMS
ASSUME_NO_SHARED_COMMON_PARMS
ASSUME_NO_FLOATING_INVARIANT
```

These options can appear anywhere within a program unit:

```
ASSUME_NO_SIDE_EFFECTS
ASSUME_PARM_TYPES_MATCHED
LOOP_UNROLL
```

Toggling/ Duration The optimize options remain in effect until they are changed by another OPTIMIZE directive.

Impact on Performance This directive can improve performance. Loop unrolling, which usually improves performance, can occasionally degrade performance because of large loops (register spilling) and code expansion (crossing the page boundary causing cache misses and TLB misses.)

Flagging Uninitialized Variables

When the compiler is performing level 2 optimization, it will detect any uninitialized non-static simple local variables. However, it will not detect uninitialized common variables, static variables, or variables of character and complex type. For example:

```
$OPTIMIZE
    FUNCTION func(type)
    COMMON /a/comvar
    SAVE statvar
    REAL foo,type
    type = 10.2
    foo = comvar
    foo = statvar
    foo = typo
    RETURN
    END
```

The variable `typo` is flagged as an uninitialized variable because it was typed incorrectly and, therefore, not initialized. However, `statvar` and `comvar` are not flagged because of their global and static characteristics. A warning message will be issued when an uninitialized variable is detected.

Example

```
C     Start with minimum level 2 optimization.
$OPTIMIZE LEVEL2_MIN

      PROGRAM FEQ7
      INTEGER num(10), ans, calculate
      CHARACTER*2 option(10)
C
C     For the next two calls, the parameter type declarations are the same in
C     the main program and the subroutine or function.  Therefore, we can
C     further optimize the program by setting the following optimizer option.
$OPTIMIZE ASSUME_PARM_TYPES_MATCHED ON
      call getnum_option(num,option)
C
C     For the next call, the function will not change the parameter value or
C     any global variables in COMMON blocks.  Therefore, we can further
C     optimize the program by setting the following optimizer option.
$OPTIMIZE ASSUME_NO_SIDE_EFFECTS ON
      ans= calculate(num,option)
$OPTIMIZE ASSUME_NO_SIDE_EFFECTS OFF
      WRITE(6,*) 'Result = ',ans
      END
C
C     For the next subroutine, we know that the actual parameters passed to
C     this subroutine are not overlapped with each other, from a shared
C     common block, nor from another space different from the user's own
C     program, thus we can further optimize the program by setting the
C     following optimizer options.
$OPTIMIZE ASSUME_NO_PARAMETER_OVERLAPS ON
$OPTIMIZE ASSUME_NO_EXTERNAL PARMS ON
$OPTIMIZE ASSUME_NO_SHARED_COMMON PARMS ON
      SUBROUTINE getnum_option(value,operation)
      INTEGER value(10)
      CHARACTER*2 operation(10)

      DO 10  i = 1,10
20    WRITE(6,*) 'Please input operation type and integer value :'
      READ(5,*) operation(i),value(i)

      IF (operation(i).EQ.' ') GOTO 30

      IF ((operation(i).NE.'**').AND.
     /    (operation(i).NE.'*' ).AND.
     /    (operation(i).NE.'/' ).AND.
     /    (operation(i).NE.'-' ).AND.
     /    (operation(i).NE.'+' )) GOTO 20
10    CONTINUE
30    RETURN
      END
C
```

```
C     For the next subroutine, we know that the actual parameters passed to
C     this subroutine are not overlapped with each other, not from
C     external space, nor from a shared common block.  We can thus leave the
C     ASSUME_NO_PARAMETER_OVERLAPS, ASSUME_NO_EXTERNAL_PARMS, and
C     ASSUME_NO_SHARED_COMMON_PARMS settings ON.
C
      FUNCTION calculate(value,operation)
      INTEGER value(10),calculate,ans
      CHARACTER*2 operation(10)

      ans = 0
      DO 10  i = 1,10

      IF (operation(i).EQ.' ') GOTO 30

      IF (operation(i).EQ.'**') THEN
           ans = ans ** value(i)
      ELSE IF (operation(i).EQ.'*' ) THEN
           ans = ans * value(i)
      ELSE IF (operation(i).EQ.'/' ) THEN
           ans = ans / value(i)
      ELSE IF (operation(i).EQ.'-' ) THEN
           ans = ans - value(i)
      ELSE IF (operation(i).EQ.'+' ) THEN
           ans = ans + value(i)
      ENDIF
10    CONTINUE
30    calculate = ans
      RETURN
      END
```

Loop Unrolling

```
                              ┌ON         ┐
                              │OFF        │
$OPTIMIZE LOOP_UNROLL         │COPIES = n │
                              │,SIZE = n  │
                              └STATISTICS ┘
```

ON	Turns on loop unrolling. ON is the default at level 2.
OFF	Turns off loop unrolling.
COPIES = *n*	Tells the compiler to unroll the loop *n* times. The default is four times.
SIZE = *n*	Tells the compiler to unroll the loops that have less than *n* operations. The default is 60 operations.
STATISTICS	Tells the compiler to give statistics about the unrolled loops.

Limits on Use

DO loops at level 2 are unrolled four times by default. If the loop limit is either not known at compile time or is less than four times, an extra copy of the DO loop body is generated. This is called unrolling the loop four or more times.

Although loop unrolling optimization usually increases performance, it can occasionally degrade performance because of large loops (register spilling) and code expansion (crossing the page boundary causing cache misses and TLB misses.) When you encounter these circumstances, you can turn off loop unrolling locally by using the compiler directive. Use the compiler directive $OPTIMIZE to specify optimization level in the source and for changing the assumptions made by the compiler. You can use a suboption LOOP_UNROLL to control some constraints:

```
$OPTIMIZE LOOP_UNROLL
```

You can also use the LOOP_UNROLL suboption on the $OPTIMIZE directive to change the DO LOOP constraints for unrolling dynamically:

- You can unroll a DO loop more than four times.
- You can force a DO loop to unroll despite its large size.
- You can find the reason why a DO loop is not unrolled.

The highest level of optimization must be on for LOOP_UNROLL to work. Otherwise, LOOP_UNROLL is ignored. If LOOP_UNROLL is ignored, but STATISTICS has been specified, you will still get the DO loop statistics.

Note

The number of operations reported by STATISTICS is approximate. Each assignment, arithmetic operation, and logical operation counts as an operation. Each subscript of a subscripted variable counts as a separate operation.

To unroll the loop two times instead of four times (which is the default), use

```
$OPTIMIZE LOOP_UNROLL COPIES=2
```

To unroll a DO loop that is larger than the default, use

```
$OPTIMIZE LOOP_UNROLL COPIES=2, SIZE=500
```

substituting an appropriate size for the digit 500.

Example

```
C Example to illustrate the use of LOOP_UNROLL
$OPTIMIZE ON
           PROGRAM UNROLL_EXAMPLE

           DIMENSION A(10), B(10,10)
           DIMENSION X(10,10,10), Y(10,10,10), Z(10,10,10)
           . .
           . .
C The inner loop has only one statement.  The loop can be unrolled
C 10 times avoiding a branch and an extra copy of the loop.  A straight
C line code is generated for the inner loop.

$OPTIMIZE LOOP_UNROLL COPIES=10

           DO 20 J=1,10
           DO 10 I=1,10
             A(I) = A(I) + B(I,J)
     10    CONTINUE
     20    CONTINUE

C  Change COPIES back to default.
$OPTIMIZE LOOP_UNROLL COPIES=4
           . .
           . .
C This DO loop has more than 60 operations.
C This does not get unrolled by default.  The LOOP_UNROLL option is used
C to unroll it two times by increasing the SIZE to a large value.

$OPTIMIZE LOOP_UNROLL COPIES=2, SIZE=200

          DO 40 I=1,10
          DO 30 J=1,20
            V1 = X(I,J+1,K) - X(I,J-1,K)
            V2 = Y(I,J+1,K) - Y(I,J-1,K)
            V3 = Z(I,J+1,K) - Z(I,J-1,K)
            X(I,J,K) = X(I,J,K) + A11*V1 + A2*V2 +
     *       A3*V3 + S*(Y(I+1,J,K)-2.0*X(I,J,K)+X(I-1,J,K))

            Y(I,J,K) = Y(I,J,K) + A1*V1 + A2*V2 +
     *       A3*V3 + S*(Y(I+1,J,K)-2.0*Y(I,J,K)+Y(I-1,J,K))
```

```
            Z(I,J,K) = Z(I,J,K) + A1*V1 + A2*V2 +
     *       A3*V3 + S*(Z(I+1,J,K)-2.0*Z(I,J,K)+Z(I-1,J,K))
   30          CONTINUE
   40    CONTINUE

C Change the options back to the default values.
$OPTIMIZE LOOP_UNROLL COPIES=4, SIZE=60
          . .
          . .
          STOP
          END
```

PAGE Directive

The PAGE directive sends a form feed to the list file or device, which causes a skip to a new page before continuing with the program listing.

Syntax

```
$PAGE
```

Default	None.
Location	The PAGE directive can appear anywhere within the program unit.
Toggling/ Duration	Cannot be toggled.

Page Eject with Control-L

A control-L (ASCII 12) in column 1 of any source line has the same effect as if the line were preceded by the PAGE directive; the compiler removes the control-L from the listing.

If a control-L is found anywhere else on a source line, it is treated like a blank and remains in the program listing. Its presence may affect an output device that displays the listing file.

PAGEWIDTH Directive

The PAGEWIDTH directive allows you to specify the length of output lines in the listing file.

Syntax

```
   $PAGEWIDTH n
```

n	is an integer constant from 79 to 150.
Default	80. Values outside of this range will cause a warning and the value will be ignored. Output lines longer than *n* are broken into multiple lines as necessary so that no line has more than *n* columns of data on the listing file. The value of *n* does not include any appended newline characters used to break the line.
Location	May occur anywhere within a program.
Toggling/ Duration	Applies until another PAGEWIDTH directive is encountered.

POSTPEND Directive

The POSTPEND Directive allows C programmers to access FORTRAN routines and data as per BSD programming standards.

syntax

$$\text{\$POSTPEND} \begin{bmatrix} \text{ON} \\ \text{OFF} \end{bmatrix}$$

Default	Off.
Location	This directive must appear before any nondirective statement in a program unit, including the program head.

Other Information

The $POSTPEND directive postpends an underbar to the end of the names of references to user declared routines, declarations of user routines and references and declarations of user declared COMMON blocks.

External names defined by the ALIAS or EXTERNAL_ALIAS directive are not affected by the POSTPEND directive and will not have an underbar postpended to them even if the POSTPEND directive is on.

In the following example, a FORTRAN 77 subprogram file has the POSTPEND directive on and declares a function called *ftnsub* which adds two numbers and returns the result. A C program file calls the FORTRAN 77 subroutine *ftnsub* by referring to `ftnsub`.

Example

FORTRAN 77 File:

```
$POSTPEND ON
   SUBROUTINE ftnsub()
   PRINT *, "In FORTRAN routine ftnsub."
   RETURN
   END
```

C File:

```
main()
{
 ftnsub_(); /* The call to ftnsub_ is resolved to the FORTRAN 77 routine ftnsub.*/
}
```

RANGE Directive

The RANGE directive turns on or off compile-time bounds checking for:

- subscript and substring expressions
- bit-manipulation intrinsic functions
- DO loop increment counts that are not equal to zero
- assigned GOTOs

Range checking is not performed on the final dimension of assumed-sized arrays.

Syntax

$$\texttt{\$RANGE} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; no compile-time bounds checking for subscript and substring expressions, bit-manipulation intrinsic functions, DO loop increment counts that are not equal to zero, and assigned GOTOs is done.
Location	The RANGE directive can appear anywhere within the program unit.
Toggling/ Duration	Applies until another RANGE directive is encountered.

RLFILE Directive

This directive causes each program unit to be compiled into its own object module.

Syntax

```
$RLFILE
```

Default	None; the compiler creates a relocatable object file containing one module for all procedures.
Location	The RLFILE directive must occur before any nondirective statement in the program unit, including the program head.
Impact on Performance	This directive results in less efficient use of library space than compiling each procedure individually from separate source files because, for each procedure in the source program file, a module is either added or updated (if it already exists) in a relocatable library. (Refer to the *HP Link Editor/iX Reference Manual* for more information on relocatable libraries.) This directive puts additional information into the object file (of type NMRL), thus significantly increasing the object file size.

Additional Information

Each program unit (identified by a PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement) is compiled into its own object module and placed into a file of type NMRL. The default file name is $OLDPASS. In subsequent compilations into the same RL file, program units will replace corresponding object modules of the same name. This functionality allows you to add and delete entries on the program unit level (versus module level) using the Link Editor.

This directive provides the same functionality available with FORTRAN 77/V when doing multiple compiles into the same USL file.

RLINIT Directive

This directive causes the compiler to initialize the RL file to empty (thus deleting all object modules) before placing any object code into it. This occurs before compilation begins. This directive provides the same functionality available with USLINIT in FORTRAN 77/V.

Syntax

```
$RLINIT
```

Default None. If the RLINIT directive is not used and an RL file is the compilation target, all object modules with entry points duplicated in the current compilation unit are replaced; other object modules are then left intact.

If the RLINIT directive is used without the RLFILE directive, all program units in the current compilation are compiled into a single object module. If the specified target does not exist, a file of type NMRL is created.

Location The RLINIT directive must occur before any nondirective statements in the program.

SAVE_LOCALS Directive

The SAVE_LOCALS directive automatically saves any local variables encountered. This is the same as specifying SAVE in each subprogram of each source file. This directive forces static storage for all local variables in order to provide a convenient path for importing FORTRAN 66 and 77 programs that were written to depend on static allocation of memory (that is, variables retaining their values between invocations of the respective program units).

Syntax

$$\texttt{\$SAVE\_LOCALS} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; local variables are not automatically saved.
Location	The SAVE_LOCALS directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in the program unit.

SEGMENT Directive

This directive, included for compatibility with programs in earlier versions of FORTRAN, is the same as the LOCALITY directive. The SEGMENT directive produces the warning “SEGMENT has been mapped to LOCALITY on this operating system.”

SET Directive

The SET directive assigns values to identifiers used in IF directives.

Syntax

$$\texttt{\$SET (}\ \mathit{flag1}\ \texttt{=} \left\{\begin{array}{l}\texttt{.TRUE.}\\ \texttt{.FALSE.}\end{array}\right\} \left[\ \texttt{,}\ \mathit{flag2}\ \texttt{=} \left\{\begin{array}{l}\texttt{.TRUE.}\\ \texttt{.FALSE.}\end{array}\right\}\right] [\ldots]\texttt{)}$$

flag	is one or more identifiers given logical constant values. The identifiers in SET and IF compiler directives are in no way related to FORTRAN variables in the source text. That is, if the same variable name is used both as an identifier for one of these directives and elsewhere within a program, the one has no effect upon the other.
Default	None.
Location	The SET directive can appear anywhere within a program unit.
Toggling/ Duration	The identifier retains its value until changed by another SET directive.

Examples

```
$SET (TOGGLE=.TRUE.,DEBUG=.FALSE.)
$SET (SYSTEM1=.TRUE.)
```

SHORT Directive

The SHORT directive sets the default size for integer and logical data types and constants to two bytes. The INTEGER and LOGICAL type names are set equivalent to INTEGER*2 and LOGICAL*2, respectively.

Syntax

```
$SHORT [INTEGERS]
```

The keyword INTEGERS in the directive is optional and has no effect.

Default	Long (4 bytes) if neither the LONG nor the SHORT directive is given.
Location	The SHORT directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in the program unit.

STANDARD_LEVEL Directive

The STANDARD_LEVEL directive sets the level of syntax that the compiler processes routinely.

Syntax

$$\texttt{\$STANDARD\_LEVEL} \left\{ \begin{array}{l} \texttt{ANSI} \\ \texttt{HP} \\ \texttt{SYSTEM} \end{array} \right\}$$

If the compiler encounters a FORTRAN language feature not legal at the specified level, it issues a warning message on the listing and then compiles the feature normally.

ANSI	refers to the ANSI 77 FORTRAN standard (ANSI X3.9-1978). Specifying the ANSI level is semantically equivalent to specifying `$ANSI ON`. This level has the fewest language features of the three. Warnings are given for any non-ANSI features.
HP	the default, indicates Hewlett-Packard Standard FORTRAN (FORTRAN 77). This level allows more language features than the ANSI level and includes the MIL-STD 1753 extensions. Warnings are given for system-specific features only.
SYSTEM	indicates FORTRAN 77 plus additional system dependent features added to the language. This level has the most language features of the three. No warnings are given for nonstandard features.

Default	HP; allows MIL-STD 1753 extensions.
Location	STANDARD_LEVEL must appear before any nondirective statement in a program unit, including the program head.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

Example

```
$STANDARD_LEVEL HP
```

SUBTITLE Directive

The SUBTITLE directive lists the subtitle string on the second line of each page of the program listing following the appearance of the directive in the source code.

Syntax

$$\texttt{\$SUBTITLE} \left\{ \begin{array}{l} \text{'}\,\textit{subtitle\_string}\text{'} \\ \text{"}\textit{subtitle\_string}\text{"} \end{array} \right\}$$

Default None; no subtitle string is listed.

If the subtitle string is longer than 72 characters, it is truncated to 72.

Location The SUBTITLE directive can appear anywhere within the program unit.

Toggling/ Duration Applies until another SUBTITLE directive is encountered.

Example

```
$SUBTITLE 'Assigned GOTO''s or, The Joy of FORTRAN'
```

SYMDEBUG Directive

The SYMDEBUG directive causes the compiler to include in the object file the information needed by a symbolic debugger.

Syntax

$$\texttt{\$SYMDEBUG} \begin{bmatrix}\texttt{XDB}\\ \texttt{TOOLSET}\end{bmatrix} \begin{bmatrix}\texttt{ON}\\ \texttt{OFF}\end{bmatrix}$$

`$SYMDEBUG ON`	(or `$SYMDEBUG` or `$SYMDEBUG TOOLSET`) the compiler places debug information into the object file for HP Toolset/iX to use.
`$SYMDEBUG XDB ON`	(or `$SYMDEBUG XDB`) the compiler places xdb-specific information into the object file for xdb to use.

Default	Off; no symbolic debugger information is included in the object file.
Location	The SYMDEBUG directive must appear before any nondirective statements in the program unit, including the program head.
Toggling/ Duration	Symbolic debugging information continues to be generated for all program units until `$SYMDEBUG OFF` is encountered.

Warning

The SYMDEBUG directive cannot be used on optimized code. If the SYMDEBUG directive is used with the OPTIMIZE directive, a warning is issued and the OPTIMIZE directive is ignored.

SYMTABLE Directive

The SYMTABLE directive, included for compatibility with programs in earlier versions of FORTRAN, is the same as the TABLES directive. The symbol table information is printed even if an error occurs at compile time.

Syntax

```
$SYMTABLE [ON ]
          [OFF]
```

Default	Off.
Location	The SYMTABLE directive must precede any nondirective statements in a program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

Example

```
     0    1       $SYMTABLE ON
     1    2              PROGRAM TEST
     2    3              INTEGER I,J(20)
     3    4              CHARACTER*30 NAME
     4    5              COMMON /COM1/ I
     5    6              NAME = 'JOE SMITH'
     6    7              DO 100 ,I=1,20
     7    8      1       J(I) = I
     8    9      1 100   CONTINUE
     9   10              CALL ROUTINE1
    10   11              END
     0   12

Name            Class           Type            Offset          Location
----            -----           ----            ------          --------

/COM/           Common
I               Variable        Integer*4       COM+0           /COM1/
J               Array (1 Dim)   Integer*4       SP -160         Local
NAME            Variable        Character*30    SP -80          Local
routine1        Subroutine
test            Program
100             Stmt Label      Executable                          8


     1   13              SUBROUTINE ROUTINE1
     2   14              WRITE(6,*) 'END OF TEST'
     3   15              END
```

```
Name            Class           Type           Offset         Location
----            -----           ----           ------         --------

routine1        Subroutine

   NUMBER OF ERRORS =     0   NUMBER OF WARNINGS =      0
```

SYSINTR Directive

The SYSINTR directive permits you to specify a Pascal intrinsic file to be searched for a subroutine or function declared with the SYSTEM INTRINSIC directive.

Syntax

$$\texttt{\$SYSINTR} \left\{ \begin{array}{l} \textit{'sysintr\_filename'} \\ \textit{"sysintr\_filename"} \end{array} \right\}$$

sysintr_filename	is the name of a Pascal intrinsic file.
Default	SYSINTR.PUB.SYS will be the Pascal intrinsic file searched.
Location	The SYSINTR directive can appear anywhere in a program.
Toggling/ Duration	A file specified in a SYSINTR directive remains in effect until SYSINTR appears again. The SYSINTR directive is not in effect until specified, and it remains in effect until another SYSINTR directive appears.

Additional Information

You can provide a complete pathname for *sysintr_filename*.

The MPE/iX file SYSINTR.PUB.SYS contains information about the attributes of subprograms. These subprograms are usually user-callable system subprograms, such as FOPEN. All intrinsics mentioned in the MPE/iX manuals must be accessed through this facility. The information about a particular subprogram includes such items as the number and type of parameters, whether parameters are called by *value*, *reference*, ANYVAR, UNCHECKABLE_ANYVAR, or READONLY, and whether the subprogram has DEFAULT_PARMS parameters, EXTENSIBLE parameters, or both. See the section "SYSTEM INTRINSIC Statement (Nonexecutable)" found in Chapter 3 of this reference manual for an explanation of the preceding terms. FORTRAN reads the SYSINTR file for specially designated subprograms and generates the indicated code sequences.

System intrinsic files can be created with Pascal by using Pascal's BUILDINT compiler directive, described in the *HP Pascal Programmer's Guide*.

SYSTEM INTRINSIC Directive

The SYSTEM INTRINSIC directive functions exactly the same as the SYSTEM INTRINSIC statement.

Syntax

`$SYSTEM INTRINSIC` *intrinsic_name* [,*intrinsic_name*] [,. . .]

intrinsic_name	is the name of a system intrinsic.

Default	None.
Location	The SYSTEM INTRINSIC directive must appear before the first nondirective statement of the program unit in which it is to start taking effect. The SYSTEM INTRINSIC directive must be the only directive on a line.
Toggling/ Duration	Information associated with the SYSTEM INTRINSIC *directive* is retained across all program units in the file following the specification, while the SYSTEM INTRINSIC *statement* is in effect only for the program unit in which it is declared. See System Intrinsic Statement.

TABLES Directive

The TABLES directive turns on or off the symbol table information in the listing file. The symbol table information is printed even if an error occurs at compile time.

Syntax

```
         [ON ]
$TABLES  [OFF]
```

Default	Off; no symbol table information is included in the listing file.
Location	The TABLES directive must appear before any nondirective statements in the program unit.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in a program unit.

Example

```
     0    1       $TABLES ON
     1    2              PROGRAM TEST
     2    3              INTEGER I,J(20)
     3    4              CHARACTER*30 NAME
     4    5              COMMON /COM1/ I
     5    6              NAME = 'JOE SMITH'
     6    7              DO 100 ,I=1,20
     7    8      1       J(I) = I
     8    9      1 100   CONTINUE
     9   10              CALL ROUTINE1
    10   11              END
     0   12

Name            Class           Type            Offset          Location
----            -----           ----            ------          --------

/COM/           Common
I               Variable        Integer*4       COM+0           /COM1/
J               Array (1 Dim)   Integer*4       SP -160         Local
NAME            Variable        Character*30    SP -80          Local
routine1        Subroutine
test            Program
100             Stmt Label      Executable                          8

     1   13              SUBROUTINE ROUTINE1
     2   14              WRITE(6,*) 'END OF TEST'
     3   15              END
```

```
Name            Class           Type           Offset         Location
----            -----           ----           ------         --------

routine1        Subroutine

   NUMBER OF ERRORS =     0   NUMBER OF WARNINGS =      0
```

TITLE Directive

The TITLE directive lists the title string at the top of each page of output following the appearance of the directive in the source code.

If the title string is longer than 72 characters, it is truncated to 72.

Syntax

$$\texttt{\$TITLE} \left\{ \begin{array}{l} \text{'}\textit{title\_string}\text{'} \\ \text{"}\textit{title\_string}\text{"} \end{array} \right\}$$

Default	None; no title string is listed at the top of each page of output.
Location	The TITLE directive can appear anywhere within the program unit.
Toggling/ Duration	The TITLE directive remains in effect until another TITLE directive is encounters.

Example

```
$TITLE 'Optimization of CONTINUE statements'
```

UPPERCASE Directive

The UPPERCASE directive turns on or off shifting to upper case of all FORTRAN external names. This directive does not affect the external name of the ALIAS and EXTERNAL_ALIAS directives or intrinsic names if `$LITERAL_ALIAS ON` has been specified. See "ALIAS Directive" and "LITERAL"ALIAS Directive".

Specifying `$LOWERCASE ON` (or `$LOWERCASE`) is equivalent to specifying `$UPPERCASE OFF`.
Specifying `$LOWERCASE OFF` is equivalent to specifying `$UPPERCASE ON` (or `$UPPERCASE`).

Syntax

$$\texttt{\$UPPERCASE} \begin{bmatrix} \texttt{ON} \\ \texttt{OFF} \end{bmatrix}$$

Default	Off; external names are not shifted to upper case.
Location	The UPPERCASE directive can appear anywhere within the program unit.
Toggling/ Duration	The UPPERCASE directive remains in effect until either a LOWERCASE or another UPPERCASE directive is encountered.

If the UPPERCASE directive is toggled within a program unit, the point of declaration (or the point of first use if implicitly declared) determines the case of external names.

VERSION Directive

The VERSION directive inserts a string specified by the user into the auxiliary record header of the executable file, for purposes of version identification.

If the version string is longer than 72 characters, it is truncated to 72.

Syntax

$$\texttt{\$VERSION} \left\{ \begin{array}{l} \text{'}\mathit{version\_id}\text{'} \\ \text{"}\mathit{version\_id}\text{"} \end{array} \right\}$$

Default	None; nothing is inserted.
Location	The VERSION directive must appear before any nondirective statement in a program unit, including the program head.
Toggling/ Duration	The VERSION directive must be specified for each program unit within which it is to take effect.

WARNINGS Directive

The WARNINGS directive turns on or off the output of warnings.

Syntax

$$\text{\$WARNINGS}\begin{bmatrix}\texttt{ON}\\ \texttt{OFF}\end{bmatrix}$$

Default	On; warnings are output.
Location	The WARNINGS directive can appear anywhere within the program unit.
Toggling/ Duration	The WARNINGS directive remains in effect until another WARNINGS directive is encountered.

XREF Directive

The XREF directive produces a cross reference listing of a program unit. It is equivalent to the CROSSREF directive.

Syntax

```
         [ON ]
   $XREF [OFF]
```

Default	Off; no cross reference listing is produced.
Location	The XREF directive must appear before any nondirective statements in the program unit and it must precede an executable program unit like PROGRAM, FUNCTION, or SUBROUTINE.
Toggling/ Duration	Cannot be toggled after the appearance of nondirective statements in the program unit.

Example

The following is a sample program using the XREF directive.

```
$XREF
      INTEGER FUNCTION icp(op)
      COMMON /chars/ iblnk,ibkslsh,iequal,irparen,ilparen
     1 icomma,iperiod,iplus,iminus,isemi,idollar,ileta,iletz
C     ...
C     ... returns incoming priority of op
      INTEGER op
      INTEGER legal(34)
      data legal /2h( ,2h+ ,2h- ,2h* ,2h/ ,2h^ ,2h-1,2h-2,
     1            2h-3,2h-4,2h-5,2h-6,2h-7,2h-8,2h-9,2h10,
     2            2h11,2h12,2h13,2h14,2h15,2h16,2h17,2h18,
     3            2h19,2h20,2h21,2h22,2h23,2h24,2h25,2h26,
     4            2h27,2h) /
      icp=-1
      DO 20 i=1,34
      IF (op .EQ. legal(i)) GO TO 30
   20 CONTINUE
C     ...
C     ... illegal op to icp
      CALL eror(7)
      RETURN
C     ...
   30 GO TO (80,40,40,50,50,60,70,70,70,70,70,70,70,70,70,
     1       70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,
     2       70,70,70,80),i
   40 icp=1
      RETURN
   50 icp=2
      RETURN
   60 icp=4
```

```
31            RETURN
32         70 icp=5
33            RETURN
34         80 icp=100
35            RETURN
36            END
```

The following is the cross reference listing for the above program. The default line width is 80 columns. It can be changed with the PAGEWIDTH directive.

```
SYMBOL    TYPE    FILE   LINE
------    ----    ----   ----
CHARS/    (COMMN) xref.f $3
EROR      (PROC ) xref.f #20
I         (VAR  ) xref.f !15   16   25
IBKSLSH   (VAR  ) xref.f %3
IBLNK     (VAR  ) xref.f %3
ICOMMA    (VAR  ) xref.f %4
ICP       (PROC ) xref.f *2    !14  !26  !28  !30  !32  !34
IDOLLAR   (VAR  ) xref.f %4
IEQUAL    (VAR  ) xref.f %3
ILETA     (VAR  ) xref.f %4
ILETZ     (VAR  ) xref.f %4
ILPAREN   (VAR  ) xref.f %3
IMINUS    (VAR  ) xref.f %4
IPERIOD   (VAR  ) xref.f %4
IPLUS     (VAR  ) xref.f %4
IRPAREN   (VAR  ) xref.f %3
ISEMI     (VAR  ) xref.f %4
LEGAL     (VAR  ) xref.f *8    @9   16
OP        (ARGMT) xref.f %2    *7   16

   NUMBER OF ERRORS =     0   NUMBER OF WARNINGS =    0
```

`SYMBOL`	The symbol name.
`TYPE`	The class of the symbol: ■ ARGMT - argument passed to a procedure or function. ■ COMMN - name of a common block. ■ CONST - named constant in a PARAMETER statement. ■ NMLST - NAMELIST variable. ■ PROC - internal or external procedure or function name. ■ VAR - variables.
`FILE`	The file where the symbol is found.
`LINE`	One or more rows of line numbers that indicate where the symbol is found. Each line has a suffix that indicates the status of the symbol at time of access. ■ blank - symbol is being referenced. ■ `!` - symbol is being modified. ■ `*` - symbol is being defined and declared. ■ `%` - symbol is being declared. ■ `^` - symbol is being declared, defined, and modified. ■ `#` - symbol is being called. ■ `$` - symbol is declared, defined and used. ■ `@` - symbol is declared and modified.

8

Interfacing with Non-FORTRAN Subprograms

Any non-FORTRAN program unit can be used as a part of an executable FORTRAN program if the program unit has a calling sequence and method of execution compatible with FORTRAN 77. In addition, a FORTRAN subprogram can be used by programs written in other languages if the FORTRAN subprogram is compatible with the calling program's requirements. This chapter discusses issues to consider when interfacing FORTRAN programs to other languages.

Parameter Passing Methods

All arguments of a subprogram written in FORTRAN are passed by *reference*, except for character variables, which are passed by descriptor. This means that the addresses of the values are passed instead of the actual values of the arguments. Therefore, a FORTRAN subprogram expects a list of addresses for the formal arguments passed to it (for character variables, FORTRAN expects an address and a length), one for each argument and in the order given by the formal argument list contained within the subprogram.

Although values are normally passed by reference (indirectly), they can be passed by value (directly) to invoke non-FORTRAN program units that allow passing arguments by value. To accomplish this, use the ALIAS compiler directive to indicate how each of the parameters is to be passed. The language option of the ALIAS directive can also indicate the language of the routine being called so the compiler can pass arguments to the routine in the manner expected. For example, in Pascal, when a formal parameter is a PAC variable (PACKED ARRAY[1..*n*] OF CHAR), only a pointer to the variable is expected. However, FORTRAN 77 also passes the length of the character variable. If you indicate with the ALIAS directive that the routine being called is a Pascal routine, FORTRAN 77 will not pass the length of the character variable.

FORTRAN does not allow arrays to be passed by value. It is not possible to interface a FORTRAN program to a non-FORTRAN program unit that requires an array parameter to be passed by value.

Also, as in calling FORTRAN functions that have no parameters, an empty parameter list, ( ), must be given when referencing any non-FORTRAN function that has no parameters.

Use of COMMON and Labels

Non-FORTRAN program units cannot access a FORTRAN COMMON area and FORTRAN program units cannot access the global variables of non-FORTRAN programs. All data must be passed through the parameter lists. If a FORTRAN program calls a non-FORTRAN program unit, the FORTRAN program can contain COMMON areas, but the non-FORTRAN unit cannot use global variables. If a non-FORTRAN program calls a FORTRAN subprogram, the program can use global variables and the FORTRAN routine can use COMMON variables. The global variables and COMMON variables occupy distinct areas of memory.

Labels cannot be used as parameters when calling non-FORTRAN program units.

Files

A FORTRAN unit number cannot be passed to a non-FORTRAN subprogram to perform input/output on the associated file. Similarly, file variables of a non-FORTRAN program cannot be passed to a FORTRAN program to enable file access. However, a file can always be accessed by using system intrinsics. Support for such access is provided by the FSET and FNUM intrinsics, described in chapter 5, “File Handling”.

FORTRAN and C

This section describes how to interface with C.

Logicals

C has no logical type; it uses integers instead. A FORTRAN LOGICAL*2 is represented by a C short integer, and a LOGICAL*4 by a C long integer. FORTRAN and C do not share a common definition of true and false. In C, zero is false, and any nonzero value is true.

Arrays

FORTRAN stores arrays in column-major order, while C stores them in row-major order.

Files

A FORTRAN unit cannot be passed to a C routine to perform I/O on the associated file. Nor can a C file pointer be used by a FORTRAN routine. However, a file created by a program written in either language can be used by a program of the other language if the file is declared and opened within the latter program.

C accesses files using its own I/O subroutines and intrinsics. This method of file access can also be used from FORTRAN instead of FORTRAN I/O. Be aware that HP FORTRAN 77 on HP 3000 Series 900 MPE/iX uses the unbuffered I/O system calls `read` and `write` (described in the *MPE/iX Reference* manual) for terminal I/O, magnetic tape I/O, and direct access I/O.

Parameter Passing Methods

FORTRAN passes noncharacter parameters by reference, while FORTRAN character strings are "passed by descriptor." The descriptors are system-defined and are described under "Character" later in this section. Therefore all actual parameters in a C call to a FORTRAN routine must be pointers or variables prefixed with the address operator (&), and all formal parameters in a C routine called from FORTRAN must be pointer variables, unless a FORTRAN ALIAS directive defines them as value parameters. FORTRAN character data passed as parameters or to a C routine can be handled in a special manner by specifying the `C` option to the ALIAS directive, or by using the ALIAS directive with the optional parameter information list, as described under "ALIAS Directive," later in this chapter. Alternately, a structure can be defined corresponding to the FORTRAN character descriptor.

If a FORTRAN program is receiving a parameter from a C program, the parameter is by reference if it is an array; otherwise, the parameter is passed by value if it is less than or equal to 64 bits, or by reference if it is greater.

If a FORTRAN program is passing a parameter to a C program, and the C program declares it to be of type array, the C program expects the parameter to be passed by reference. If the C program declares it to be a structure or union greater than 64 bits, the C program copies the parameter into a temporary memory location. The parameter-passing mechanism is then by reference, but the effect is as if by value, because the value cannot be changed. If the parameter is less than or equal to 64 bits, it is passed by value.

This parameter passing is handled correctly by specifying `C` in the language option of the ALIAS directive. (See "ALIAS Directive" earlier in this chapter.)

Complex Numbers

C has no complex numbers. However, a COMPLEX*8 number can be represented in C by the following structure:

```
struct complex {
            float real_part, imaginary_part;
          }
```

Similarly, a FORTRAN COMPLEX*16 number can be represented by the same structure with the real and imaginary parts being of C type double.

Character

When FORTRAN passes character parameters, it passes them by descriptor. The descriptor includes two items: a pointer to the first character in the string and an integer value for the declared length of the string. When passing FORTRAN character strings to C subprograms, you must be sure to accommodate the length descriptor. Use the default character passing method to accomodate the length descriptor when passing a character string to a C subprogram. Note that parameters from FORTRAN are all passed by reference, except for the character length descriptors, which are passed by value.

Default Character Passing Method

When you use the default method of character passing, an integer length descriptor is passed after each character parameter. Therefore, you need to provide two variables in the parameter list of the C subprogram for each FORTRAN character variable passed. For example, if FORTRAN passes the C subprogram two parameters (two strings and one integer), the C subprogram must be able to accept five parameters (two strings and three integers). This is illustrated in the following example:

FORTRAN code:

```
INTEGER*4 num
CHARACTER*10 str1,str2

CALL testproc (str1,str2,num)
```

C code:

```
testproc (str1,strlen1,str2,strlen2,num)
int*num;
char*str1;
char*str2;
int strlen1;
int strlen2;
```

Hollerith

The FORTRAN Hollerith data type is similar to the C char array.

FORTRAN and Pascal Data Types

When a FORTRAN program interfaces with a Pascal program unit, be aware of the corresponding data types shown in table 8-1. In particular, note the differences between character strings and Boolean variables between the two languages.

Table 8-1. HP FORTRAN 77 and HP Pascal Data Types

HP FORTRAN 77 Type	HP Pascal Type
INTEGER*4	INTEGER *or* integer subrange beyond the range 0 .. 65535
INTEGER*2	SHORTINT *or* integer subrange inside the range 0 .. 65535
REAL*4	REAL
REAL*8	LONGREAL
BYTE, LOGICAL*1	Integer subrange inside the range 0..255
CHARACTER	CHAR
CHARACTER**n*	PACKED ARRAY [1..*n*] OF CHAR
LOGICAL*4	INTEGER *or* SET (4 bytes)
LOGICAL*2	Integer subrange inside the range 0..65535 *or* SET (2 bytes)

COMPLEX*8	RECORD
	real_part : REAL;
	imag_part : REAL;
	END;

COMPLEX*16	RECORD
	real_part : LONGREAL;
	imag_part : LONGREAL;
	END;

REAL*16	No corresponding Pascal data type.

HP FORTRAN 77 has a one-word descriptor that describes the maximum length of the string while PACs (PACKED ARRAY[1..*n*] OF CHAR) in Pascal do not. Therefore, when you pass a character string to a Pascal string, Pascal expects a pointer to that string only.

Boolean variables also differ between the two languages. Pascal Boolean variables are one-byte variables, while FORTRAN logical variables are two or four bytes (LOGICAL*2 or LOGICAL*4).

Also note the following when a FORTRAN program interfaces with a Pascal program unit:

- HP FORTRAN 77 cannot pass arrays by value, so you cannot call a Pascal routine with a value parameter of a type corresponding to an HP FORTRAN 77 array type.
- All data must be passed through the parameter lists because HP FORTRAN 77 cannot specify global variables and Pascal cannot specify COMMON blocks.
- HP FORTRAN 77 expects parameters to be passed by reference, with the exception of the maximum length of a character string, as described earlier.
- Parameter type checking should be turned off because HP FORTRAN 77 generates different types of check values from Pascal.
- Files and labels cannot be passed between HP FORTRAN 77 and Pascal.

Condition Codes

Frequently, condition codes are returned to a FORTRAN 77 program by system intrinsics. These condition codes are listed in table 8-2. Specific meanings depend on individual intrinsics; refer to the *MPE/iX Intrinsics Reference Manual* for condition codes of specific intrinsics.

Table 8-2. Condition Codes

Condition Code	Meaning
CCE	Condition code is zero. This generally indicates that the request was granted.
CCG	Condition code is greater than zero. A special condition occurred but it might not have affected the execution of the request. (For example, the request was executed, but the default values were assumed as intrinsic call parameters.)
CCL	Condition code is less than zero. The request was not granted, but the error condition might be recoverable.

Beside condition codes, some intrinsics return additional error information to the calling program through their return values.

The condition code can be checked with the CCODE( ) function. CCODE returns an integer value indicating the condition code resulting from a call to a system intrinsic.

It is good practice to check the condition code using an arithmetic IF statement immediately following the intrinsic call, as shown in the following example. Similarly, the CCODE function must not appear in the INTRINSIC statement that is to be passed as an actual parameter to a subroutine that expects a procedure parameter because the condition code would be lost.

The following program checks the condition code after calling a system intrinsic.

```
      PROGRAM printop2
C
C EXAMPLE PROGRAM TO CALL SYSTEM INTRINSIC PRINTOP
C
      CHARACTER message*14
      LOGICAL lmessage (7)
      SYSTEM INTRINSIC printop
      EQUIVALENCE (lmessage, message)
      message='This is a test'
      CALL printop (lmessage, -14, 0)
      IF (CCODE()) 20, 10, 20
   10 STOP 'Successful Write'
   20 STOP 'Intrinsic returned bad condition code'
      END
```

Built-In Functions

Arguments of a FORTRAN subprogram are passed by reference, except for character variables, which are passed by descriptor. To call subprograms written in another language, you might have to pass arguments that are different from those used by FORTRAN. To handle this difference, HP FORTRAN 77/iX has built-in functions, as summarized below:

Intrinsic Function	**Description**	**Restrictions**
`%VAL(`*arg*`)`	Passes the argument as an immediate value. If the argument is shorter than 32 bits, it is sign-extended to a 32-bit value.	*arg* can be a constant, variable, array element, or an expression.
`%REF(`*arg*`)`	Passes the address of the value.	*arg* can be a numeric value, a character expression, an array, an array element, or a procedure name.
`%LOC(`*value*`)`	Returns the internal address of a storage element. The result is of type INTEGER*4. %LOC is equivalent to the FORTRAN intrinsic BADDRESS.	*value* can be a variable name, an array element name, an array name, a character substring name, or an external procedure name.

These functions are extensions to the ANSI standard.

To change the form of the argument, the built-in functions can be used in the argument list of a CALL statement or a function reference, as shown below.

Examples	**Notes**
`CALL routine(%VAL(`*a*`),b)`	Passes the argument *a* as an immediate value.
`CALL routine2(%REF(`*a*`))`	Passes the argument *a* by reference.

Note

The built-in functions %VAL or %REF can *only* be called in the actual argument list.

9

Managing Run-Time Errors and Exceptions

This chapter describes tools and methods for managing run-time errors and exceptions. Common run-time errors and exceptions include memory violations, floating-point exceptions, and input/output errors. The input/output error messages and recommended solutions are given in Appendix A.

Trapping Run-Time Errors

The MPE/iX implementation of FORTRAN 77 provides a trap handling mechanism that allows you to control how a program interruption is handled. An interruption may be handled by:

- Executing a specified procedure.
- Ignoring the interruption.
- Aborting the program.

The trap-handling mechanism is initiated with the ON statement.

Whenever a major error occurs during the execution of a program, of a hardware instruction, of a procedure from the System Library, or of a user-called intrinsic, your program normally aborts and an error message is printed. You can change this action by establishing traps for any of the following kinds of interruptions:

- Arithmetic errors.
- Basic external function errors.
- Internal function errors.
- Control-Y user interrupts.

This is a program that traps external errors:

```
PROGRAM TEST
REAL*8 A
ON EXTERNAL ERROR CALL ERRORHANDLE
A = 8.0
PRINT *, "01   DACOS(A) ", DACOS(A)
PRINT *, "Normal Exit from Main"
END

SUBROUTINE ERRORHANDLE(ERRORNUM, RESULT, OP1, OP2)
INTEGER*4 ERRORNUM
REAL*8 RESULT, OP1, OP2
PRINT *, "Control returned to SUBROUTINE ERROR"
PRINT *, "Internal Error occured ERROR NUMBER = ", ERRORNUM
PRINT *, "What error number to be passed to caller = "
READ *, ERRORNUM
PRINT *, "What result to be passed to caller  = "
READ *, RESULT
PRINT *, "oop1 = ", OP1
END
```

This program shows how an internal error can be trapped to the user-defined error recovery routine ERRORHANDLE.

```
*   ERRORNUM is the error that is generated in function DNUM.
*   RESULT is the result computed before the error occurs.  The
*          computation process is not yet complete.
*   OPERAND1 is the operand that is passed to DNUM.
*
*   ERRORNUM can be changed in the trap routine ERRORHANDLE.  When the error
*   number is set to zero, then a normal termination sequence occurs and
*   the standard error message prints.  When the error number is set to
*   a non-zero value, a user defined result can be passed back by the
*   trap routine.  The error number that is modified in the trap routine
*   won't modify the error generated in DNUM.
*
      PROGRAM TESTING
      REAL *8 A
      ON INTERNAL ERROR CALL ERRORHANDLE
100   A = DNUM('A')
      PRINT *,A
*     GENERATE ERROR 61 ** Number out of range **
      A = DNUM('12.000E+8934')
      PRINT *,A
      GOTO 100
      END

      SUBROUTINE ERRORHANDLE(ERRORNUM, RESULT, OPERAND1, NUMBER)
      INTEGER*4 ERRORNUM
      REAL*8 RESULT
      CHARACTER OPERAND1*(*)
      INTEGER*2 NUMBER
      PRINT *, "Control returned to SUBROUTINE ERROR"
      PRINT *, "Internal Error occured ERROR NUMBER = ", ERRORNUM
      PRINT *, "What error number to be passed to caller = "
      READ *, ERRORNUM
      PRINT *, "What result to be passed to caller  = "
      READ *, RESULT
      PRINT *, "Operand1 = ", OPERAND1
      PRINT *, NUMBER
      END
```

This program shows how a divide by zero can be trapped for `libf` math function FTN_DTOD(a,b).

```
*  If ERRORNUM = 0, the program will ABORT.  A non-zero value for
*  ERRORNUM causes the function to return the value of the result.
*
*      The result in the error recovery must match the parameter that was
*      passed.
*
*      This will cause Internal error 68.
*
       PROGRAM TESTING
       REAL *8 A, B, C
       INTEGER*4 I, J, K
       ON INTERNAL ERROR CALL ERRORHANDLE
       C = -2.0
       A = 0.0
       B = A**C
       PRINT *, "DtoD B= ", B
       PRINT *, "Main End."
       END

       SUBROUTINE ERRORHANDLE(ERRORNUM, RESULT)
       INTEGER*4 ERRORNUM
       REAL*8 RESULT
       PRINT *, "Control returned to SUBROUTINE ERROR"
       PRINT *, "Internal Error occured ERROR NUMBER = ", ERRORNUM
       PRINT *, "What error number to be passed to caller = "
       READ *, ERRORNUM
       PRINT *, "What result to be passed to caller  = "
       READ *, RESULT
       END
```

Refer to the *HP Compiler Library/iX Reference Manual* for more information about trapping errors.

Trap Actions

The action taken after an interrupt is trapped depends on the specification in the most recently executed ON statement for that interrupt condition.

- If ABORT was specified, a standard error message is generated and the program is aborted.
- If IGNORE was specified, processing continues with the next instruction.

  If the condition causing the interrupt is an integer division by zero, the result is set to zero. For other conditions, the previous content of the target register is supplied as the result. IGNORE is particularly valuable for preventing Control-Y interrupts at inconvenient times in a program.
- If CALL was specified, the normal (ABORT) error message is suppressed, and control is transferred to the specified trap procedure.

  Zero or more arguments describing the error are passed to the trap procedure, which can attempt to analyze or recover from the error, or can execute some other programming path specified by the user, such as an alternate return.

  Further details are given in the following sections.

Arithmetic Trap Procedure

For the ON INTEGER*2 OVERFLOW statement to be effective, the CHECK_OVERFLOW INTEGER compiler option must also be enabled. This is the default on MPE/iX.

If emulated floating point numbers have been selected by the HP3000_16 directive, any reference in the ON statement to the INEXACT or ILLEGAL trap is ignored, the traps are not set, and the compiler generates a warning message.

In each of the above cases, the corresponding trap requires one reference parameter that is of the same type as that associated with the error condition. When the trap is called, the parameter is the result of the operation that caused the trap to be invoked.

System Trap Procedure

The trap procedure that is called for system errors must have one parameter that is an integer array. The link editor performs parameter type checking. When the trap procedure is called, the contents of the parameter is an array of eight parameters defined by the system. For more details, refer to the *Trap Handling* manual. This parameter group immediately follows the parameters to the intrinsic in which the error occurred.

Basic External Function Trap Procedure

The trap procedure that is called for external function errors must have four formal arguments in the following order:

1. A single integer containing the error number that is determined by the external function in which the error occurred.
2. The result.
3. The first operand.
4. The second operand.

If the trap procedure returns normally and has set the error number (first argument) to zero, a standard message is printed and the program aborts.

Internal Function Trap Procedure

The trap procedure that is called for internal function errors must have two formal arguments in the following order:

1. A single integer containing the error number that is determined by the internal function in which the error occurred.
2. The result.

If the trap procedure returns normally and has set the error number (first argument) to zero, a standard message is printed and the program aborts.

Control-Y Trap Procedure

The specified user subroutine is called if you specify CONTROLY in an ON statement and if you type Control-Y from the terminal while the program is running.

No parameters are permitted in this trap procedure.

The actions to be taken after a trap occurs are specified by the CALL *procedure*.

If you specify CALL *procedure* for an error condition when an error occurs, the corresponding trap (if enabled) suppresses output of the normal error message, transfers control to a user-defined trap procedure, and passes zero or more parameters describing the error to this procedure. This procedure can attempt to analyze or recover from the error, or can execute another programming path you specify.

Exiting a Trap Procedure

Upon exit from a trap procedure, control returns to the instruction following the one that activated the trap. However, in the case of external and internal function traps, if the trap procedure returns normally and has set the error number (first argument) to zero, a standard message is printed and the program aborts.

I/O Run-Time Errors

During the execution of a FORTRAN 77 program, error messages may be printed on the output unit by the input/output library supplied for FORTRAN programs. The error message and a complete listing of run-time errors returned in the IOSTAT variable or printed on the output unit are listed in Appendix A.

If the IOSTAT and ERR specifiers are present, or the END specifier, or all three, the I/O error number is stored in the IOSTAT variable and control transfers to the ERR label, where a user routine can decode and handle the error if desired. Range checking errors (when the RANGE directive is ON) in arguments to input/output statements occur outside the input/output system and are handled as fatal out-of-range errors; they are not trapped by the IOSTAT and ERR specifiers.

10

Data Format in Memory

HP FORTRAN 77 has the following data types:

General Name	Data Type
Integer	BYTE (LOGICAL*1)
	INTEGER*2
	INTEGER*4
Real	REAL*4
	REAL*8
	REAL*16
Complex	COMPLEX*8
	COMPLEX*16
Logical	LOGICAL*2
	LOGICAL*4
Character	CHARACTER

In addition, the Hollerith format is available for compatibility with older programs and with some system routines.

This chapter describes the format of each data type when stored in memory.

Note

In the floating-point formats, when dealing with numbers at or close to the limits of the range, a program can exceed the range during ASCII-to-binary conversion and vice-versa. This is due to rounding errors.

Note

Make sure all variables are properly initialized. The MPE/iX Link Editor does not initialize all the stack space as the Segmenter does on MPE V. Uninitialized variables that did not cause problems on MPE V/E-based systems might cause programs to abort on MPE/iX-based systems.

HP FORTRAN 77/V stores variables greater than eight bytes indirectly; HP FORTRAN 77/iX stores the variables directly.

Overflow Conditions

Each data type has its own format and range. An overflow condition occurs when numbers outside the range of a particular type are assigned to, or read into, the corresponding variables at run-time.

For read operations, an error message is issued and the program terminates unless the ERR or IOSTAT specifier is present. For assignments, truncation occurs. For integer variables, this truncation is performed such that the high order bits are ignored. For REAL*4 variables assigned as REAL*8 values, the low order bits are ignored, preserving the magnitude, but losing the precision. Truncation for complex variables is the same as that for real variables.

This process is not to be confused with evaluation of an expression whose result is too large or small for the data type involved, which causes machine errors unless the corresponding ON statement trap has been set.

BYTE (LOGICAL*1) Format

A BYTE or LOGICAL*1 datum is always an exact representation of a one-byte integer whose values can be positive, negative, or zero.

The BYTE format occupies eight bits and has a range of:

-128 to $+127$

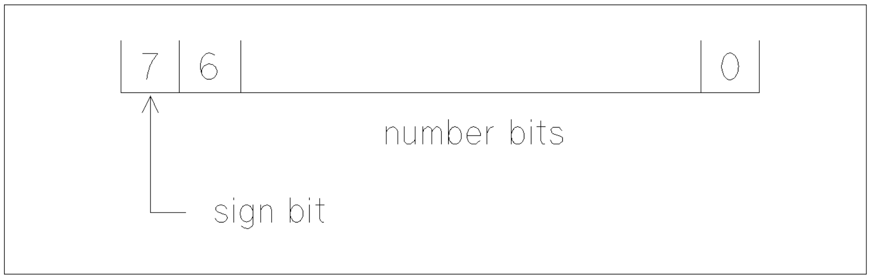


Figure 10-1. BYTE (LOGICAL*1) Format

A BYTE datum can also be used as a logical value, representing true or false. If bit 0 is 1, the value is true; otherwise it is false.

INTEGER*2 Format

An INTEGER*2 datum is always an exact representation of an integer, whose values can be positive, negative, or zero.

An INTEGER*2 datum occupies half a 32-bit word (two bytes), in two's complement format, and has a range of:

-32768 to $+32767$ (-2^{15} to $+2^{15}-1$)

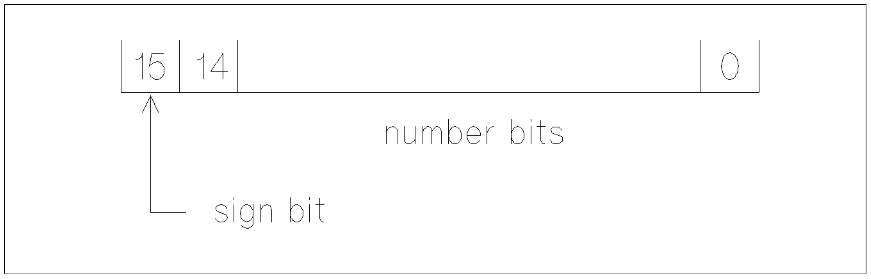


Figure 10-2. INTEGER*2 Format

INTEGER*4 Format

An INTEGER*4 datum is always an exact representation of an integer, whose values can be positive, negative, or zero.

An INTEGER*4 datum occupies one 32-bit word (four bytes), in two's complement format, and has a range of:

-2147483648 to $+2147483647$ (-2^{31} to $+2^{31}-1$)

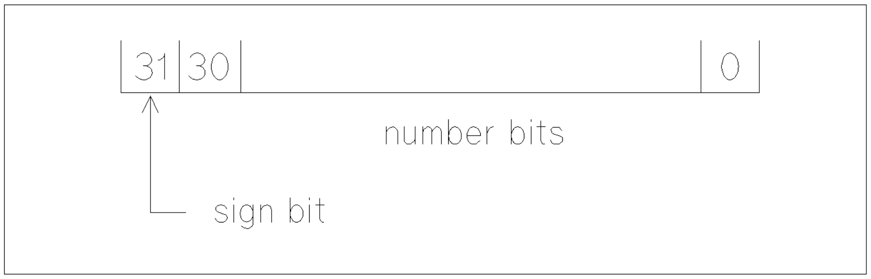


Figure 10-3. INTEGER*4 Format

REAL*4 Format

A REAL or REAL*4 datum is a processor approximation of a real number, whose values can be positive, negative, or zero.

A REAL*4 datum occupies one 32-bit word in memory, in floating-point format. It has an approximate normalized range of:

0.0
and
$\pm 1.175494\times10^{-38}$ to $\pm 3.402823\times10^{+38}$

In addition, it has an approximate denormalized range of:

$\pm 1.401298\times10^{-45}$ to $\pm 1.175494\times10^{-38}$

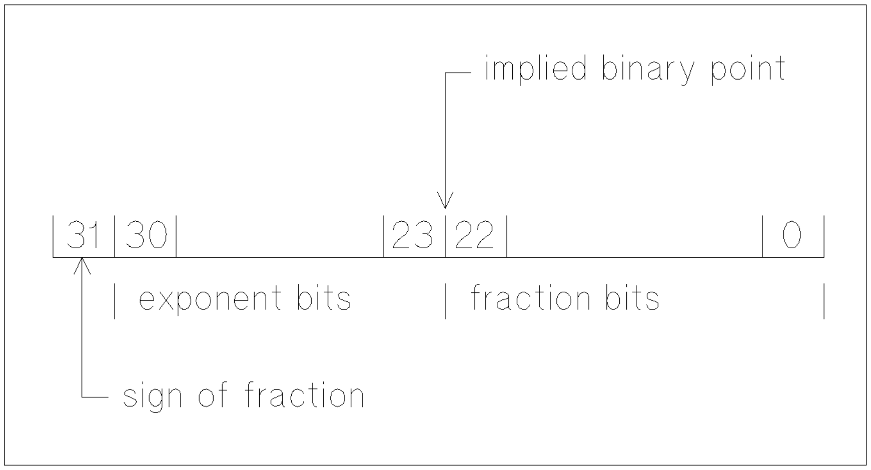


Figure 10-4. REAL*4 Format

The REAL*4 format has an 8-bit exponent and a 23-bit fraction. Significance to the user is approximately seven decimal digits. The sign bit is zero for plus, 1 for minus. The exponent field contains 127 plus the actual exponent (power of two) of the number. Exponent fields containing all zeros and all ones are "reserved." If the exponent is zero and the fraction zero, the number is interpreted as a signed zero. If the exponent is zero and the fraction not zero, the number is called "denormalized." A floating-point number stored in a "normalized" form has a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point; the denormalized number does not have this implied leading 1 to the left of the binary point.

If the exponent is all ones and the fraction is zero, the number is regarded as a signed infinity. If the exponent is all ones and the fraction is not zero, then the interpretation is "not-a-number" (NaN). Attempts to operate on infinities and NaNs cause a system trap.

REAL*8 Format

A REAL*8 or DOUBLE PRECISION datum is a processor approximation to a real number, whose values can be positive, negative, or zero.

A REAL*8 datum occupies two consecutive 32-bit words in memory in floating-point format. It has an approximate normalized range of:

0.0
and
$\pm 2.225073858507202 \times 10^{-308}$ to $\pm 1.797693134862315 \times 10^{+308}$

In addition, it has an approximate denormalized range of:

$\pm 4.940656458412466 \times 10^{-324}$ to $\pm 2.225073858507201 \times 10^{-308}$

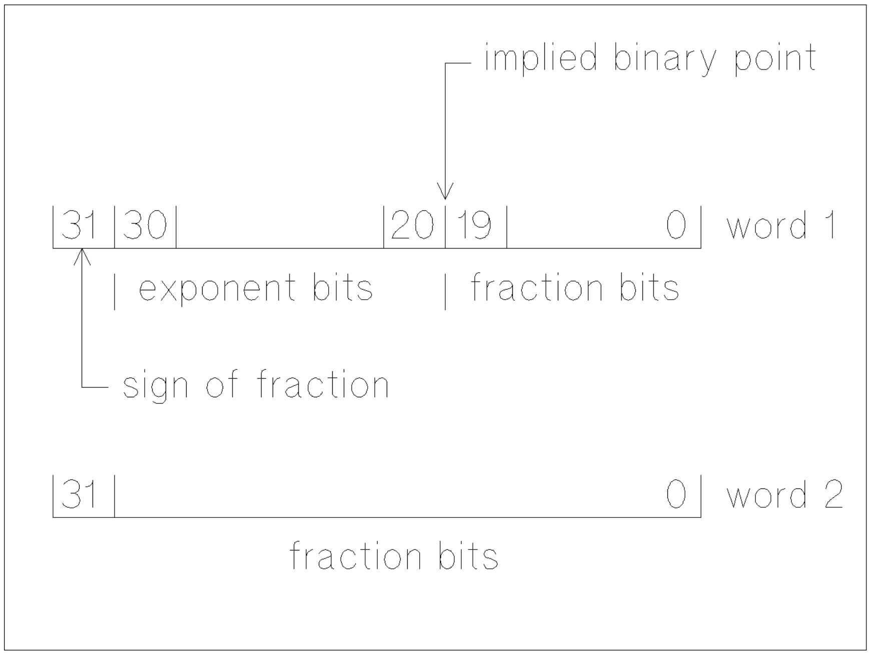


Figure 10-5. REAL*8 Format

A REAL*8 datum is 64-bit-aligned; that is, its address is divisible by eight.

The REAL*8 format has an 11-bit exponent and a 52-bit fraction. Significance to the user is approximately 16 decimal digits. The sign bit is zero for plus, one for minus. The exponent field contains 1023 plus the actual exponent (power of 2) of the number. Exponent fields containing all zeros and all ones are "reserved." If the exponent is zero and the fraction zero, the number is interpreted as a signed zero. If the exponent is zero and the fraction not zero, the number is called "denormalized." A floating-point number stored in a "normalized" form has a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point; a denormalized number does not have this implied leading 1 to the left of the binary point.

If the exponent is all ones and the fraction is zero, the number is regarded as a signed infinity. If the exponent is all ones and the fraction is not zero, then the interpretation is “not-a-number” (NaN). Attempts to operate on denormalized numbers, infinities, and NaNs cause a system trap.

REAL*16 Format

A REAL*16 datum is a processor approximation to a real number, whose values can be positive, negative, or zero. This is an extension to the ANSI 77 standard.

An REAL*16 datum occupies four consecutive 32-bit words in memory, in floating-point format. It has an approximate normalized range of:

0.0
and
$\pm 3.36210314311209350626267781732175 3\times 10^{-4932}$
to
$\pm 1.18973149535723176508575932662800 7\times 10^{+4932}$

In addition, it has an approximate denormalized range of:

$\pm 6.47517511943802511092443895822764 7\times 10^{-4966}$
to
$\pm 3.36210314311209350626267781732175 2\times 10^{-4932}$

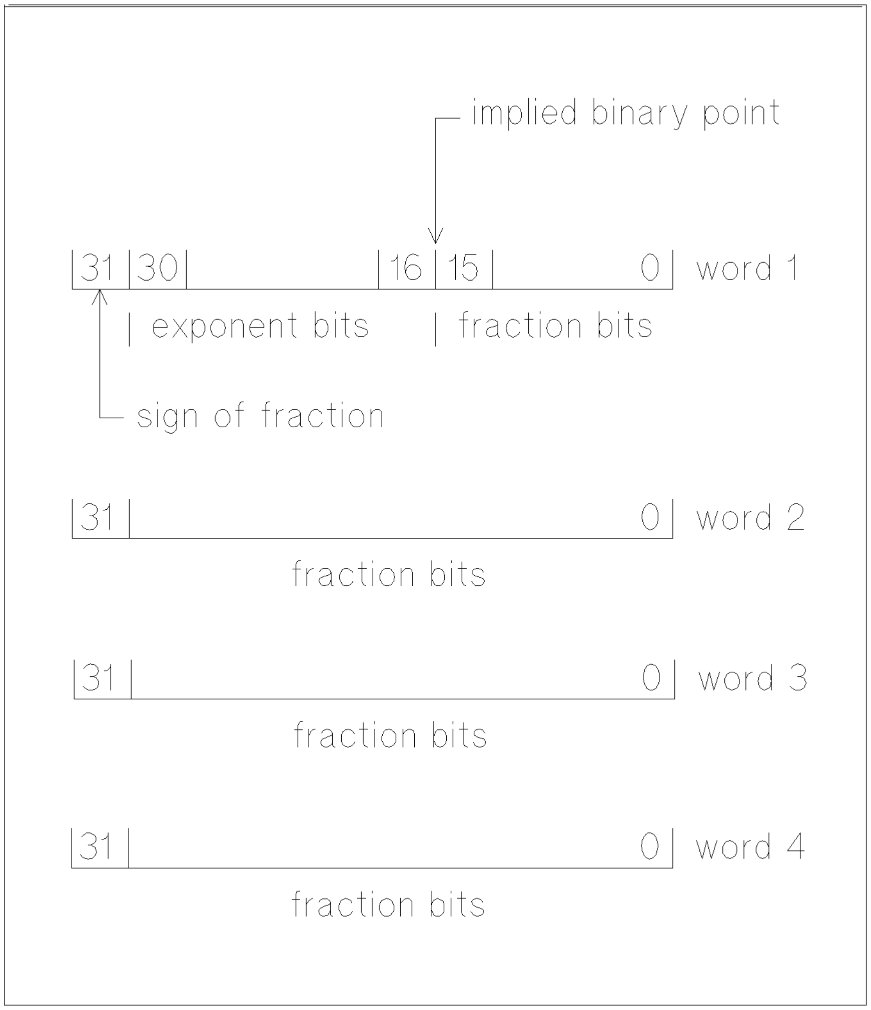


Figure 10-6. REAL*16 Format

A REAL*16 datum is 64-bit-aligned; that is, its address is divisible by eight.

The REAL*16 format has a 15-bit exponent and a 112-bit fraction. Significance to the user is approximately 34 decimal digits. The sign bit is zero for plus, one for minus. The exponent field contains 16,383 plus the actual exponent (power of 2) of the number. Exponent fields containing all zeros and all ones are "reserved." If the exponent is zero and the fraction zero, the number is interpreted as a signed zero. If the exponent is zero and the fraction not zero, the number is called "denormalized." A floating-point number stored in a "normalized" form has a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point; a denormalized number does not have this implied leading 1 to the left of the binary point.

Note

If the ANSI directive is ON, the use of REAL*16 intrinsics, constants or directives produces an ANSI warning at compile time.

If the exponent is all ones and the fraction is zero, the number is regarded as a signed infinity. If the exponent is all ones and the fraction is not zero, then the interpretation is "not-a-number" (NaN). Attempts to operate on denormalized numbers, infinities, and NaNs cause a system trap.

COMPLEX*8 Format

A COMPLEX or COMPLEX*8 datum is a processor approximation to the value of a complex number.

A COMPLEX*8 datum occupies two consecutive 32-bit words in memory. The real and the imaginary parts are each stored in one word, in the format of a REAL*4 datum. The value of each part is determined as for a REAL*4 datum.

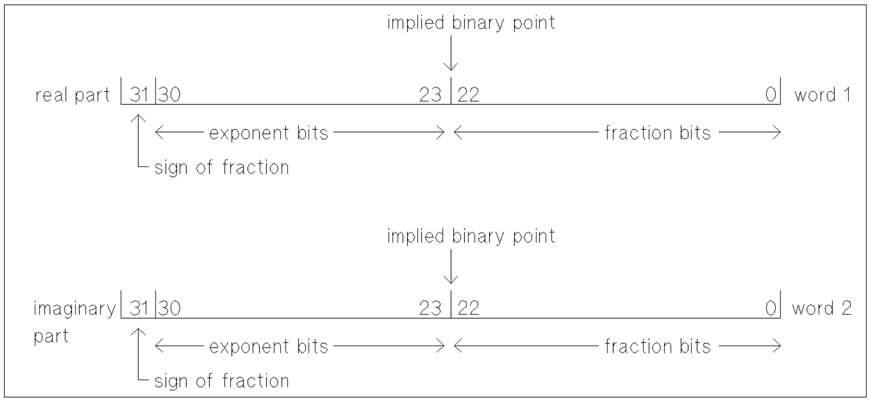


Figure 10-7. COMPLEX*8 Format

COMPLEX*16 Format

A COMPLEX*16 or DOUBLE COMPLEX datum is a processor approximation to the value of a complex number.

A COMPLEX*16 datum occupies four consecutive 32-bit words in memory. The real and the imaginary parts are each stored in two words, in the format of a REAL*8 datum. The value of each part is determined as for a REAL*8 datum.

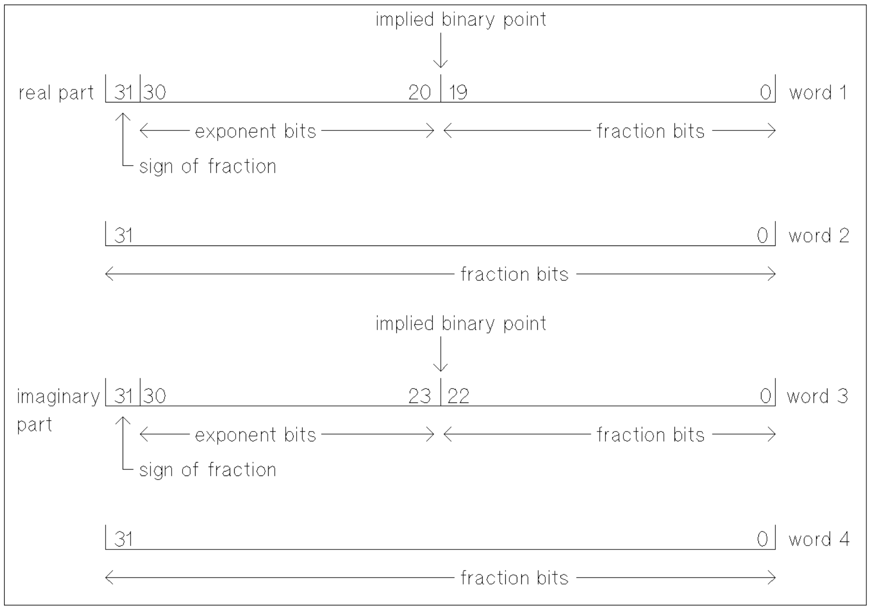


Figure 10-8. COMPLEX*16 Format

LOGICAL*2 Format

A LOGICAL*2 datum is a representation of true or false.

The LOGICAL*2 format occupies half of one 32-bit word in memory.

If the least significant bit (8) of the most significant byte is 1, the value is true; otherwise it is false.

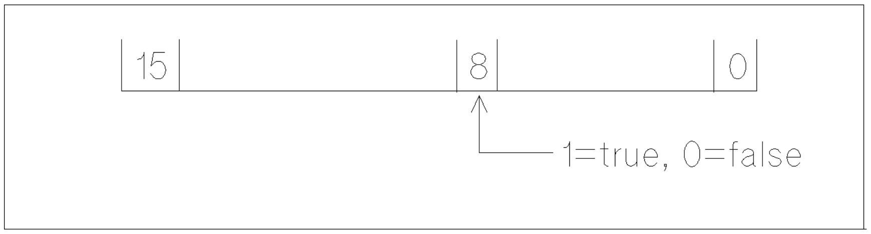


Figure 10-9. LOGICAL*2 Format

LOGICAL*4 Format

A LOGICAL*4 datum is a representation of true or false.

The LOGICAL*4 format occupies one 32-bit word in memory.

If the least significant bit (24) of the most significant byte is 1, the value is true; otherwise it is false.

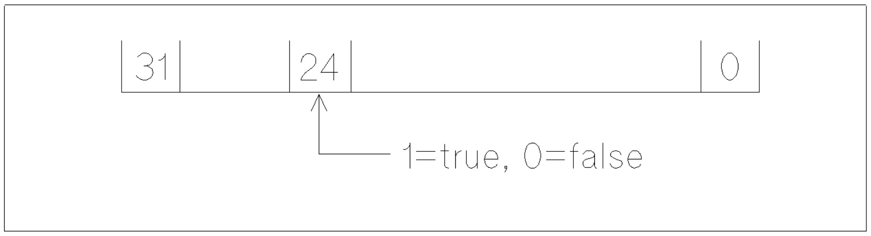


Figure 10-10. LOGICAL*4 Format

Character Format

A character datum is a character string taken from the HP ASCII character set. An ASCII character occupies one byte (eight bits) of a 32-bit word, and are packed four to a word in memory.

Character variables and constants can start or end or both in the middle of a word. The other bytes of the word may be used by the compiler as part of other variables or constants, or they may be unused.

When character items are passed as actual arguments, the address of the characters is passed, followed by an integer containing the number of characters in the string. The integer is passed by value.

Hollerith Format

The Hollerith format is available for compatibility with older programs and with some system routines. Hollerith constants are described in "Hollerith Constants" in Chapter 2. A Hollerith constant has the same format when stored in memory as a character datum.

Hollerith constants start on word boundaries. When passed as actual parameters, the word address is used (no descriptor).

A

Diagnostic Messages

Errors can be detected during several stages of the program development cycle. Errors resulting from illegal FORTRAN syntax and semantics are compile-time errors, and those from logic are run-time errors. During the execution of FORTRAN programs, errors can be generated from several sources, including library routine reference errors, input/output formatter errors, and remote file access errors.

Note The generic term "error," when referring to the diagnostic messages, includes warnings, disasters, and all errors.

Compile-Time Diagnostics

Table A-1 lists the three types of FORTRAN compiler diagnostics.

Table A-1. Types of FORTRAN Diagnostics

Type	Description
Warning	The compiler continues to process the statement, but the object code may be erroneous. The program should be recompiled. Warnings are numbered 700 and above. Warnings 800 through 829 are warnings of non-ANSI features and are issued only when the ANSI compiler directive is specified. Message 830 appears when the ANSI compiler directive is specified or when the STANDARD_LEVEL compiler directive is set to `HP`.
Error	The compiler ignores the remainder of the erroneous source statement, including any continuation lines. No object code is generated, and the program must be recompiled. Compilation continues beyond the erroneous statement, but only to check for errors. Errors are numbered below 700.
Disaster	The compiler ignores the remainder of the FORTRAN source file. The error must be corrected before compilation can proceed.

Run-Time Errors

During execution of the object program, error messages may be printed on the output unit by the input/output library supplied for FORTRAN programs. The error message is printed in the form:

```
*** FORTRAN zzz Error nnn: mmm
      File: fff, Unit: uuu
      Last format: xxx
```

where:

zzz	is "I/O" or "RANGE".
nnn	is the error number.
mmm	is the error message.
fff	is the name of the file upon which I/O failed.
uuu	is the unit number associated with the file.
xxx	is the most recent format attempted or completed.

The `File:` line is issued only if there is an I/O error. If an internal file is being used, the message is `Internal File`.

The `Last format:` line is issued only if applicable, that is, only if there is an error in a formatted transfer.

If the `IOSTAT=ios` and `ERR=label` specifiers are present, or the `END=label` specifier, or all three, the I/O error number is stored in `ios` and control transfers to `label`, where a user routine can decode and handle the error if desired. If the `END=` specifier is present and an end-of-file is encountered on a READ, `ios` is set to -1.

In the following list of messages, where more than one meaning is given in the CAUSE section, there are several possible causes, though probably only one will pertain to a specific occurrence. Where more than one action is given in the ACTION section, there are several possible solutions. Try one at a time, recompile, then run the program again.

Note

The run-time diagnostics are distinct from the compile-time diagnostics, except for format errors, which are issued at compile time when detected by the compiler or at run time when detected by the I/O library.

When the compiler finds run-time errors, it issues error messages in the range 900 to 981. The run-time errors are implementation dependent. Errors 5000 and above are internal errors. If you receive one of these errors, please contact your HP service representative.

Compile-Time Errors

A compile-time error is one detected by the compiler. There are two types of compile-time errors:

- Those that can be attributed to a specific source statement.
- Those that can be detected only after processing of an entire module.

1	COMPILE-TIME ERROR	`I/O control parameter expected`
	CAUSE	An unrecognized word was found where an I/O control keyword is required.
2	COMPILE-TIME ERROR	`Expecting operator`
	CAUSE	An operator is missing.
3	COMPILE-TIME ERROR	`String expected`
	CAUSE	A string expression in the syntax is missing.
4	COMPILE-TIME ERROR	`"/" expected`
	CAUSE	A "/" is missing.
5	COMPILE-TIME ERROR	`"-" expected`
	CAUSE	A "-" is missing.

6	`COMPILE-TIME ERROR`	`Integer or string expected`
	`CAUSE`	An integer or a string expression is missing.
8	`COMPILE-TIME ERROR`	`Integer expected`
	`CAUSE`	An integer item is missing where it is required.
11	`COMPILE-TIME ERROR`	`Expecting expression or subexpression`
	`CAUSE`	An expression is missing where it is required.
12	`COMPILE-TIME ERROR`	`Logical end of statement already encountered`
	`CAUSE`	More input is found in the input line after the logical end of a statement.
13	`COMPILE-TIME ERROR`	`Unrecognizable statement`
	`CAUSE`	The input line is not a legal FORTRAN statement.
14	`COMPILE-TIME ERROR`	`"THEN" expected`
	`CAUSE`	The “THEN” keyword is missing in an IF-THEN statement.
15	`COMPILE-TIME ERROR`	`"," or ")" expected`
	`CAUSE`	A “,” or a “)” is missing where it is required.

16	`COMPILE-TIME ERROR`	`","  expected`
	`CAUSE`	A "," is missing where it is required.
17	`COMPILE-TIME ERROR`	`"(" or "=" expected`
	`CAUSE`	A "(" or "=" is missing where it is required.
18	`COMPILE-TIME ERROR`	`")" expected`
	`CAUSE`	The number of opening and closing parentheses is unbalanced.
19	`COMPILE-TIME ERROR`	`"(" expected`
	`CAUSE`	The number of opening and closing parentheses is unbalanced.
20	`COMPILE-TIME ERROR`	`"=" expected`
	`CAUSE`	A "=" is missing where it is required.
21	`COMPILE-TIME ERROR`	`Value expected`
	`CAUSE`	An expression or subexpression is missing.
22	`COMPILE-TIME ERROR`	`Integer expression expected`
	`CAUSE`	An integer expression is missing where it is required.

23	COMPILE-TIME ERROR	`Identifier expected`
	CAUSE	An identifier is missing where it is required.
25	COMPILE-TIME ERROR	`"TO" expected`
	CAUSE	The keyword “TO” is missing.
26	COMPILE-TIME ERROR	`"DS" expected`
	CAUSE	
28	COMPILE-TIME ERROR	`Expecting I/O control list or unit specifier`
	CAUSE	There is no I/O control list or unit specifier in an I/O statement.
29	COMPILE-TIME ERROR	`Expecting format specifier`
	CAUSE	The format specifier in a formatted I/O statement is missing.
30	COMPILE-TIME ERROR	`Label, statement, or "THEN" expected`
	CAUSE	
31	COMPILE-TIME ERROR	`"DO", "IF", or "WHILE" expected`
	CAUSE	The keyword “DO”, “IF”, or “WHILE” is expected.

32	COMPILE-TIME ERROR	Label or control variable expected
	CAUSE	
33	COMPILE-TIME ERROR	"IF" expected
	CAUSE	The keyword “IF” is missing.
34	COMPILE-TIME ERROR	Expecting I/O control list or format specifier
	CAUSE	
35	COMPILE-TIME ERROR	Expecting identifier, "(", or label
	CAUSE	
36	COMPILE-TIME ERROR	Expecting ":"
	CAUSE	A “:” is missing where it is required.
37	COMPILE-TIME ERROR	Label expected
	CAUSE	
38	COMPILE-TIME ERROR	"DO" expected
	CAUSE	The keyword “DO” is missing.

39	COMPILE-TIME ERROR	Bad expression
	CAUSE	
40	COMPILE-TIME ERROR	"FUNCTION" or identifier expected
	CAUSE	
41	COMPILE-TIME ERROR	Expecting identifier or "("
	CAUSE	
42	COMPILE-TIME ERROR	Expecting length specification
	CAUSE	
44	COMPILE-TIME ERROR	Expecting a type
	CAUSE	
46	COMPILE-TIME ERROR	"," or "/" expected
	CAUSE	A "," or "/" is missing where it is expected.
48	COMPILE-TIME ERROR	Expecting identifier or "/"
	CAUSE	

49	COMPILE-TIME ERROR	`Expecting identifier or ","`
	CAUSE	
50	COMPILE-TIME ERROR	`Expecting letter`
	CAUSE	
51	COMPILE-TIME ERROR	`Expecting constant`
	CAUSE	A constant expression is missing.
60	COMPILE-TIME ERROR	`"ON" condition expected`
	CAUSE	
61	COMPILE-TIME ERROR	`"CALL" or "ABORT" expected`
	CAUSE	
101	COMPILE-TIME ERROR	`Routine or array name used illegally`
	CAUSE	Some unqualified identifiers (for example, subprogram or array names) can be used only as actual arguments. This error is issued when one of these identifiers is used as other than an actual argument.

102	COMPILE-TIME ERROR	`Constant expression required`
	CAUSE	An expression that must be constant (evaluated at compile time) using literals and named constants cannot be evaluated, either because a term is not constant or an operation (for example, a function call) cannot be performed.
103	COMPILE-TIME ERROR	`Variable, array element, or substring name expected`
	CAUSE	An assignable variable reference is expected but a nonassignable expression is encountered.
104	COMPILE-TIME ERROR	`Numeric expression required`
	CAUSE	The expression cannot be of type logical or character.
105	COMPILE-TIME ERROR	`Integer expression required`
	CAUSE	The expression must be of type integer.
106	COMPILE-TIME ERROR	`Integer variable required`
	CAUSE	An assignable integer variable reference (for example, one assignable as an I/O parameter) is required.
107	COMPILE-TIME ERROR	`Complex expression not allowed`
	CAUSE	

108	COMPILE-TIME ERROR	Logical expression required
	CAUSE	
109	COMPILE-TIME ERROR	Character expression required
	CAUSE	
110	COMPILE-TIME ERROR	Types of operands incompatible with operator and/or each other
	CAUSE	The type conversions that permit the operation to occur cannot be performed. Either the operands are not compatible with each other, or one or more are not compatible with the operator.
111	COMPILE-TIME ERROR	Left-hand and right-hand sides are not assignment compatible
	CAUSE	Where an assignment must be performed, the value of the expression to be assigned cannot be converted into the type of the variable necessary for assignment.
112	COMPILE-TIME ERROR	Illegal or inconsistent type for routine
	CAUSE	A FUNCTION or ENTRY statement has a bad type. This error is usually caused by a mismatch between the types of entries into a subprogram.
113	COMPILE-TIME ERROR	Duplicate declaration or definition
	CAUSE	An identifier or some attribute of an identifier (for example, type) is multiply defined.

114	COMPILE-TIME ERROR	Array not declared
	CAUSE	An identifier used as an array is not declared as such.
115	COMPILE-TIME ERROR	This statement not allowed in this kind of module
	CAUSE	1. A RETURN statement appears in a program module. 2. An executable statement appears in a block data module.
116	COMPILE-TIME ERROR	Program, block data, or common block name used illegally
	CAUSE	
117	COMPILE-TIME ERROR	Constant name used illegally
	CAUSE	
118	COMPILE-TIME ERROR	Illegal or inconsistent use of routine name
	CAUSE	1. An intrinsic function is used as a subroutine, or an intrinsic subroutine as a function. 2. An unrecognized routine name is declared intrinsic. 3. A routine used as an actual argument is not declared intrinsic or external.

119	COMPILE-TIME ERROR	Illegal to use argument here
	CAUSE	
120	COMPILE-TIME ERROR	Illegal use of label
	CAUSE	1. A label is inconsistently used. (The three types of labels - for formats, executable statements, and nonexecutable statements - are mutually exclusive in usage.) 2. A label is referenced but never defined.
121	COMPILE-TIME ERROR	Illegal label name
	CAUSE	
122	COMPILE-TIME ERROR	Illegal module or entry name
	CAUSE	A module or entry name is either illegal or previously declared as something else.
123	COMPILE-TIME ERROR	Referenced identifier is not a function or array
	CAUSE	
124	COMPILE-TIME ERROR	Subroutine or function name expected
	CAUSE	An identifier not representing a subprogram name is declared external or intrinsic.

125	COMPILE-TIME ERROR	`Illegal statement function name`
	CAUSE	
126	COMPILE-TIME ERROR	`Number of subscripts does not match declared number`
	CAUSE	
127	COMPILE-TIME ERROR	`Number of parameters does not match declared number`
	CAUSE	The declared number of parameters to a statement function does not match the number used.
128	COMPILE-TIME ERROR	`Illegal or inconsistent alternate return`
	CAUSE	1. Alternate returns are declared in a program unit, where they are illegal (they are allowed only in subroutines and entries within subroutines). 2. A return statement with an alternate return value is found within a program unit in which alternate returns are illegal.
129	COMPILE-TIME ERROR	`Illegal operator`
	CAUSE	
130	COMPILE-TIME ERROR	`Illegal or duplicate argument`
	CAUSE	A formal argument in a program, subprogram, or entry statement either appears more than once or is previously defined.

131	COMPILE-TIME ERROR	Illegal or duplicate SAVE statement
	CAUSE	1. A formal argument or variable in common is saved. 2. A variable is saved more than once. 3. A SAVE statement saving all variables is encountered with SAVE statements saving individual variables.
132	COMPILE-TIME ERROR	Illegal or duplicate COMMON definition
	CAUSE	1. An identifier not representing a variable is placed in a common block. 2. A variable is declared to be in common more than once. 3. A formal argument or saved variable is placed in common.
133	COMPILE-TIME ERROR	Illegal length in type statement
	CAUSE	A length field is attached to a simple type forming an unsupported or illegal type.

134	COMPILE-TIME ERROR	`Structure no declared`
	CAUSE	A reference to an undeclared structure in a RECORD statement.
135	COMPILE-TIME ERROR	`Recursive structure declaration`
	CAUSE	A reference was made to a structure inside itself.
136	COMPILE-TIME ERROR	`Illegal or duplicate IMPLICIT declaration`
	CAUSE	1. In an IMPLICIT statement, an illegal prefix character (that is, not `a` through `z` or `A` through `Z`) is encountered. 2. A prefix character is declared implicit more than once. 3. IMPLICIT NONE is encountered with other IMPLICIT specifications.

137	COMPILE-TIME ERROR	`Identifier not declared`
	CAUSE	An identifier does not appear in a type statement when IMPLICIT NONE is specified.
138	COMPILE-TIME ERROR	`Illegal or inconsistent EQUIVALENCE statement`
	CAUSE	1. A formal parameter or a dynamic array is equivalenced using the EQUIVALENCE statement. 2. An identifier not representing a variable is equivalenced using the EQUIVALENCE statement. 3. A variable is illegally equivalenced to more than one position. 4. A variable is forced to be misaligned with the current EQUIVALENCE statement.
139	COMPILE-TIME ERROR	`Illegal literal`
	CAUSE	
140	COMPILE-TIME ERROR	`Illegal DO termination statement`
	CAUSE	A statement not allowed as the last statement in a DO loop is used as such.
141	COMPILE-TIME ERROR	`Illegal control flow - transfer into block`
	CAUSE	

143	COMPILE-TIME ERROR	`Missing label on FORMAT statement`
	CAUSE	A FORMAT statement is used without a label.
144	COMPILE-TIME ERROR	`Illegal array declaration`
	CAUSE	1. A calculated dimension size has overflowed the word size of the machine. 2. A zero or negative dimension size has been declared. 3. A nonconstant array dimension is improperly specified.
145	COMPILE-TIME ERROR	`No result assigned to function`
	CAUSE	There is no assignment of any value to the function result.
146	COMPILE-TIME ERROR	`Missing FORMAT statement`
	CAUSE	
147	COMPILE-TIME ERROR	`Illegal initialization`
	CAUSE	1. variable is initialized in a program unit in which initialization is not permitted. 2. A data value is not constant. 3. A subscript or substring expression in a DATA statement is not constant.

148	COMPILE-TIME ERROR	Subscript out of range
	CAUSE	
149	COMPILE-TIME ERROR	Illegal assignment to DO index
	CAUSE	
150	COMPILE-TIME ERROR	Illegal implied DO expression
	CAUSE	
151	COMPILE-TIME ERROR	Duplicate label definition
	CAUSE	
152	COMPILE-TIME ERROR	Declaration of routine as both EXTERNAL and INTRINSIC
	CAUSE	
153	COMPILE-TIME ERROR	Inconsistent parameter type
	CAUSE	A parameter identifier and the expression to be assigned to it are not assignment compatible.

154	COMPILE-TIME ERROR	Illegal parameter in option
	CAUSE	
155	COMPILE-TIME ERROR	Illegal use of string whose length is unknown
	CAUSE	
156	COMPILE-TIME ERROR	Illegal STOP or PAUSE value
	CAUSE	The value to be written by a STOP or PAUSE statement is not an integer or character literal.
157	COMPILE-TIME ERROR	Incompatible types
	CAUSE	
158	COMPILE-TIME ERROR	Duplicate initialization
	CAUSE	A variable is initialized more than once with DATA statements or in type declaration statements.
159	COMPILE-TIME ERROR	Illegal number and/or types of arguments to intrinsic function
	CAUSE	The number or types of arguments to an intrinsic routine are incompatible with the routine or each other (they must all be of the same type).

160	COMPILE-TIME ERROR	`Illegal use of intrinsic function as actual argument`
	CAUSE	A generic intrinsic routine name is passed as an actual argument to a subprogram.
161	COMPILE-TIME ERROR	`Named constant typed following definition in PARAMETER statement`
	CAUSE	
162	COMPILE-TIME ERROR	`Illegal use of Hollerith or octal or hexadecimal constant`
	CAUSE	A disallowed operation is attempted on a Hollerith, octal, or hexadecimal constant.
163	COMPILE-TIME ERROR	`Assigned GOTO on wrong type`
	CAUSE	The assigned variable is not of type INTEGER*4.
164	COMPILE-TIME ERROR	`ASSIGN statement: bad type or unreferenced label`
	CAUSE	1. The assigned variable not of type INTEGER*4. 2. The label being assigned is attached to a nonexecutable statement.

165	COMPILE-TIME ERROR	`Illegal typing of identifier`
	CAUSE	An attempt is made to type an identifier representing an entity that cannot be typed (for example, a subroutine name or a program name).
166	COMPILE-TIME ERROR	`Expression does not represent a value`
	CAUSE	An expression (such as an array name, external routine name, or intrinsic function name), although valid as an actual argument, does not represent a single value and is therefore meaningless in this context. Parameterless function calls without empty parentheses, `()`, can also cause this error.
168	COMPILE-TIME ERROR	`System intrinsic name not found in system intrinsic file`
	CAUSE	
169	COMPILE-TIME ERROR	`Invalid system intrinsic name`
	CAUSE	
170	COMPILE-TIME ERROR	`Result type of function not compatible with system intrinsic definition`
	CAUSE	The user-specified function type is incompatible with the type in the system intrinsic definition.

171	COMPILE-TIME ERROR	This type of parameter not allowed in a system intrinsic call
	CAUSE	
172	COMPILE-TIME ERROR	Null parameter not permitted here
	CAUSE	A null parameter is passed to a system intrinsic that is not OPTION VARIABLE.
173	COMPILE-TIME ERROR	This system intrinsic cannot be called by FORTRAN routines
	CAUSE	The system intrinsic procedure contains formal parameters that cannot be folded to any FORTRAN data type.
174	COMPILE-TIME ERROR	PROCEDURE or FUNCTION call not compatible with system intrinsic definition
	CAUSE	A procedure call is made to a system intrinsic that expects a function call, or a function call is made to a system intrinsic that expects a procedure call.
175	COMPILE-TIME ERROR	PROGRAM parameter must be INTEGER*4 or CHARACTER*(*) value
	CAUSE	The program parameter on a HP 3000 system is declared explicitly or typed implicitly with a type other than INTEGER*4 or CHARACTER*(*).
176	COMPILE-TIME ERROR	Illegal namelist group name
	CAUSE	

177	COMPILE-TIME ERROR	Illegal identifier in namelist group
	CAUSE	
178	COMPILE-TIME ERROR	Dummy arguments not allowed in namelist
	CAUSE	
179	COMPILE-TIME ERROR	Number of parameters not compatible with system intrinsic definition
	CAUSE	
180	COMPILE-TIME ERROR	This statement not part of HP standard FORTRAN 77
	CAUSE	This statement is not available on all implementations of HP FORTRAN 77.
181	COMPILE-TIME ERROR	IGNORE option not allowed with ON statement on this operating system
	CAUSE	
182	COMPILE-TIME ERROR	This type of ON condition not allowed with this FORTRAN data type
	CAUSE	

183	COMPILE-TIME ERROR	Illegal type in ON statement
	CAUSE	
184	COMPILE-TIME ERROR	This ON condition not allowed on this operating system
	CAUSE	
185	COMPILE-TIME ERROR	Incorrect control character specified for this operating system
	CAUSE	
186	COMPILE-TIME ERROR	ABORT not allowed with this ON condition
	CAUSE	
187	COMPILE-TIME ERROR	Initialization of shared common blocks not allowed
	CAUSE	Initialization of common blocks declared to be in shared memory cannot be initialized at compile-time.
188	COMPILE-TIME ERROR	Expression with concatenation or substrings may not be passed by value
	CAUSE	

189	COMPILE-TIME ERROR	Built-in functions %REF, %VAL used in invalid context
	CAUSE	Illegal arguments used in built-in functions.
191	COMPILE-TIME ERROR	Illegal structure declaration
	CAUSE	Either a field name was given at the outermost level of a structure declaration or the use of dynamic, assumed size, or adjustable arrays. Passed length and variable length character items are also not permitted with a structure declaration.
192	COMPILE-TIME ERROR	Illegal use of %FILL
	CAUSE	%FILL was used outside of a structure declaration.

201	COMPILE-TIME ERROR	No matching IF statement
	CAUSE	An ENDIF statement appears without a matching IF statement.
202	COMPILE-TIME ERROR	Expecting ENDIF statement
	CAUSE	An IF statement appears without a matching ENDIF statement.
203	COMPILE-TIME ERROR	Expecting DO terminator
	CAUSE	An ENDDO statement is missing.
204	COMPILE-TIME ERROR	Wrong DO terminator
	CAUSE	
205	COMPILE-TIME ERROR	Premature or unexpected DO terminator
	CAUSE	
206	COMPILE-TIME ERROR	No matching WHILE statement
	CAUSE	
207	COMPILE-TIME ERROR	No matching DO statement
	CAUSE	An ENDDO statement was found with no matching DO statement.

208	COMPILE-TIME ERROR	Expecting END DO for DO WHILE
	CAUSE	
210	COMPILE-TIME ERROR	IMPLICIT not allowed in or after executable statements
	CAUSE	The IMPLICIT statement is a specification statement and cannot appear in or after any executable statements.
211	COMPILE-TIME ERROR	PARAMETER not allowed in or after executable statements
	CAUSE	
212	COMPILE-TIME ERROR	This specification statement not allowed in or after executable statements
	CAUSE	
213	COMPILE-TIME ERROR	Statement function not allowed in or after executable statements
	CAUSE	
214	COMPILE-TIME ERROR	ENTRY not allowed within DO, DO WHILE, or block IF statement
	CAUSE	

215	COMPILE-TIME ERROR	Missing END statement
	CAUSE	A program unit has no END statement. Note that this can be caused by a compiler directive appearing in the wrong location between compilation units.
216	COMPILE-TIME ERROR	IMPLICIT not allowed after previous specification statements
	CAUSE	
217	COMPILE-TIME ERROR	IMPLICIT not allowed after DATA or statement function
	CAUSE	
218	COMPILE-TIME ERROR	PARAMETER or ALIAS not allowed after DATA or statement function
	CAUSE	
219	COMPILE-TIME ERROR	This specification statement not allowed after DATA or statement function
	CAUSE	
220	COMPILE-TIME ERROR	No matching STRUCTURE, UNION, or MAP statement
	CAUSE	An END STRUCTURE, END UNION, or END MAP was encountered without a respective declaration statement.

221	COMPILE-TIME ERROR	This statement is only allowed within a structure declaration
	CAUSE	UNION or MAP statement encountered outside of a structure declaration.
222	COMPILE-TIME ERROR	Missing END [STRUCTURE | UNION | MAP] statement
	CAUSE	An executable statement was encountered before a corresponding END statement in a STRUCTURE, UNION, OR MAP declaration.
223	COMPILE-TIME ERROR	This statement not allowed within a structure declaration
	CAUSE	Illegal nonexecutable statement encountered in a structure declaration; statements can include EQUIVALENCE, IMPLICIT, COMMON, DIMENSION, EXTERNAL, INTRINSIC, DATA, AND NAMELIST.
224	COMPILE-TIME ERROR	Expecting MAP statement
	CAUSE	Within a UNION only MAP declarations are permitted.
225	COMPILE-TIME ERROR	This statement is only allowed within a union declaration
	CAUSE	A MAP declaration was encountered outside of a UNION declaration.
241	COMPILE-TIME ERROR	Multiple main programs
	CAUSE	

301	COMPILE-TIME ERROR	Undefined character(s) on line
	CAUSE	
302	COMPILE-TIME ERROR	Ill-formed number
	CAUSE	
303	COMPILE-TIME ERROR	Ill-formed FORMAT string
	CAUSE	
304	COMPILE-TIME ERROR	Error in Hollerith literal
	CAUSE	
305	COMPILE-TIME ERROR	Continuation line error
	CAUSE	
306	COMPILE-TIME ERROR	Error in label field
	CAUSE	
307	COMPILE-TIME ERROR	Unmatched quotation marks
	CAUSE	

309	COMPILE-TIME ERROR	Arithmetic overflow
	CAUSE	
310	COMPILE-TIME ERROR	Array subscript overflow
	CAUSE	
311	COMPILE-TIME ERROR	End of file encountered
	CAUSE	
312	COMPILE-TIME ERROR	End of line encountered
	CAUSE	
313	COMPILE-TIME ERROR	This character or group of characters not permitted here
	CAUSE	
314	COMPILE-TIME ERROR	CHARACTER*(*) is only permitted in the outermost block with the directives being used
	CAUSE	CHARACTER*(*) cannot be used outside the outermost block with the $FTN3000_66 directive.
315	COMPILE-TIME ERROR	Label field of continuation line is not blank
	CAUSE	

316	COMPILE-TIME ERROR	Continuation limit exceeded
	CAUSE	The number of continuation lines exceeds the default or specified limit. The default number of continuation lines is 19.
317	COMPILE-TIME ERROR	Entry points not consistent for subroutine or function with the directives being used
	CAUSE	When using directives that change the calling convention, all entry points to a program unit must be consistent.
318	COMPILE-TIME ERROR	Variable length specifier not a formal argument
	CAUSE	When using a variable length specifier in a declaration statement, that variable must be a formal argument to the program unit.
325	COMPILE-TIME ERROR	Unknown logical operator or constant
	CAUSE	
330	COMPILE-TIME ERROR	Unknown I/O control word
	CAUSE	
335	COMPILE-TIME ERROR	Unknown compiler directive
	CAUSE	

340	COMPILE-TIME ERROR	`Compiler buffering limit exceeded`
	CAUSE	Data buffers inside the compiler have been filled to capacity. No further processing can be done on this statement. The statement's data requirements must be decreased.
350	COMPILE-TIME ERROR	`This statement is too complicated`
	CAUSE	The expression in this statement is too complicated for the compiler to handle.
360	COMPILE-TIME ERROR	`Structure name required at outer structure level`
	CAUSE	At the outermost structure declaration level, the structure must be named.
361	COMPILE-TIME ERROR	`Field name required for structure`
	CAUSE	A nested substructure must have a field name.

393	COMPILE-TIME ERROR	Unbalanced quotes
	CAUSE	Missing a beginning or closing quote in a statement.
394	COMPILE-TIME ERROR	Unbalanced parentheses
	CAUSE	Parentheses are not matched in a statement.
395	COMPILE-TIME ERROR	Undefined symbol
	CAUSE	
397	COMPILE-TIME ERROR	Undefined token class
	CAUSE	

398	COMPILE-TIME ERROR	Undefined character class
	CAUSE	
399	COMPILE-TIME ERROR	Undefined special character
	CAUSE	
400	COMPILE-TIME ERROR	Invalid type of UNIT specifier
	CAUSE	The UNIT specifier must be an INTEGER expression.
401	COMPILE-TIME ERROR	UNIT specifier not an external unit
	CAUSE	The UNIT specifier was an internal file.
402	COMPILE-TIME ERROR	Invalid UNIT for auxiliary statement
	CAUSE	The UNIT specifier given was an internal file. An internal file cannot be used with this type of I/O statement.

403	COMPILE-TIME ERROR	`Invalid FMT identifier`
	CAUSE	The FMT specifier must be one of the following: 1. Statement label of a FORMAT statement. 2. Variable that has been assigned the statement label of a FORMAT statement. 3. Character or non-character array name that contains the representation of a FORMAT statement. 4. A character expression. 5. An asterisk.
404	COMPILE-TIME ERROR	`Internal file requires FORMATTED or list-directed use`
	CAUSE	Attempted UNFORMATTED I/O on an internal file.
405	COMPILE-TIME ERROR	`If REC appears, END cannot`
	CAUSE	The END specifier cannot appear in an I/O statement if the REC specifier is present.
406	COMPILE-TIME ERROR	`Invalid type of REC specifier`
	CAUSE	The record specifier for direct access is not of type integer or does not represent a value.
407	COMPILE-TIME ERROR	`IOSTAT specifier not INTEGER*4 or INTEGER*2 variable or array element`
	CAUSE	The IOSTAT specifier must be an INTEGER*4 or INTEGER*2 variable or array element.

410	COMPILE-TIME ERROR	`FILE specifier not character expression`
	CAUSE	The FILE specifier is valid only in an INQUIRE or OPEN statement and must be a character variable, array element, or substring.
412	COMPILE-TIME ERROR	`STATUS specifier not character expression`
	CAUSE	The STATUS specifier is valid only in an OPEN statement and must be a character variable, array element, or substring.
413	COMPILE-TIME ERROR	`ACCESS specifier not character expression`
	CAUSE	The ACCESS specifier is valid only in an INQUIRE or OPEN statement and must be a character variable, array element, or substring.
414	COMPILE-TIME ERROR	`FORM specifier not character expression`
	CAUSE	The FORM specifier is valid only in an INQUIRE or OPEN statement and must be a character variable, array element, or substring.
415	COMPILE-TIME ERROR	`BLANK specifier not character expression`
	CAUSE	The BLANK specifier is valid only in an INQUIRE or OPEN statement and must be a character variable, array element, or substring.
416	COMPILE-TIME ERROR	`Both FILE and UNIT may not appear`

`CAUSE`	The FILE and UNIT specifiers cannot both appear in an input/output statement.

417	COMPILE-TIME ERROR	EXIST not LOGICAL*4 variable or array element name
	CAUSE	The EXIT= specifier of an INQUIRE statement must be assignable and of type LOGICAL*4.
418	COMPILE-TIME ERROR	OPENED not LOGICAL*4 variable or array element name
	CAUSE	The OPENED= specifier of an INQUIRE statement must be assignable and of type LOGICAL*4.
419	COMPILE-TIME ERROR	NAMED not LOGICAL*4 variable or array element name
	CAUSE	The NAMED= specifier of an INQUIRE statement must be assignable and of type LOGICAL*4.
420	COMPILE-TIME ERROR	NUMBER not INTEGER*4 variable or array element name
	CAUSE	The NUMBER= specifier of an INQUIRE statement must be assignable and of type LOGICAL*4.
422	COMPILE-TIME ERROR	NEXTREC not INTEGER*4 variable or array element name
	CAUSE	The NEXTREC= specifier of an INQUIRE statement must be assignable and of type LOGICAL*4.
423	COMPILE-TIME ERROR	NAME not character variable or array element name
	CAUSE	The NAME= specifier of an INQUIRE statement must be an assignable character data item.

424	COMPILE-TIME ERROR	`ACCESS not character variable or array element name`
	CAUSE	The `ACCESS=` specifier of an INQUIRE statement must be an assignable character data item.
425	COMPILE-TIME ERROR	`SEQUENTIAL not character variable or array element name`
	CAUSE	The `SEQUENTIAL=` specifier of an INQUIRE statement must be an assignable character data item.
426	COMPILE-TIME ERROR	`DIRECT not character variable or array element name`
	CAUSE	The DIRECT specifier is valid only in an INQUIRE statement and must be a character variable, array element, or substring.
427	COMPILE-TIME ERROR	`FORM not character variable or array element name`
	CAUSE	The `FORM=` specifier of an INQUIRE statement must be an assignable item of type CHARACTER.
428	COMPILE-TIME ERROR	`FORMATTED not character variable or array element name`
	CAUSE	The FORMATTED specifier is valid only in an INQUIRE statement and must be a character variable, array element, or substring.
429	COMPILE-TIME ERROR	`UNFORMATTED not character variable or array element name`
	CAUSE	The UNFORMATTED specifier is valid only in an INQUIRE statement and must be a character variable, array element, or substring.

430	COMPILE-TIME ERROR	`BLANK not character variable or array element name`
	CAUSE	The `BLANK=` specifier of an INQUIRE statement must be an assignable item of type CHARACTER.
434	COMPILE-TIME ERROR	`Duplicate declaration of specifier`
	CAUSE	A particular specifier was used twice within an I/O statement.
435	COMPILE-TIME ERROR	`Unit specifier required`
	CAUSE	I/O statement requires a UNIT specifier.
436	COMPILE-TIME ERROR	`If WRITE, no END specifier allowed`
	CAUSE	The END specifier is not allowed in a WRITE statement.
437	COMPILE-TIME ERROR	`RECL not INTEGER*4 variable or array element name`
	CAUSE	The `RECL=` specifier in an INQUIRE statement must be of type INTEGER*4.
438	COMPILE-TIME ERROR	`Illegal I/O parameter`
	CAUSE	This I/O specifier is allowed in other kind(s) of I/O statements(s), but is not allowed in this kind.
441	COMPILE-TIME ERROR	`FILE or UNIT parameter required`
	CAUSE	The INQUIRE statement requires either a `FILE=` specifier or a `UNIT=` specifier.

447	COMPILE-TIME ERROR	`Internal file must be sequential`
	CAUSE	Direct access I/O was attempted on an internal file.
448	COMPILE-TIME ERROR	`Direct access file cannot use list directed I/O`
	CAUSE	Attempted to do list-directed I/O on a file opened for direct access.
449	COMPILE-TIME ERROR	`Assumed size array not allowed`
	CAUSE	The internal files may not be assumed size arrays.
450	COMPILE-TIME ERROR	`NML not allowed with formatted or list directed I/O`
	CAUSE	Attempted to use NML specifier for formatted or list-directed I/O.
451	COMPILE-TIME ERROR	`Direct access file cannot use namelist directed I/O`
	CAUSE	Namelist directed I/O is not allowed on a file opened for direct access.
452	COMPILE-TIME ERROR	`I/O list not allowed in namelist directed I/O statement`
	CAUSE	Namelist directed I/O statement must not have I/O list.

454	COMPILE-TIME ERROR	`Array cannot be adjustable or assumed size in ENCODE/DECODE`
	CAUSE	An adjustable or assumed size array was used as the unit parameter in an ENCODE/DECODE statement.
455	COMPILE-TIME ERROR	`Illegal use of aggregate reference in I/O list`
	CAUSE	unformatted I/O is the only I/O that can be performed on aggregate record references.
500	COMPILE-TIME ERROR	`Fatal internal compiler error`
	CAUSE	An error occurred in the compiler or run-time library that was improperly handled. Please report this problem to your HP representative.
501	COMPILE-TIME ERROR	`Include file cannot be opened`
	CAUSE	The include file cannot be found, is already opened exclusively, or is some other type of illegal file.
502	COMPILE-TIME ERROR	`Include level limit exceeded`
	CAUSE	The nesting level limit has been exceeded for include statements or directives. The maximum nesting depth is 8.

503	COMPILE-TIME ERROR	`Nesting level limit exceeded`
	CAUSE	The maximum level of nesting (75) for global values has been reached.
504	COMPILE-TIME ERROR	`Range violation detected at compilation time`
	CAUSE	The compiler detected an attempt to access an array outside its declared bounds.
505	COMPILE-TIME ERROR	`Unable to open file`
	CAUSE	The file is not present or not readable.
506	COMPILE-TIME ERROR	`Unable to open FORTRAN message catalog`
	CAUSE	The internal catalog containing the error messages cannot be opened. (And note that only the number, and not the text, of this error message appears.)
508	COMPILE-TIME ERROR	`Unable to obtain extra data segment`
	CAUSE	The system configuration is insufficient to provide compiler's extra data segment. (On MPE/V systems only.)
558	COMPILE-TIME ERROR	`Unable to open compiler communication file`
	CAUSE	Unable to open UCODEIN, a file used for communication between the two compiler processes. (On MPE/V systems only.)

559	COMPILE-TIME ERROR	`Unable to open compiler communication file`
	CAUSE	Unable to open UCODEOUT, a file used for communication between the two compiler processes. (On MPE/V systems only.)
561	COMPILE-TIME ERROR	`Create process failed: no process handling capability`
	CAUSE	The second compiler process could not be created due to a lack of process handling capabilities. (On MPE/V systems only.)
564	COMPILE-TIME ERROR	`Create process failed: out of system resources`
	CAUSE	The system did not have enough resources to create the second compiler process. (On MPE/V systems only.)
566	COMPILE-TIME ERROR	`Create process failed: FTN2.PUB.SYS does not exist`
	CAUSE	The second compiler process could not be created because its program file does not exist. (On MPE/V systems only.)
575	COMPILE-TIME ERROR	`Create process failed: exceeds configuration`
	CAUSE	The system configuration did not allow the second compiler process to be created. (On MPE/V systems only.)
576	COMPILE-TIME ERROR	`Create process failed: hard load error occurred`
	CAUSE	Error 16 is returned from CREATEPROCESS intrinsic. (On MPE/V systems only.)

577	COMPILE-TIME ERROR	`Create process failed: illegal priority class specified`
	CAUSE	Error 17 is returned from CREATEPROCESS intrinsic. (On MPE/V systems only.)
601	COMPILE-TIME ERROR	`Unable to open system intrinsic file`
	CAUSE	The system intrinsic file cannot be opened, is already opened exclusively, or is some other type of illegal file.
602	COMPILE-TIME ERROR	`Corrupt system intrinsic file`
	CAUSE	
603	COMPILE-TIME ERROR	`Invalid system intrinsic file name`
	CAUSE	The specified file is not a system intrinsic file.
604	COMPILE-TIME ERROR	`Functions SECNDS and RAN are not supported in compatibility mode.`
	CAUSE	These functions were written for IEEE floating-point format only. Do not use them with the $HP3000_16 ON directive.

Compile-Time Warnings

700	COMPILE-TIME WARNING	`Missing semantics in option: option ignored`
	CAUSE	Internal error. Please notify your HP representative.
701	COMPILE-TIME WARNING	`Arithmetic overflow`
	CAUSE	The operation can cause arithmetic overflow.
703	COMPILE-TIME WARNING	`Type conversion performed`
	CAUSE	During evaluation, a smaller, simpler data type had to be converted to another data type for compatibility.
704	COMPILE-TIME WARNING	`Overflow in numeric literal`
	CAUSE	The value specified in this numeric constant is too large in absolute value for its data type on this machine.
705	COMPILE-TIME WARNING	`Illegal SHORT option type`
	CAUSE	The type suboption specified is not a legal type for the SHORT compiler option.
706	COMPILE-TIME WARNING	`Illegal LONG option type`
	CAUSE	The type suboption specified is not a legal type for the LONG compiler option.

708	COMPILE-TIME WARNING	`Directive continuation line not found`
	CAUSE	Next noncomment line is not a directive.
709	COMPILE-TIME WARNING	`This option is allowed only at the beginning of a program unit: ignored`
	CAUSE	For consistency, this option must be declared before the PROGRAM or SUBROUTINE statement.
710	COMPILE-TIME WARNING	`ALIAS directive improperly applied: directive ignored`
	CAUSE	Improper specification of the ALIAS directive.
711	COMPILE-TIME WARNING	`This compiler option not available on this operating system: option ignored`
	CAUSE	The compiler has recognized this compiler option as valid; however, it is not available on the host operating system.
712	COMPILE-TIME WARNING	`Illegal comment`
	CAUSE	The "!" is not allowed for embedding a comment.
714	COMPILE-TIME WARNING	`Conditional compilation nesting level exceeded: this option and corresponding ELSE and ENDIF options ignored`
	CAUSE	The nesting level of the conditional compilation directives has exceeded the limit. The limit is 16.

716	COMPILE-TIME WARNING	`Compiler directive does not begin in column 1`
	CAUSE	A dollar sign, “$”, denoting a compiler directive was found in a column other than column 1.
717	COMPILE-TIME WARNING	`Short doubles not allowed by this compiler`
	CAUSE	The REAL*6 data type is not permitted.
718	COMPILE-TIME WARNING	`")" expected in option: ignored`
	CAUSE	The “) ” is expected in the directive `$IF (expr)` or `$SET (list)`.
719	COMPILE-TIME WARNING	`"(" expected in option: ignored`
	CAUSE	The “(” is expected in the directive `$IF (expr)` or `$SET (list)`.
720	COMPILE-TIME WARNING	`"=" expected in option: ignored`
	CAUSE	The directive `$SET` expects expressions in the form: $\texttt{\$SET (flat = } \left\{ \begin{array}{l} \texttt{.TRUE.} \\ \texttt{.FALSE.} \end{array} \right\} \texttt{)}$
721	COMPILE-TIME WARNING	`Unsupported specifier: option ignored`
	CAUSE	This nonstandard I/O specifier is not supported.

722	COMPILE-TIME WARNING	`Illegal form of UNIT specified: ignored`
	CAUSE	The UNIT specifier in this statement was of improper form, and thus ignored.
723	COMPILE-TIME WARNING	`Identifier expected in option: ignored`
	CAUSE	This option requires an identifier.
724	COMPILE-TIME WARNING	`Compiler option identifier expected`
	CAUSE	A valid compiler directive word was expected, but none encountered.
725	COMPILE-TIME WARNING	`OPTIMIZE and SYMDEBUG mutually exclusive: option ignored`
	CAUSE	The `$OPTIMIZE` cannot be used when `$SYMDEBUG` is ON.
726	COMPILE-TIME WARNING	`If trap handling procedure modifies globals, optimization may fail`
	CAUSE	The optimizer assumes that the trap handling code does not effect globals. Violating this assumption can cause incorrect code.
728	COMPILE-TIME WARNING	`Divide by zero detected at compile time; zero result inserted`
	CAUSE	During compilation, a divide by zero was detected.
729	COMPILE-TIME WARNING	`This ON condition not available with emulated floating point programs: trap not set`

`CAUSE`	The specified error condition may not be trapped with emulated floating point programs.

730	COMPILE-TIME WARNING	`Illegal string in option or statement: ignored`
	CAUSE	An illegal date in the `$COPYRIGHT` directive; check the syntax.
731	COMPILE-TIME WARNING	`Expecting constant in option: ignored`
	CAUSE	A `.TRUE.` or `.FALSE.` is expected in the `$SET` option.
732	COMPILE-TIME WARNING	`Logical end of option already encountered: option ignored`
	CAUSE	Extra character found in an option that has already been processed.
734	COMPILE-TIME WARNING	`"/" expected in option: ignored`
	CAUSE	A syntax error was found in the COMMON statement.
735	COMPILE-TIME WARNING	`Unrecognizable compiler option or suboption`
	CAUSE	The compiler option or suboption does not exist.
736	COMPILE-TIME WARNING	`"HP9000" and "FTN3000_66" mutually exclusive: previous option turned off`
	CAUSE	
739	COMPILE-TIME WARNING	`Illegal literal in option: ignored`
	CAUSE	A number was not enclosed in quotation marks, or there was an invalid notation such as a number 9 for an octal literal.

740	COMPILE-TIME WARNING	This language cannot be specified on this operating system: HP FORTRAN 77 assumed
	CAUSE	Illegal language suboption was specified for the $ALIAS directive on host operating system.
741	COMPILE-TIME WARNING	Expecting identifier or "(" in option: ignored
	CAUSE	
742	COMPILE-TIME WARNING	Expecting .AND. or .OR. in option: ignored
	CAUSE	
743	COMPILE-TIME WARNING	System intrinsic file name too long: truncated
	CAUSE	The file name can not exceed the length allowed by the operating system.
744	COMPILE-TIME WARNING	System intrinsic file format may be incorrect for this operating system
	CAUSE	The system intrinsic file is not in the correct format.
745	COMPILE-TIME WARNING	Conditional compilation expression too complicated: .TRUE. assumed
	CAUSE	The limit of the number of conditional expressions has been exceeded.
746	COMPILE-TIME WARNING	Duplicate specification of system intrinsic
	CAUSE	A system intrinsic is specified more than once.

747	COMPILE-TIME WARNING	`Compiler suboption identifier expected`
	CAUSE	This suboption requires an identifier.
748	COMPILE-TIME WARNING	`Expecting identifier or " / " in option: ignored`
	CAUSE	The identifier or "/" expected by the directive must be specified, such a `$EXTERNAL.ALIAS` used with no name specifier.
749	COMPILE-TIME WARNING	`$SEGMENT has been mapped to $LOCALITY on this operating system`
	CAUSE	The $SEGMENT directive used on MPE/V is accepted as a synonym for $LOCALITY, though their memories are not identical.
750	COMPILE-TIME WARNING	`Types of arguments to function do not agree`
	CAUSE	The formal and actual arguments to a function do not match.
752	COMPILE-TIME WARNING	`Nonintrinsic function declared INTRINSIC`
	CAUSE	This function is not in the intrinsic table.
753	COMPILE-TIME WARNING	`Incorrect number of values in DATA list`
	CAUSE	The number of values in DATA list is fewer than or exceeds the number of variable, array and substring elements in the DATA statement.

754	COMPILE-TIME WARNING	Illegal parameter in option: option ignored
	CAUSE	This option cannot have the given parameters.
755	COMPILE-TIME WARNING	Identifier has been truncated to 32 characters
	CAUSE	Identifiers can have only 32 significant characters. Any identifiers longer than 32 characters are truncated to 32.
756	COMPILE-TIME WARNING	Duplicate specification of shared memory option: this option ignored
	CAUSE	The KEY option in `$SHARED_MEMORY` is already specified.
757	COMPILE-TIME WARNING	SHARED_COMMON key has been truncated to maximum length
	CAUSE	The name field for the KEY option in `$SHARED_COMMON` has been exceeded.
758	COMPILE-TIME WARNING	Procedure or function call not compatible with system intrinsic definition
	CAUSE	
759	COMPILE-TIME WARNING	OPTIMIZER detected potential uninitialized variable
	CAUSE	
761	COMPILE-TIME WARNING	ALIAS option not allowed in or after executable statements: option ignored
	CAUSE	All ALIAS options must be specified before executable statements.

762	COMPILE-TIME WARNING	If "PFA" is specified, "SYMDEBUG" cannot be turned off
	CAUSE	
763	COMPILE-TIME WARNING	This option cannot be turned off on this operating system
	CAUSE	Some options are intrinsically ON in the operating system.
764	COMPILE-TIME WARNING	This algebraic expression can be reduced
	CAUSE	
767	COMPILE-TIME WARNING	Duplicate declaration or definition, using first type
	CAUSE	Redeclaration of a variable with a different type.
768	COMPILE-TIME WARNING	"OPTIMIZE" and DEBUG options mutually exclusive: DEBUG option ignored.
	CAUSE	
769	COMPILE-TIME WARNING	"OPTIMIZE" and DEBUG options mutually exclusive: OPTIMIZE ignored.
	CAUSE	
770	COMPILE-TIME WARNING	DESTINATION ARCHITECTURE was previously specified. Directive ignored.
	CAUSE	

771	COMPILE-TIME WARNING	DESTINATION SCHEDULER was previously specified. Directive ignored.
	CAUSE	
772	COMPILE-TIME WARNING	Attempting to pass entire array as value parameter. Arguments specified pass by value in ALIAS Directive. Unexpected results may occur.
	CAUSE	
775	COMPILE-TIME WARNING	"," or ")" expected in option: ignored
	CAUSE	A comma "," or a right parenthesis ")" is expected in the directive.
776	COMPILE-TIME WARNING	"," expected in option: ignored
	CAUSE	
777	COMPILE-TIME WARNING	Array with (*) dimensions cannot be range checked
	CAUSE	Range checking can only be done for arrays with known dimensions.

778	COMPILE-TIME WARNING	`Array reference out of bounds`
	CAUSE	The array is referenced by a subscript value that is out of the declared bounds.
779	COMPILE-TIME WARNING	`Undefined conditional compilation variable: .TRUE. assumed`
	CAUSE	The indentifier(s) used within the condition list in the `$IF` was not `$SET`.
781	COMPILE-TIME WARNING	`Test may fail due to floating point imprecision`
	CAUSE	.EQ. and .NE. operators test for exact bitwise equality, an unlikely result from floating-point operations. Unless this is a FORTRAN 66 program with characters stored in the floating-point items, this is probably a coding error.
782	COMPILE-TIME WARNING	`Unable to load "LANG" environment variable for NLS: proceeding with n-computer`
	CAUSE	The LANG variable has not been set prior to program execution or has been set to an illegal value.
783	COMPILE-TIME WARNING	`Unable to load "NLDATALANG" JCW for NLS: proceeding with n-computer`
	CAUSE	The NLDATALANG variable has not been set prior to program execution or has been set to an illegal value. (On MPE/iX systems only.)
784	COMPILE-TIME WARNING	`Unable to load collation table for NLS: proceeding with n-computer`
	CAUSE	There is no colation table on the system for the specified LANG value.

785	COMPILE-TIME WARNING	`Illegal FORTRAN NLS call: FORTRAN source code must be compiled with -Y`
	CAUSE	The FORTRAN source file was not compiled with the `-Y` option and NLS features were used.
791	COMPILE-TIME WARNING	`No matching IF directive: ignored`
	CAUSE	An `ENDIF` with no matching `IF`.
792	COMPILE-TIME WARNING	`Expecting ENDIF directive: ignored`
	CAUSE	An `IF` with no matching `ENDIF`.
793	COMPILE-TIME WARNING	`Missing or unsupported suboption encountered: Directive ignored.`
	CAUSE	
798	COMPILE-TIME WARNING	`Missing semantics for this option: ignored`
	CAUSE	Internal error. Please notify your HP representative.
799	COMPILE-TIME WARNING	`Error in option: option ignored`
	CAUSE	The option is ignored because of an error in the specification.

ANSI Warnings

801	ANSI WARNING	`ANSI Warning: mixed character and noncharacter data in EQUIVALENCE`
	CAUSE	To be ANSI standard, an entity of type CHARACTER may be equivalenced only with other entities of type CHARACTER.
802	ANSI WARNING	`ANSI Warning: mixed character and noncharacter data in COMMON block`
	CAUSE	To be ANSI standard, all entities in a COMMON block must be of type CHARACTER if a character variable or character array is present in that block.
803	ANSI WARNING	`ANSI Warning: mixed lengths for types of entries`
	CAUSE	The types of entries are of different lengths.
804	ANSI WARNING	`ANSI Warning: item in COMMON block needs to be aligned`
	CAUSE	Item did not align along machine required boundaries producing non-contiguous allocation of space for variables in `COMMON` block.
805	ANSI WARNING	`ANSI Warning: use of octal or hexadecimal constant`
	CAUSE	Octal and hexadecimal constants are HP extensions to ANSI FORTRAN 77.
806	ANSI WARNING	`ANSI Warning: logical operation performed on integer data`
	CAUSE	One of the operands in a logical operation (`.NOT, .AND,` etc.) is integer.

807	ANSI WARNING	`ANSI Warning: use of block DO, DO WHILE, or END DO statement`
	CAUSE	These looping constructs are MIL-STD 1753 and HP extensions to ANSI FORTRAN 77.
808	ANSI WARNING	`ANSI Warning: use of IMPLICIT NONE statement`
	CAUSE	`IMPLICIT NONE` is a MIL-STD 1753 extension to ANSI FORTRAN 77.
809	ANSI WARNING	`ANSI Warning: length specified for noncharacter data type`
	CAUSE	INTEGER*2, INTEGER*4, REAL*4, REAL*8, LOGICAL*1, LOGICAL*2, LOGICAL*4, COMPLEX*8, and COMPLEX*16 data types are HP extensions to ANSI FORTRAN 77.
810	ANSI WARNING	`ANSI Warning: use of DOUBLE COMPLEX data type`
	CAUSE	DOUBLE COMPLEX data items are an HP extension to ANSI FORTRAN 77.
811	ANSI WARNING	`ANSI Warning: use of PROGRAM parameters`
	CAUSE	Parameters appearing in a PROGRAM statement is an HP extension to ANSI FORTRAN 77.
812	ANSI WARNING	`ANSI Warning: more than seven array dimensions`
	CAUSE	It is non-ANSI standard to have more than seven array dimensions.

813	ANSI WARNING	ANSI Warning: FUNCTION, SUBROUTINE, or ENTRY name called recursively
	CAUSE	Allowing program units to call themselves is an HP extension to ANSI FORTRAN 77.
814	ANSI WARNING	ANSI Warning: use of INCLUDE statement
	CAUSE	The INCLUDE statement is a MIL-STD 1753 extension to ANSI FORTRAN 77.
815	ANSI WARNING	ANSI Warning: improper use of CHARACTER*(*)
	CAUSE	The concatenation of character variables of length (*) is non-ANSI standard.
816	ANSI WARNING	ANSI Warning: use of lowercase letters
	CAUSE	Use of lower case letters outside character constants is an HP extension to ANSI FORTRAN 77.
817	ANSI WARNING	ANSI Warning: use of end-of-line comments
	CAUSE	Use of "!" is an extension to the ANSI standard.
818	ANSI WARNING	ANSI Warning: use of name(s) greater than six characters long
	CAUSE	ANSI standard allows names of length six or less.

819	ANSI WARNING	`ANSI Warning: use of underscore or dollar sign in identifier(s)`
	CAUSE	The use of an underscore ("_") or dollar sign ("$") in symbolic names is a non-ANSI standard feature.
820	ANSI WARNING	`ANSI Warning: noncharacter array used as FORMAT specifier`
	CAUSE	Use of non-character arrays as a FORMAT specifier is non-ANSI standard.
821	ANSI WARNING	`ANSI Warning: use of nonstandard intrinsic function`
	CAUSE	ANSI standard allows use of purely integral expression in a computed GOTO statement.
822	ANSI WARNING	`ANSI Warning: noninteger expression in computed GOTO statement`
	CAUSE	
823	ANSI WARNING	`ANSI Warning: use of Hollerith literal`
	CAUSE	Use of Hollerith literals is an extension to the ANSI standard.
824	ANSI WARNING	`ANSI Warning: use of double quotation mark (")`
	CAUSE	The ANSI standard only supports the apostrophe (') as string delimiters.
825	ANSI WARNING	`ANSI Warning: substring extracted from named constant`
	CAUSE	

826	ANSI WARNING	`ANSI Warning: use of nonstandard FORMAT descriptor`
	CAUSE	A format descriptor was used that is an HP extension to ANSI FORTRAN 77.
827	ANSI WARNING	`ANSI Warning: initialization of integer with character data`
	CAUSE	Initialization of integer variables (INTEGER and BYTE) with character data is an extension to the ANSI standard.
828	ANSI WARNING	`ANSI Warning: use of named constant inside a literal`
	CAUSE	Use of named constants in a `COMPLEX` literal is an extension to the ANSI standard.
829	ANSI WARNING	`ANSI Warning: use of I or J suffix with integer constant`
	CAUSE	I and J suffixes used to denote INTEGER*2 and INTEGER*4 constants are HP extensions to ANSI FORTRAN 77.
830	ANSI WARNING	`ANSI Warning: this system-specific feature is not part of HP standard FORTRAN 77`
	CAUSE	This feature is not available on all HP operating systems.
831	ANSI WARNING	`ANSI Warning: branch into IF block`
	CAUSE	It is non-ANSI standard to transfer control into the range of an IF block.

832	ANSI WARNING	`ANSI Warning: use of ENCODE/DECODE`
	CAUSE	ENCODE/DECODE is not part of ANSI FORTRAN 77.
833	ANSI WARNING	`ANSI Warning: use of "BYTE" or "LOGICAL*1" data type`
	CAUSE	BYTE or LOGICAL*1 is not part of ANSI FORTRAN 77.
834	ANSI WARNING	`ANSI Warning: use of dynamic array`
	CAUSE	Use of dynamic arrays is an extension to the ANSI standard.
835	ANSI WARNING	`ANSI Warning: use of variable length specifier`
	CAUSE	A variable length specifier used in a type declaration statement is a non-ANSI standard feature.
836	ANSI WARNING	`ANSI Warning: transfer into the range of a DO loop or IF block`
	CAUSE	It is non-ANSI standard to transfer control into the range of a DO loop or an IF block.
837	ANSI WARNING	`ANSI Warning: use of arithmetic "IF" with two labels`
	CAUSE	An IF statement only contains two labels. The standard requires that three labels be specified.
838	ANSI WARNING	`ANSI Warning: use of nonstandard record specifier following unit`
	CAUSE	The UNIT=num@rec format of specifying records for direct access files is part of FORTRAN 66.

839	ANSI WARNING	`ANSI Warning: use of "TYPE" statement`
	CAUSE	TYPE statement is a non-ANSI standard feature. The PRINT statement is the equivalent ANSI standard statement.
840	ANSI WARNING	`ANSI Warning: data initialization in type declaration statement`
	CAUSE	It is non-ANSI standard to initialize data in a type declaration statement.
841	ANSI WARNING	`ANSI Warning: use of nonstandard "PARAMETER" statement`
	CAUSE	The PARAMETER statement is in FORTRAN 66 format.
842	ANSI WARNING	`ANSI Warning: use of noninteger in integer context`
	CAUSE	A noninteger is found in places where an integer expression is required.
843	ANSI WARNING	`ANSI Warning: use of logical in numeric context. Converted to integer.`
	CAUSE	A logical expression is found in places where a numeric expression is required.
844	ANSI WARNING	`ANSI Warning: mixing logical with numeric type`
	CAUSE	A logical expression is mixed with a numeric expression.

845	ANSI WARNING	ANSI Warning: COMMON variables initialized in non-BLOCK DATA subprograms
	CAUSE	It is non-ANSI standard to initialize COMMON variables in non-BLOCK DATA subprograms.
846	ANSI WARNING	ANSI Warning: no result assigned to function
	CAUSE	
847	ANSI WARNING	ANSI Warning: use of illegal lexical item
	CAUSE	
848	ANSI WARNING	ANSI Warning: use of tab indentation
	CAUSE	
849	ANSI WARNING	ANSI Warning: use of consecutive arithmetic operators
	CAUSE	
850	ANSI WARNING	ANSI Warning: use of nonstandard syntax
	CAUSE	A comma preceding the iolist in a WRITE statement is non-ANSI standard.
851	ANSI WARNING	ANSI Warning: use of %REF or %VAL built-in functions
	CAUSE	

852	ANSI WARNING	ANSI Warning: use of nonstandard EQUIVALENCE
	CAUSE	
853	ANSI WARNING	ANSI Warning: zero passed by value for NULL parameters
	CAUSE	
854	ANSI WARNING	ANSI Warning: use of VIRTUAL statement
	CAUSE	
855	ANSI WARNING	ANSI Warning: use of nonstandard I/O specifier
	CAUSE	
856	ANSI WARNING	ANSI Warning: use of the ACCEPT statement
	CAUSE	
857	ANSI WARNING	ANSI Warning: blank common initialized in block data subprogram
	CAUSE	
858	ANSI WARNING	ANSI Warning: use of quad precision constant
	CAUSE	
861	ANSI WARNING	ANSI Warning: use of record types
	CAUSE	

Note

Errors 5000 and above are internal errors. If you receive one of these errors, please contact your HP service representative and report the details of this message. This means more investigation is required.

Run-Time Errors

900	RUN-TIME ERROR	`Error in format`
	CAUSE	Format specification contains unrecognizable code or string, contains an impossible format (F0.s or G2.9, for example), or a format descriptor describes a field too wide for internal buffers.
	ACTION	Change format specification to proper syntax.
901	RUN-TIME ERROR	`Negative unit number specified`
	CAUSE	Unit number was not greater than or equal to zero.
	ACTION	Use a nonnegative unit number.
902	RUN-TIME ERROR	`Formatted I/O attempted on unformatted file`
	CAUSE	Formatted I/O was attempted on a file opened for unformatted I/O.
	ACTION	Open the file for formatted I/O; do unformatted I/O on this file.
903	RUN-TIME ERROR	`Unformatted I/O attempted on formatted file`
	CAUSE	Unformatted I/O was attempted on a file opened for formatted I/O. Terminals, for example, are opened as formatted files.
	ACTION	Open the file for unformatted I/O; do formatted I/O on this file.

904	RUN-TIME ERROR	`Direct I/O attempted on sequential file`
	CAUSE	Direct operation attempted on sequential file; direct operation attempted on opened file connected to a terminal.
	ACTION	Use sequential operations on this file; open file for direct access; do not do direct I/O on a file connected to a terminal.
905	RUN-TIME ERROR	`Error in list-directed read of logical data`
	CAUSE	Found repeat value, but no asterisk; first character after optional decimal point was not T or F.
	ACTION	Change input data to correspond to syntax expected by list-directed input of logicals; use input statement that corresponds to syntax of input data.
907	RUN-TIME ERROR	`Error in list-directed I/O read of character data`
	CAUSE	Found repeat value, but no asterisk; characters item not delimited by quotation marks.
	ACTION	Change input data to correspond to syntax expected by list-directed input of characters; use input statement that corresponds to syntax of input data.
908	RUN-TIME ERROR	`Could not open file specified`
	CAUSE	Tried to open a file that the system would not allow because: (1) access to the file was denied by the file system due to access restriction, or (2) named file does not exist, or (3) type of access requested is impossible.

`ACTION`	Correct the name to invoke the file intended.

909	RUN-TIME ERROR	`Sequential I/O attempted on direct access file`
	CAUSE	Attempted a BACKSPACE, REWIND, or ENDFILE on a terminal or other device for which these operations are not defined.
	ACTION	Do not use BACKSPACE, REWIND, or ENDFILE.
910	RUN-TIME ERROR	`Access past end of record attempted`
	CAUSE	Tried to do I/O on record of a file past beginning or end of record.
	ACTION	Perform I/O operation within bounds of the record; increase record length.
912	RUN-TIME ERROR	`Error in list I/O read of complex data`
	CAUSE	Problem reading complex data: (1) no left parenthesis and no repeat value, or (2) found repeat value, but no asterisk, or (3) no comma after real part, or (4) no closing right parenthesis.
	ACTION	Change input data to correspond to syntax expected by list-directed input of complex numbers; use input statement corresponding to syntax of input data.
913	RUN-TIME ERROR	`Out of free space`
	CAUSE	Library cannot allocate an I/O block (from an OPEN statement), parse array (for formats assembled at run time), file name string (from OPEN), or characters from list-directed read.
	ACTION	Allocate more free space in the heap area; open fewer files; use FORMAT statements in place of assembling formats at run time in character arrays; read fewer characters. Use the

“MAXSIZE” kernel parameter to change the heap size for the Series 800.

914	RUN-TIME ERROR	`Access of unconnected unit attempted`
	CAUSE	Unit specified in I/O statement has not previously been connected to anything.
	ACTION	Connect unit before attempting I/O on it (that is, OPEN it); perform I/O on another already connected unit.
915	RUN-TIME ERROR	`Read unexpected character`
	CAUSE	Read a character that is not admissible for the type conversion being performed; input a value into a variable that was too large for the type of the variable.
	ACTION	Remove from input data any characters that are illegal in integers or real numbers.
916	RUN-TIME ERROR	`Error in read of logical data`
	CAUSE	An illegal character was read when logical data was expected.
	ACTION	Change input data to correspond to syntax expected when reading logical data; use input statement corresponding to syntax of input data.
917	RUN-TIME ERROR	`Open with named scratch file attempted`
	CAUSE	Executed OPEN statement with STATUS='SCRATCH', but also named the file.
	ACTION	Either open file with STATUS='SCRATCH', or name the file in an OPEN statement.

918	RUN-TIME ERROR	`Open of existing file with STATUS='NEW' attempted`
	CAUSE	Executed OPEN statement with STATUS='NEW', but file already exists.
	ACTION	Use OPEN without STATUS specifier, or with STATUS='OLD', or STATUS='UNKNOWN'.
920	RUN-TIME ERROR	`Open of file connected to different unit attempted`
	CAUSE	Executed OPEN statement with a UNIT specifier that is already associated with a different file name.
	ACTION	Use an OPEN statement with a UNIT specifier that is not connected to a file name; open the connected file to the same unit name.
921	RUN-TIME ERROR	`Unformatted open with BLANK specifier attempted`
	CAUSE	OPEN statement specified with FORM='UNFORMATTED' and BLANK=XX.
	ACTION	Use either FORM='FORMATTED' or BLANK=XX when opening files.
922	RUN-TIME ERROR	`Read on illegal record attempted`
	CAUSE	Attempted to read a record of a formatted or unformatted direct file that is beyond the current end-of-file.
	ACTION	Read records that are within the bounds of the file.

923	RUN-TIME ERROR	`Open with illegal FORM specifier attempted`
	CAUSE	FORM specifier did not begin with “F”, “f”, “U”, or “u”.
	ACTION	Use either ’FORMATTED’ or ’UNFORMATTED’ for the FORM specifier in an OPEN statement.
924	RUN-TIME ERROR	`Close of scratch file with STATUS='KEEP' attempted`
	CAUSE	The file specified in the CLOSE statement was previously opened with ’SCRATCH’ specified in the STATUS specifier.
	ACTION	Open the file with a STATUS other than ’SCRATCH’; do not specify STATUS=’KEEP’ in the CLOSE statement for this scratch file.
925	RUN-TIME ERROR	`Opened with illegal STATUS specifier attempted`
	CAUSE	STATUS specifier did not begin with “O”, “o”, “N”, “n”, “S”, “s”, “U”, or “u”.
	ACTION	Use ’OLD’, ’NEW’, ’SCRATCH’, or ’UNKNOWN’ for the STATUS specifier in OPEN statement.
926	RUN-TIME ERROR	`Close with illegal STATUS specifier attempted`
	CAUSE	STATUS specifier did not begin with “K”, “k”, “D”, or “d”.
	ACTION	Use ’KEEP’ or ’DELETE’ for the STATUS specifier in a CLOSE statement.

927	RUN-TIME ERROR	`Open with illegal ACCESS specifier attempted`
	CAUSE	ACCESS specifier did not begin with “S”, “s”, “D”, or “d”.
	ACTION	Use ’SEQUENTIAL’ or ’DIRECT’ for the ACCESS specifier in an OPEN statement.
929	RUN-TIME ERROR	`Open of direct file with no RECL specifier attempted`
	CAUSE	OPEN statement had ACCESS=’DIRECT’, but no RECL specifier.
	ACTION	Add RECL specifier; specify ACCESS=’SEQUENTIAL’.
930	RUN-TIME ERROR	`Open with RECL less than 1 attempted`
	CAUSE	RECL specifier in OPEN statement was less than or equal to zero.
	ACTION	Use a positive number for RECL specifier in OPEN statement.
931	RUN-TIME ERROR	`Open with illegal BLANK specifier attempted`
	CAUSE	BLANK specifier did not begin with “N”, “n”, “Z”, or “z”.
	ACTION	Use ’NULL’ or ’ZERO’ for BLANK specifier in OPEN statement.
933	RUN-TIME ERROR	`Sequential end-of-file with no END= specifier`
	CAUSE	End-of-file mark read by a READ with no END= specifier indicating a label to which to jump.

`ACTION`	Use the END= specifier to handle the EOF; check logic.

934	RUN-TIME ERROR	OPEN of readonly file with ACCESS='APPEND' ATTEMPTED
	CAUSE	
	ACTION	
936	RUN-TIME ERROR	Append I/O attempted on sequential only file/device
	CAUSE	
	ACTION	
937	RUN-TIME ERROR	Illegal record number specified
	CAUSE	Record number less than one was specified for direct I/O.
	ACTION	Use record numbers greater than zero.
942	RUN-TIME ERROR	Error in list-directed read - character data read for assignment to noncharacter variable
	CAUSE	A character string was read for a numerical or logical variable.
	ACTION	Check input data and input variable type.
944	RUN-TIME ERROR	Record too long in direct unformatted I/O
	CAUSE	Output requested is too long for specified (or preexisting) record length.
	ACTION	Make the number of bytes output by WRITE less than or equal to the file record size.
945	RUN-TIME ERROR	Error in formatted I/O

`CAUSE`	More bytes of I/O were requested than exist in the current record.
`ACTION`	Match the format to the data record.

950	RUN-TIME ERROR	Subscript, substring, or parameter out of bounds at statement number *nnn*
	CAUSE	An index to an array or substring reference was outside of the declared limits at the specified statement number.
	ACTION	Check all indexes to arrays and substrings in and around the given statement number.
951	RUN-TIME ERROR	Label out of bounds in assigned GOTO at statement number *nnn*
	CAUSE	The value of the variable did not correspond to any of the labels in the list in an assigned GOTO statement.
	ACTION	Check for a possible logic error in the program or an incorrect list in the assigned GOTO statement at or near the given statement number.
952	RUN-TIME ERROR	Zero increment value in DO loop at statement number *nnn*
	CAUSE	A DO loop with a zero increment has produced an infinite loop.
	ACTION	Check for a logic error in or near the given statement number.
953	RUN-TIME ERROR	No repeatable format descriptor in format string
	CAUSE	No format descriptor was found to match I/O list items.
	ACTION	Add at least one repeatable format descriptor to format statement.

954	RUN-TIME ERROR	`Illegal use of empty format`
	CAUSE	An empty format, (), was used with list items specified.
	ACTION	Remove items from I/O list; fill in format specifications with appropriate format descriptors.
955	RUN-TIME ERROR	`Open with no FILE= and STATUS='OLD' or 'NEW'`
	CAUSE	OPEN statement is incomplete.
	ACTION	Change status to 'SCRATCH' or 'UNKNOWN' or add file specifier.
956	RUN-TIME ERROR	`File system error`
	CAUSE	The file system returned an error status during an I/O operation.
	ACTION	See the associated file system error message.
957	RUN-TIME ERROR	`Format descriptor incompatible with numeric item in I/O list`
	CAUSE	A numeric item in the I/O list was matched with a nonnumeric format descriptor.
	ACTION	Match format descriptors to I/O list.

958	RUN-TIME ERROR	`Format descriptor incompatible with character item in I/O list`
	CAUSE	A character item in the I/O list was matched with a format descriptor other than "A" or "R".
	ACTION	Match format descriptors to I/O list.
959	RUN-TIME ERROR	`Format descriptor incompatible with logical item in I/O list`
	CAUSE	A logical item in the I/O list was matched with a format descriptor other than "L".
	ACTION	Match format descriptors to I/O list.
960	RUN-TIME ERROR	`Format error: Missing starting left parenthesis`
	CAUSE	Format did not begin with a left parenthesis.
	ACTION	Begin format with left parenthesis.
961	RUN-TIME ERROR	`Format error: Invalid format descriptor`
	CAUSE	Format descriptor did not begin with a character that can start a legal format descriptor.
	ACTION	Specify correct format descriptor.

962	RUN-TIME ERROR	`Unexpected character found following a number in the format string`
	CAUSE	Format error: character in the set IFEDGMNK@OLAR(PHX expected and not found.
	ACTION	Specify correct format descriptor to follow number.
963	RUN-TIME ERROR	`Format error: Trying to scale unscalable format specifier`
	CAUSE	The specifier being scaled is not “F”, “E”, “D”, “M”, “N”, or “G”.
	ACTION	Scale only specifiers for floating-point I/O.
964	RUN-TIME ERROR	`Format error: Parentheses too deeply nested`
	CAUSE	Too many left parentheses for the format processor to stack.
	ACTION	Nest parentheses less deeply.
965	RUN-TIME ERROR	`Format error: Invalid tab specifier`
	CAUSE	A specifier beginning with “T” is not a correct tab specifier.
	ACTION	Correct the specifier beginning with “T”.
966	RUN-TIME ERROR	`Format error: Invalid blank specifier`
	CAUSE	A specifier beginning with “B” did not have “N” or “Z” as the next character.
	ACTION	Correct the specifier beginning with “B”.

967	RUN-TIME ERROR	`Format error: Specifier expected but end of format found`
	CAUSE	The end of the format was reached when another specifier was expected.
	ACTION	Check the end of the format for a condition that would lead the processor to look for another specifier (possibly a missing right parenthesis).
968	RUN-TIME ERROR	`Format error: Missing separator`
	CAUSE	Other specifier found when `/`, `:`, or `)` expected.
	ACTION	Insert separator where needed.
969	RUN-TIME ERROR	`Format error: Digit expected`
	CAUSE	Number not found following format descriptor requiring a field width.
	ACTION	Specify field width where required.
970	RUN-TIME ERROR	`Format error: Period expected in floating point format descriptor`
	CAUSE	No period was found to specify the number of decimal places in an "F", "G", "E", or "D" format descriptor.
	ACTION	Specify the number of decimal places for the field.

971	RUN-TIME ERROR	`Format error: Unbalanced parentheses`
	CAUSE	More right parentheses than left parentheses were found.
	ACTION	Correct format so parentheses balance.
972	RUN-TIME ERROR	`Format error: Invalid string in format`
	CAUSE	String extends past the end of the format or is too long for buffer.
	ACTION	Check for unbalanced quotation mark or for "H" format count too large; or break up long string.
973	RUN-TIME ERROR	`Record length different in subsequent OPEN`
	CAUSE	Record length specified in redundant OPEN conflicted with the value as opened.
	ACTION	Only BLANK= specifier may be changed by a redundant OPEN.
974	RUN-TIME ERROR	`Record accessed past end of internal file record (variable)`
	CAUSE	An attempt was made to transfer more characters than internal file length.
	ACTION	Match READ or WRITE with internal file size.
975	RUN-TIME ERROR	`Illegal new file number requested in fset function`
	CAUSE	The file number requested to be set was not a legal file system file number.
	ACTION	Check that the FOPEN jsucceeded and the file number is correct.

976	RUN-TIME ERROR	`Unexpected character in "NAMELIST" read`
	CAUSE	Unexpected character in NAMELIST read.
	ACTION	Remove illegal character from data.
977	RUN-TIME ERROR	`Illegal subscript or substring in "NAMELIST" read`
	CAUSE	Illegal subscript or substring in NAMELIST read.
	ACTION	Specify only array elements within the bounds of the array being read.
978	RUN-TIME ERROR	`Too many values in "NAMELIST" read`
	CAUSE	Too many values in NAMELIST read.
	ACTION	Supply only as many values as the length of the array.
979	RUN-TIME ERROR	`Variable not in "NAMELIST" group`
	CAUSE	Variable not in NAMELIST group in NAMELIST read.
	ACTION	Read only the variables in this NAMELIST.
980	RUN-TIME ERROR	`"NAMELIST" I/O attempted on unformatted file`
	CAUSE	NAMELIST I/O on unformatted (binary) file.
	ACTION	Use NAMELIST I/O only on formatted files.

981	RUN-TIME ERROR	`Value out of range in numeric read`
	CAUSE	Value read for numeric item is to big/small.
	ACTION	Read only values that fit in the range of the numeric type being read.
989	RUN-TIME ERROR	`Illegal FORTRAN NLS call: FORTRAN source code must be compiled with -Y`
	CAUSE	The FORTRAN source file was not compiled with the `-Y` option and NLS features were used. The problem is critical enough that program execution cannot continue.
	ACTION	Recompile the FORTRAN source code with `-Y` option.
990	RUN-TIME ERROR	`Open with illegal record type specifier`
	CAUSE	
	ACTION	

B

Intrinsic Functions and Math Subroutines

An intrinsic function is a built-in function that returns a single value. A math subroutine is a predefined subroutine that performs a particular mathematical task. Intrinsic functions and math subroutines convert values from one data type to another, perform data manipulation, and also perform basic mathematical functions, such as calculating sines, cosines, and square roots of numbers.

This chapter describes the intrinsic functions and predefined math subroutines of HP FORTRAN 77.

Invoking an Intrinsic Function

An intrinsic function is invoked when the function name and any argument appear in an expression. For example, the statement

```
root = SQRT(value1 + value2)
```

invokes the function SQRT, which computes the square root of `value1 + value2`. The resulting square root is then assigned to the variable `root`.

Some intrinsics, such as MVBITS, are actually subroutines. A subroutine is invoked in the following way:

`CALL` *subroutine_name*

You can define and call your own function subprogram with the same name as an intrinsic function. However, the intrinsic function will be used unless your function is declared as an external function with the EXTERNAL statement. Refer to your HP FORTRAN 77 Programmer's Guide for information on defining your own function subprogram.

Declaring an intrinsic function in a type statement has no effect on the type of the intrinsic function. For example, the statements:

```
INTEGER*4 float
x = float(y)
```

do not change the data type of `float` to INTEGER*4; the type of `float` remains REAL*4. An IMPLICIT statement does not change the type of an intrinsic function either.

Generic and Specific Function Names

Each of the intrinsic functions has a generic name, one or more specific names, or both generic and specific names. A generic name can be used with any of the valid data types for the function. A specific name can be used with only the specified data type. If both a generic and a specific name exist, either can be used to invoke the function. However, generic names are recommended because they are more flexible.

For example, the generic function `COS(`*arg*`)` can have an argument of type REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16. The data type of the result will be the same as that of the argument.

The specific function `DCOS(`*arg*`)` takes a REAL*8 argument. Any other data type is invalid. The data type of the result is REAL*8. Similarly, the specific function `CCOS(`*arg*`)` can take only a COMPLEX*8 argument.

If a function (such as MOD) requires more than one argument, all arguments to that function must be the same *general* data type. For example, INTEGER*2 arguments can be mixed with INTEGER*4 arguments, but integer and real arguments cannot be mixed.

Note, however, that if mixed types are used, INTEGER*2 arguments will be sign-extended when promoted to INTEGER*4 arguments.

If a subroutine (such as MVBITS) requires more than one argument, all arguments to that subroutine must be the same data type. For example, INTEGER*2 arguments cannot be mixed with INTEGER*4 arguments.

If a generic or specific name appears as a formal argument, that name does not identify an intrinsic function in that program unit or statement function. For example, in this subroutine

```
SUBROUTINE sub(log,f)
    .
    .
    .
f = log(f)
END
```

`log` is not an intrinsic function.

Summary of the Intrinsic Functions

This section lists the intrinsic functions of HP FORTRAN 77. Tables B-1 through B-6 show the definition of each function, the number of arguments, the generic name for each group of functions, the specific name for each function, the types of arguments allowed, and the argument and function type. Table B-7 lists the random number generator functions available in FORTRAN.

Table B-8 lists the built-in functions available in FORTRAN.

For complete descriptions of many of the FORTRAN intrinsic functions, see "Function Descriptions".

Table B-1. Arithmetic Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute value	|a|	1	ABS	JIABS†	INTEGER*4	INTEGER*4
				HABS†	INTEGER*2	INTEGER*2
				IIABS†	INTEGER*2	INTEGER*2
				BABS†	LOGICAL*1	LOGICAL*1
				ABS	REAL*4	REAL*4
				DABS	REAL*8	REAL*8
				QABS†	REAL*16	REAL*16
				CABS	COMPLEX*8	REAL*4
				ZABS†	COMPLEX*16	REAL*8
				CDABS†	COMPLEX*16	REAL*8
			IABS	JIABS†	INTEGER*4	INTEGER*4
				HABS†	INTEGER*2	INTEGER*2
				IIABS†	INTEGER*2	INTEGER*2
				BABS†	LOGICAL*1	LOGICAL*1
Remaindering	a–INT(a/b)*b	2	MOD	JMOD†	INTEGER*4	INTEGER*4
				HMOD†	INTEGER*2	INTEGER*2
				IMOD†	INTEGER*2	INTEGER*2
				BMOD†	LOGICAL*1	LOGICAL*1
				AMOD	REAL*4	REAL*4
				DMOD	REAL*8	REAL*8
				QMOD†	REAL*16	REAL*16

† indicates that the function is an extension to the ANSI 77 standard.

(Continued on the next page)

Table B-1. Arithmetic Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Transfer of sign	$\|a\|$ if $b \geq 0$	2	SIGN	JISIGN†	INTEGER*4	INTEGER*4
	$-\|a\|$ if $b<0$			HSIGN†	INTEGER*2	INTEGER*2
				IISIGN†	INTEGER*2	INTEGER*2
				BSIGN†	LOGICAL*1	LOGICAL*1
				----	REAL*4	REAL*4
				DSIGN	REAL*8	REAL*8
				QSIGN†	REAL*16	REAL*16
			ISIGN	JISIGN†	INTEGER*4	INTEGER*4
				HSIGN†	INTEGER*2	INTEGER*2
				IISIGN†	INTEGER*2	INTEGER*2
				BSIGN†	LOGICAL*1	LOGICAL*1
Positive difference	$a-b$ if $a>b$	2	DIM	JIDIM†	INTEGER*4	INTEGER*4
	0 if $a<b$			HDIM†	INTEGER*2	INTEGER*2
				IIDIM†	INTEGER*2	INTEGER*2
				BDIM†	LOGICAL*1	LOGICAL*1
				DIM	REAL*4	REAL*4
				DDIM	REAL*8	REAL*8
				QDIM†	REAL*16	REAL*16
			IDIM	JIDIM†	INTEGER*4	INTEGER*4
				HDIM†	INTEGER*2	INTEGER*2
				IIDIM†	INTEGER*2	INTEGER*2
				BDIM†	LOGICAL*1	LOGICAL*1
REAL*8 product of REAL*4	a*b	2		DPROD	REAL*4	REAL*8
REAL*16 product of REAL*8	a*b	2		QPROD†	REAL*8	REAL*16
Choosing largest value	max (a,b, . . .)	≥ 2	MAX	IMAX0†	INTEGER*2	INTEGER*2
				JMAX0†	INTEGER*4	INTEGER*4
				AMAX1	REAL*4	REAL*4
				DMAX1	REAL*8	REAL*8
				QMAX1†	REAL*16	REAL*16
			MAX0	IMAX0	INTEGER*2	INTEGER*2
				JMAX0	INTEGER*4	INTEGER*4
			MAX1	IMAX1†	REAL*4	INTEGER*2
				JMAX1†	REAL*4	INTEGER*4
			AMAX0	AIMAX0†	INTEGER*2	REAL*4
				AJMAX0†	INTEGER*4	REAL*4

† indicates that the function is an extension to the ANSI 77 standard.

(Continued on the next page)

Table B-1. Arithmetic Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Choosing smallest value	min (a,b, . . .)	≥2	MIN MIN0 MIN1 AMIN0	---- IMIN0† JMIN0† AMIN1 DMIN1 QMIN1† IMIN0 JMIN0 IMIN1† JMIN1† AIMIN0† AJMIN0†	LOGICAL*1 INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 INTEGER*2 INTEGER*4 REAL*4 REAL*4 INTEGER*2 INTEGER*4	LOGICAL*1 INTEGER*2 INTEGER*4 REAL*4 REAL*8 REAL*16 INTEGER*2 INTEGER*4 INTEGER*2 INTEGER*4 REAL*4 REAL*4
Imaginary part of a complex argument	ai	1	IMAG†	AIMAG DIMAG†	COMPLEX*8 COMPLEX*16	REAL*4 REAL*8
Conjugate of a complex argument	(ar,−ai)	1	CONJG	CONJG DCONJG†	COMPLEX*8 COMPLEX*16	COMPLEX*8 COMPLEX*16
Logical product		2	IAND†	JIAND† HIAND† IIAND† BIAND†	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1
Logical sum		2	IOR†	JIOR† HIOR† IIOR† BIOR†	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1
Exclusive OR		2	IXOR† or IEOR†	JIEOR† JIXOR† HIEOR† IIEOR† IIXOR† BIEOR† BIXOR†	INTEGER*4 INTEGER*4 INTEGER*2 INTEGER*2 INTEGER*2 LOGICAL*1 LOGICAL*1	INTEGER*4 INTEGER*4 INTEGER*2 INTEGER*2 INTEGER*2 LOGICAL*1 LOGICAL*1
Complement		1	NOT†	JNOT† HNOT† INOT† BNOT†	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1	INTEGER*4 INTEGER*2 INTEGER*2 LOGICAL*1
† indicates that the function is an extension to the ANSI 77 standard.						

Table B-2. Bit Manipulation Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Bit test		2	BTEST†	BJTEST†	INTEGER*4	LOGICAL*4
				HTEST†	INTEGER*2	LOGICAL*2
				BITEST†	INTEGER*2	LOGICAL*2
				BBTEST†	LOGICAL*1	LOGICAL*1
Bit set		2	IBSET†	JIBSET†	INTEGER*4	INTEGER*4
				HBSET†	INTEGER*2	INTEGER*2
				IIBSET†	INTEGER*2	INTEGER*2
				BBSET†	LOGICAL*1	LOGICAL*1
Bit clear		2	IBCLR†	JIBCLR†	INTEGER*4	INTEGER*4
				HBCLR†	INTEGER*2	INTEGER*2
				IIBCLR†	INTEGER*2	INTEGER*4
				BBCLR†	LOGICAL*1	LOGICAL*1
Bit move		5	MVBITS†	MVBITS†	INTEGER*4	---
				HMVBITS†	INTEGER*2	---
				BMVBITS†	LOGICAL*1	---
Logical shift		2	ISHFT†	JISHFT†	INTEGER*4	INTEGER*4
				HSHFT†	INTEGER*2	INTEGER*2
				IISHFT†	INTEGER*2	INTEGER*2
				BSHFT†	LOGICAL*1	LOGICAL*1
Circular shift		3	ISHFTC†	JISHFTC†	INTEGER*4	INTEGER*4
				HSHFTC†	INTEGER*2	INTEGER*2
				IISHFTC†	INTEGER*2	INTEGER*4
				BSHFTC†	LOGICAL*1	LOGICAL*1
Bit extraction		3	IBITS†	JIBITS†	INTEGER*4	INTEGER*4
				HBITS†	INTEGER*2	INTEGER*2
				IIBITS†	INTEGER*2	INTEGER*4
				BBITS†	LOGICAL*1	LOGICAL*1

† indicates that the function is an extension to the ANSI 77 standard.

Table B-3. Character Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Conversion to character		1	CHAR	---- ---- ----	INTEGER*4 INTEGER*2 LOGICAL*1	CHARACTER CHARACTER CHARACTER
Conversion to INTEGER*4		1		ICHAR	CHARACTER	INTEGER*4
Conversion to INTEGER*2		1		INUM†	CHARACTER	INTEGER*2
Conversion to INTEGER*4		1		JNUM†	CHARACTER	INTEGER*4
Conversion to REAL*4		1		RNUM†	CHARACTER	REAL*4
Conversion to REAL*8		1		DNUM†	CHARACTER	REAL*8
Conversion to REAL*16		1		QNUM†	CHARACTER	REAL*16
Length	Length of character entry	1		LEN	CHARACTER	INTEGER*4
Index of a substring	Location of substring b in string a	2		INDEX	CHARACTER	INTEGER*4
Lexically greater than or equal	a≥b	2		LGE	CHARACTER	LOGICAL*4
Lexically greater than	a>b	2		LGT	CHARACTER	LOGICAL*4
Lexically less than or equal	a≤b	2		LLE	CHARACTER	LOGICAL*4
Lexically less than	a<b	2		LLT	CHARACTER	LOGICAL*4

† indicates that the function is an extension to the ANSI 77 standard.

Table B-4. Numeric Conversion Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to INTEGER*2 and INTEGER*4 using INT(a)	1	INT	----	LOGICAL*1†	INTEGER*4
				IINT†	REAL*4	INTEGER*2
				JINT†	REAL*4	INTEGER*4
				IIDINT†	REAL*8	INTEGER*2
				JIDINT†	REAL*8	INTEGER*4
				IIQINT†	REAL*16	INTEGER*2
				JIQINT†	REAL*16	INTEGER*4
				----	COMPLEX*8	INTEGER*2
				----	COMPLEX*8	INTEGER*4
				----	COMPLEX*16	INTEGER*2
				----	COMPLEX*16	INTEGER*4
			IDINT	IIDINT†	REAL*8	INTEGER*2
				JIDINT†	REAL*8	INTEGER*4
			IQINT†	IIQINT†	REAL*16	INTEGER*2
				JIQINT†	REAL*16	INTEGER*4
Type conversion	Conversion to INTEGER*2 and INTEGER*4; zero-extend using ZEXT(i)	1	ZEXT†[27]			
			IZEXT†		LOGICAL*1†	INTEGER*2
					LOGICAL*2	INTEGER*2
					INTEGER*2	INTEGER*2
			JZEXT†		LOGICAL*1	INTEGER*4
					LOGICAL*2	INTEGER*4
					LOGICAL*4	INTEGER*4
					INTEGER*2	INTEGER*4
					INTEGER*4	INTEGER*4
Type conversion	Conversion to REAL*4	1	REAL	----	LOGICAL*1†	REAL*4
				FLOATI†	INTEGER*2	REAL*4
				FLOATJ†	INTEGER*4	REAL*4
				----	REAL*4	REAL*4
				SNGL	REAL*8	REAL*4
				SNGLQ†	REAL*16	REAL*4
				----	COMPLEX*8	REAL*4
				----	COMPLEX*16	REAL*4
			FLOAT	FLOATI	INTEGER*2	REAL*4
				FLOATJ	INTEGER*4	REAL*4

(Continued on the next page)

Table B-4. Numeric Conversion Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to REAL*8	1	DBLE	----	LOGICAL*1†	REAL*8
				DFLOTI	INTEGER*2	REAL*8
				DFLOTJ	INTEGER*4	REAL*8
				----	REAL*4	REAL*8
				DBLE	REAL*8	REAL*8
				----	REAL*16	REAL*8
				DBLEQ†	COMPLEX*8	REAL*8
				----	COMPLEX*16	REAL*8
			DFLOAT	DREAL†	INTEGER*2	REAL*8
				DFLOTI	INTEGER*4	REAL*8
				DFLOTJ		

† indicates that the function is an extension to the ANSI 77 standard.

(Continued on the next page)

Table B-4. Numeric Conversion Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to REAL*16	1	QEXT†	----	LOGICAL*1	REAL*16
				QFLOTI	INTEGER*2	REAL*16
				QFLOTJ	INTEGER*4	REAL*16
				----	REAL*4	REAL*16
				----	REAL*8	REAL*16
				QEXTD†	REAL*16	REAL*16
				----	COMPLEX*8	REAL*16
				----	COMPLEX*16	REAL*16
			QFLOAT	----	INTEGER*2	REAL*16
				QFLOTI	INTEGER*4	REAL*16
				QFLOTJ	INTEGER*4	REAL*16
Type conversion	Conversion to COMPLEX*8	1 or 2‡	CMPLX	----	LOGICAL*1†	COMPLEX*8
				----	INTEGER*2	COMPLEX*8
				----	INTEGER*4	COMPLEX*8
				----	REAL*4	COMPLEX*8
				----	REAL*8	COMPLEX*8
				----	REAL*16	COMPLEX*8
				----	COMPLEX*8	COMPLEX*8
				----	COMPLEX*16	COMPLEX*8
Type conversion	Conversion to COMPLEX*16	1 or 2‡	DCMPLX†	----	LOGICAL*1†	COMPLEX*16
				----	INTEGER*2	COMPLEX*16
				----	INTEGER*4	COMPLEX*16
				----	REAL*4	COMPLEX*16
				----	REAL*8	COMPLEX*16
				----	REAL*16	COMPLEX*16
				----	COMPLEX*8	COMPLEX*16
				----	COMPLEX*16	COMPLEX*16
Type conversion	Conversion to INTEGER*4	1		ICHAR	CHARACTER	INTEGER*4
Type conversion	Conversion to character	1	CHAR	----	INTEGER*4	CHARACTER
				----	INTEGER*2	CHARACTER
				----	LOGICAL*1†	CHARACTER

† indicates that the function is an extension to the ANSI 77 standard.

‡ if type COMPLEX*8 or COMPLEX*16 is used, there can only be one argument.

(Continued on the next page)

Table B-4. Numeric Conversion Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Truncation	REAL(INT(a))	1	AINT	AINT	REAL*4	REAL*4
	DBLE(INT(a))			DINT	REAL*8	REAL*8
				DDINT†	REAL*8	REAL*8
				QINT†	REAL*16	REAL*16
			IDINT	IIDINT†	REAL*8	INTEGER*2
				JIDINT†	REAL*8	INTEGER*4
			IQINT†	IIQINT†	REAL*16	INTEGER*2
				JIQINT†	REAL*16	INTEGER*4
Nearest whole number	INT(a+.5) if a≥0	1	ANINT	ANINT	REAL*4	REAL*4
				DNINT	REAL*8	REAL*8
	INT(a−.5) if a<0			QNINT†	REAL*16	REAL*16
Nearest integer	INT(a+.5) if a≥0	1	NINT	ININT†	REAL*4	INTEGER*2
				JNINT†	REAL*4	INTEGER*4
	INT(a−.5) if a<0			IIDNNT†	REAL*8	INTEGER*2
				JIDNNT†	REAL*8	INTEGER*4
				IIQNNT†	REAL*16	INTEGER*2
				JIQNNT†	REAL*16	INTEGER*4
			IDNINT	IIDNNT†	REAL*8	INTEGER*2
				JIDNNT†	REAL*8	INTEGER*4
			IQNINT†	IIQNNT†	REAL*16	INTEGER*2
				JIQNNT†	REAL*16	INTEGER*4

† indicates that the function is an extension to the ANSI 77 standard.

Table B-5. Transcendental Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Sine	sin(a)	1	SIN	SIN DSIN QSIN† CSIN ZSIN† CDSIN†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16
Cosine	cos(a)	1	COS	COS DCOS QCOS† CCOS ZCOS† CDCOS†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16
Tangent	tan(a)	1	TAN	TAN DTAN QTAN† CTAN ZTAN†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16
Arcsine	asin(a)	1	ASIN	ASIN DASIN QASIN†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arccosine	acos(a)	1	ACOS	ACOS DACOS QACOS†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arctangent	atan(a)	1	ATAN	ATAN DATAN QATAN†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
	atan(a/b)	2	ATAN2	ATAN2 DATAN2 QATAN2†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16

† indicates that the function is an extension to the ANSI 77 standard.

(Continued on the next page)

Table B-5. Transcendental Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Sine (degree)	sin(a)	1	SIND†	SIND† DSIND† QSIND†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Cosine (degree)	cos(a)	1	COSD†	COSD† DCOSD† QCOSD†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Tangent (degree)	tan(a)	1	TAND†	TAND† DTAND† QTAND†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arcsine (degree)	asin(a)	1	ASIND†	ASIND† DASIND† QASIND†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arccosine (degree)	acos(a)	1	ACOSD†	ACOSD† DACOSD† QACOSD†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Arctangent (degree)	atan(a)	1	ATAND†	ATAND† DATAND† QATAND†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
	atan(a/b)	2	ATAN2D†	ATAN2D† DATAN2D† QATAN2D†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16

† indicates that the function is an extension to the ANSI 77 standard.

(Continued on the next page)

Table B-5. Transcendental Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Hyperbolic sine	sinh(a)	1	SINH	SINH DSINH QSINH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic cosine	cosh(a)	1	COSH	COSH DCOSH QCOSH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic tangent	tanh(a)	1	TANH	TANH DTANH QTANH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic arcsine	asinh(a)	1	ASINH†	ASINH† DASINH† QASINH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic arccosine	acosh(a)	1	ACOSH†	ACOSH† DACOSH† QACOSH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
Hyperbolic arctangent	arccos(a)	1	ATANH†	ATANH† DATANH† QATANH†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
† indicates that the function is an extension to the ANSI 77 standard.						
(Continued on the next page)						

Table B-5. Transcendental Functions (continued)

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Square root	a**(1/2)	1	SQRT	SQRT DSQRT QSQRT† CSQRT ZSQRT† CDSQRT†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16
Exponential	e**a	1	EXP	EXP DEXP QEXP† CEXP ZEXP† CDEXP†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16
Natural logarithm	log(a)	1	LOG	ALOG DLOG QLOG† CLOG ZLOG† CDLOG†	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16	REAL*4 REAL*8 REAL*16 COMPLEX*8 COMPLEX*16 COMPLEX*16
Common logarithm	log10(a)	1	LOG10	ALOG10 DLOG10 QLOG10†	REAL*4 REAL*8 REAL*16	REAL*4 REAL*8 REAL*16
† indicates that the function is an extension to the ANSI 77 standard.						

Table B-6. Miscellaneous Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Byte address	baddress(a)	1	BADDRESS‡	--	Any	INTEGER*4
Byte address		1	%LOC†	----	Any	INTEGER*4
Bytes of storage	SIZEOF	1	SIZEOF	—-	Any except dynamic or assumed-size array.	INTEGER*4
† indicates that the function is an extension to the ANSI 77 standard.						

Table B-7. Built-in Functions

Function	Description	No. of Args.	Generic Name	Specific Name	Type of Argument	Type of Function
Pass argument by reference		1	%REF†	----	Any	----
Pass argument by value		1	%VAL†	----	Any	----
† indicates that the function is an extension to the ANSI 77 standard.						

Notes for Tables B-1 through B-8

1. For a of type INTEGER*4 or INTEGER*2, `INT(a) = a`. For a of type REAL*4 or REAL*8, there are two cases:

   A. If `|a|<1, INT(a)=0`.

   B. If `|a|>1, INT(a)` is the integer whose magnitude is the magnitude of a and whose sign is the same as that of a.

   For example:

   ```
   INT(-3.7) = -3
   ```

   For a of type COMPLEX*8, `INT(a)` is the value obtained by applying the above rule to the real part of a.

   For a of type REAL*4, `IFIX(a)` is the same as `INT(a)`. `IFIX` behaves differently when either NOSTANDARD INTRINSICS or HP9000_300 is on. See chapter 7 for more information.

   If SHORT is on, `IFIX` returns an INTEGER*2; if LONG is on, it returns an INTEGER*4.

2. For a of type REAL*4, `REAL(a)` is a. For a of type INTEGER*4 or REAL*8, `REAL(a)` is as much precision of the significant part of a as a REAL*4 datum can contain. For a of type COMPLEX*8, `REAL(a)` is the real part of a.

   For a of type INTEGER*4, `FLOAT(a)` is the same as `REAL(a)`.

3. For a of type REAL*8, `DBLE(a)` is a. For a of type INTEGER*4 or REAL*4, `DBLE(a)` is as much precision of the significant part of a as a REAL*8 datum can contain. For a of type COMPLEX*8, `DBLE(a)` is as much precision of the significant part of the real part of a as a REAL*8 datum can contain. For a of type COMPLEX*16, `DBLE(a)` is the real part of a.

4. `CMPLX` can have one or two arguments. If there is one argument, it can be of type INTEGER*4, REAL*4, REAL*8, or COMPLEX*8. If there are two arguments, they must both be of the same type and can be of type INTEGER*4, REAL*4, or REAL*8.

   For a of type COMPLEX*8, `CMPLX(a)` is a. For a of type INTEGER*4, REAL*4, or REAL*8, `CMPLX(a)` is the COMPLEX*8 value whose real part is `REAL(a)` and whose imaginary part is 0. For a of type COMPLEX*16, `CMPLX(a)` is:

   ```
   CMPLX(REAL(a), REAL(IMAG(a)))
   ```

   `CMPLX(a,b)` is the COMPLEX*8 value whose real part is `REAL(a)` and whose imaginary part is `REAL(b)`.

   These rules also apply to `DCMPLX`. For a of type COMPLEX*8, `DCMPLX(a)` is:

   ```
   DCMPLX(DBLE(a), DBLE(IMAG(a)))
   ```

5. A COMPLEX*8 value is expressed as an ordered pair of REAL*4s or REAL*8s `(ar,ai)`, where `ar` is the real part and `ai` the imaginary part. `ABS` or `CABS` is defined as:

   ```
   SQRT (ar^+2^- + ai^+2^-)
   ```

6. All angles in trigonometric functions are expressed in radians.
7. `CONJG` is defined as (ar,ai); see note 5.
8. `ISHFT(a,b)` is defined as the value of the first argument (`a`) shifted by the number of bit positions designated by the second argument (`b`). If `b>0`, shift left; if `b<0`, shift right; if `b=0`, no shift. If `b>15` or `b<-15` (`a` is INTEGER*2), or `b>31` or `b<-31` (`a` is INTEGER*4), the result is 0. Bits shifted out from the left or right end are lost, and 0's are shifted in from the opposite end. The type of the result is the same as the type of `a`.
9. `IXOR` and `IEOR` are defined as the bitwise modulo-2 sum (exclusive OR) of the two arguments. That is, if the bits match, the result bit is 1; otherwise, it is 0.
10. `ICHAR` converts from a character to an integer, based on the internal representation of the character. Characters in the ASCII character set have the standard ASCII values.

    The value of `ICHAR(a)` is an integer in the range $0 \leq$ `ICHAR(a)` ≤ 255, where `a` is an argument of type character and length 1.

    If `a` is longer than one character, the first character is used.
11. `INDEX(a,b)` returns an integer value representing the starting position within character string `a` of a substring identical to string `b`. If `b` occurs more than once within `a`, `INDEX(a,b)` returns the starting position of the first occurrence.

    If `b` does not occur in `a`, the value 0 is returned. If `LEN(a) < LEN(b)`, 0 is also returned.
12. `LGE(a,b)` returns the value `true` if `a=b` or if `a` follows `b` in ASCII collating sequence; otherwise it returns the value `false`.

    `LGT(a,b)` returns the value `true` if `a` follows `b` in ASCII collating sequence; otherwise it returns the value `false`.

    `LLE(a,b)` returns the value `true` if `a=b` or if `a` precedes `b` in ASCII collating sequence; otherwise it returns the value `false`.

    `LLT(a,b)` returns the value `true` if `a` precedes `b` in ASCII collating sequence; otherwise it returns the value `false`.

    If the operands for `LGE, LGT, LLE`, and `LLT` are of unequal length, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

    In HP FORTRAN 77, `LGE, LGT, LLE`, and `LLT` behave exactly the same as `.GE., .GT., .LE.`, and `.LT.` because HP computers use the standard ASCII character set. The intrinsic functions should be used for code that might be ported to another system because they always obey the ASCII collating sequence. Using the operators on another system may produce different results. For example:

```
LLT('9','A')
```

is always true. However:

```
('9' .LT. 'A')
```

may be false on some systems.

Note The NLS directive can change the results of these functions. See "NLS Directive" in Chapter 7.

13. As a MIL-STD-1753 standard extension to the ANSI 77 standard, `ISHFTC(a,b,c)` is defined as the right-most `c` bits of the argument `a` shifted circularly `b` places. That is, the bits shifted out of one end are shifted into the opposite end. No bits are lost. The unshifted bits of the result are the same as the unshifted bits of the argument `a`. The absolute value of the argument `b` must be less than or equal to `c`. The argument `c` must be greater than or equal to 1 and less than or equal to 16 if `a` is INTEGER*2, or less than or equal to 32 if `a` is INTEGER*4.

14. The functions `IBITS, BTEST, IBSET, IBCLR,` and `MVBITS` are defined by the MIL-STD-1753 definition, in which bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0. Note that this numbering is not necessarily used in presenting the data formats in the machine-specific supplements to this manual.

15. As a MIL-STD-1753 standard extension to the ANSI 77 standard, bit subfields can be extracted from a field. Bit subfields are referenced by specifying a bit position and a length. Bit positions within a numeric storage unit are numbered from right to left, and the rightmost bit position is numbered 0. Bit fields cannot extend from one numeric storage unit into another numeric storage unit, and the length of a field must be greater than zero.

    The function `IBITS(a,b,c)` extracts a subfield of `c` bits in length from `a`, starting with bit position `b` and extending left `c` bits. The result field is right-justified and the remaining bits set to 0. The value of `b+c` must be less than or equal to 16 if `a` is INTEGER*2, or less than or equal to 32 if `a` is INTEGER*4.

16. As a MIL-STD-1753 standard extension to the ANSI 77 standard, the bit move subroutine `CALL MVBITS (a,b,c,d,e)` moves `c` bits from positions `b` through `b+c-1` of argument `a` to positions `e` through `e+c-1` of argument `d`. The portion of argument `d` not affected by the movement of bits remain unchanged. All arguments are integer expressions, except `d`, which must be a variable or array element. Arguments `a` and `d` are permitted to be the same numeric storage unit. The values of `b+c` and `e+c` must be less than or equal to the lengths of `a` and `b` respectively.

17. As a MIL-STD-1753 standard extension to the ANSI 77 standard, individual bits of a numeric storage unit can be tested and changed with the bit processing routines described in Notes 18, 19, and 20. Each function has two arguments, `a` and `b`, which are integer expressions. `a` specifies the binary pattern. `b` specifies the bit position (rightmost bit is 0).

18. The function `BTEST` is a logical function. The `b`th bit of argument `a` is tested. If it is 1, the value of the function is true; if it is 0, the value is false. If `b` is greater than or equal to 16 or

32 (depending on whether `a` is a 16- or 32-bit element), the result is false.

19. The result of the function `IBSET(a,b)` is equal to the value of `a` with the `b`th bit set to 1. If `b` is greater than or equal to 16 or 32 (depending on whether `a` is a 16- or 32-bit element), the result is `a`.
20. The result of the function `IBCLR(a,b)` is equal to the value of `a` with the `b`th bit set to 0. If `b` is greater than or equal to 16 or 32 (depending on whether `a` is a 16- or 32-bit element), the result is `a`.
21. In `AINT(a)` or `ANINT(a)`, if `INT(a)` or `INT(a+.5)` is outside the range of integers, then these intrinsics return numbers equal to either the most positive or the most negative value (on the particular system) having the same type as the type of their argument, `a`.

22. The ZEXT function expands any fixed-point argument to either an INTEGER*2 or INTEGER*4 without extending the sign of the argument (that is, the high-order bits are set to zero). The generic name ZEXT behaves like IZEXT when the SHORT directive is enabled. ZEXT behaves like JZEXT when SHORT is not enabled.

23. The size of the integer function returned by BADDRESS is system dependent.

24. The argument of SIN, DSIN, COS, DCOS, TAN, or DTAN must be in radians, which are treated as modulo $2^*\pi$. The argument of DSIND, COSD, DCOSD, TAND, or DTAND must be in degrees, which are treated as modulo 360.

25. The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, or DATAN2 is in radians. The result of ASIND, DASIND, ACOSD, DACOSD, ATAND, DATAND, ATAN2D , or DATAN2D is in degrees.

26. MVBITS and SRAND are actually subroutine calls and not functions, and therefore do not have a resulting type.

27. If SHORT is enabled, ZEXT behaves like IZEXT and returns an INTEGER*2 result. If SHORT is not enabled, ZEXT behaves like JZEXT and returns an INTEGER*4 result.

28. Use $NOSTANDARD INTRINSICS for compatibility compatibility with DEC/VAX FORTRAN 77 and other vendors' FORTRAN compilers. Use $HP9000_300 INTRINSICS for series 300 compatibility.

FORTRAN Intrinsic Functions and Subroutines

Following are system intrinsics accessed with the NOSTANDARD SYSTEM compiler directive.

DATE Subroutine

`DATE` returns a string in the form `dd-mmm-yy` (for example, `15-SEP-88`).

`DATE` is called as follows:

```
   CHARACTER*9 DSTRING
   CALL DATE(DSTRING)
```

IDATE Subroutine

`IDATE` returns three integer values representing the current month, day, and year. `IDATE` is called as follows:

```
   INTEGER MONTH,DAY,YEAR
   CALL IDATE(MONTH,DAY,YEAR)
```

`YEAR` returns a two digit number in the range of zero to 99. To obtain a four-digit calendar year, add 1900 to `YEAR`.

EXIT Subroutine

`EXIT` causes a program to terminate as if a STOP statement without an argument was encountered.

Following is the syntax for `EXIT`:

```
   CALL EXIT()
```

The above call to `EXIT` terminates the program and returns control to the operating system.

RAN Function

`RAN` is a general random number generator of the multiplicative congruential type. The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0, exclusive. Following is a call to `RAN`:

```
   Y = RAN(ISEED)
```

where `ISEED` must be an INTEGER*4 constant, variable, or array element.

`RAN` stores a value in `ISEED` and uses it later to calculate the next random number. `RAN` uses the following algorithm to calculate the value and update the seed:

```
   SEED = MOD(69069 * SEED + 1,2 ** 32)
```

`SEED` is a 32-bit number whose high-order 24 bits are converted to a floating-point number and stored in `Y`. `RAN` returns `Y` and stores the new seed in `ISEED`.

SECNDS Function

SECNDS returns the number of seconds elapsed since midnight minus the number of seconds passed in as an argument. SECNDS measures intervals of seconds up to 24 hours. It can handle cases where the start time is before midnight and the end time is after, as long as the interval does not exceed 24 hours.

Example

```
$NOSTANDARD SYSTEM
       PROGRAM benchmark
       REAL*4 TIME0,TIME1
       INTEGER I

       TIME0 = SECNDS (0.0)
       PRINT *,TIME0

C      code to be limited
       F = 0.0
       DO I = 1,100
         F = SIN(REAL(I)) + F
       END DO
C      end code to be timed

       TIME1 = SECNDS (TIME0)
       PRINT *,TIME1
       END
```

The second call to SECNDS returns the seconds elapsed since midnight minus the number of seconds passed in by TIME0.

Note

Functions RAN and SECNDS cannot be used with the $HP3000_16 ON directive. This directive causes the floating-point format to be classic HP 3000 instead of IEEE, and will not be recognized by these functions. The compiler attempts to find a compatibility mode routine for these which does not exist.

An alternative is to use the MPE/iX TIMER intrinsic. For example, to get the number of seconds after midnight you would compile and execute:

```
$STANDARD_LEVEL SYSTEM
      PROGRAM cktimer

      SYSTEM INTRINSIC TIMER
      INTEGER*4 I, T
      T = TIMER()

      I = MOD((T/1000), 86400)
      PRINT *,I
      END
```

where 86400 is the number of seconds per day. The value is an integer.

TIME Subroutine

`TIME` returns a string in the form `hh:mm:ss` (for example, `22:10:30`).

`TIME` is called as follows:

```
$NOSTANDARD SYSTEM
      PROGRAM checktime

      CHARACTER*8 timebuff
      CALL TIME(timebuff)
      PRINT *,timebuff
      END
```

Setting the TZ Environment Variable

To get the correct time, you must set the evironment variable TZ to your local time zone. To set the TZ, use the MPE/iX SETVAR command. For example, the following command sets the time zone to Central Standard Time and Central Daylight Time, which would be correct for Chicago, Illinois:

```
:SETVAR TZ 'CST6CDT'
```

The following table lists some time zones. Check you local time zone to be sure you use the correct one.

Table B-8. Time Zones and TZ Environment Variable Values

TZ Values	Time Zone	Geographic Area
HST10	Hawaiian Standard Time, Hawaiian Daylight Time.	United States: Hawaii.
AST10ADT	Aleutian Standard Time, Aleutian Daylight Time.	United States: Alaska (parts).
YST9YDT	Yukon Standard Time, Yukon Daylight Time.	United States: Alaska (parts).
PST3PDT	Pacific Standard Time, Pacific Daylight Time.	Canada: British Columbia. United States: California, Idaho(parts), Nevada, Oregon (parts), Washington.
MST7MDT	Mountain Standard Time, Mountain Daylight Time.	Canada: Alberta, Saskatchewan (parts). United States: Colorado, Idaho (parts), Kansas (parts), Montana, Nebraska (parts), New Mexico, North Dakota (parts), Oregon (parts), South Dakota (parts), Texas (parts), Utah, Wyoming.
MST7	Mountain Standard Time.	United States: Arizona.
CST6CDT	Central Standard Time, Central Daylight Time.	Canada: Manitoba, Ontario (parts), Saskatchewan (parts). United States: Alabama, Arkansas, Florida (parts), Illinois, Iowa, Kansas, Kentucky (parts), Louisiana, Michigan (parts), Minnesota, Mississippi, Missouri, Nebraska, North Dakota, Oklahoma, South Dakota, Tennessee (parts), Texas, Wisconsin.
EST6CDT	Eastern Standard Time, Central Daylight Time.	United States: Indiana (most).
EST5EDT	Eastern Standard Time, Eastern Daylight Time.	Canada: Ontario (parts), Quebec (parts). United States: Connecticut, Delaware, District of Columbia, Florida, Georgia, Kentucky, Maine, Maryland, Massachusetts, Michigan, New Hampshire, New Jersey, New York, North Carolina, Ohio, Pennsylvania, Rhode Island, South Carolina, Tennessee (parts), Vermont, Virginia, West Virginia.
AST4ADT	Atlantic Standard Time, Atlantic Daylight Time.	Canada: Newfoundland (parts), Nova Scotia, Prince Edward Island, Quebec (parts).
NST3:30NDT	Newfoundland Standard Time, Newfoundland Daylight Time.	Canada: Newfoundland (parts).
WET0WETDST	Western European Time, Western European Time Daylight Savings Time.	Great Britain, Ireland.
PWT0PST	Portuguese Winter Time, Portuguese Summer Time	
MEZ-1MESZ	Mitteleuropaeische Zeit, Mitteleuropaeische Sommerziet.	
MET-1METDST	Middle European Time, Middle European Time Daylight Savings Time.	Belgium, Luxembourg, Netherlands, Denmark, Norway, Austria, Poland, Czechoslovakia, Sweden, Switzerland, Germany, France, Spain, Hungary, Italy, Yugoslavia.

Table B-8. Time Zones and TZ Environment Variable Values (continued)

TZ Values	Time Zone	Geographic Area
SAST-2SADT	South Africa Standard Time, South Africa Daylight Time.	South Africa.
JST-9	Japan Standard Time	Japan.
WST-8:00	Australian Western Standard Time	Australia: Western Australia
CST-9:30	Australian Central Standard Time	Australia: Northern Territory.
CST-9:30CDT	Australian Central Standard Time, Australian Central Daylight Time.	Australia: South Australia
EST-10	Australian Eastern Standard Time	Australia: Queensland.
EST-10EDT	Australian Eastern Standard Time, Australian Eastern Daylight Time	Australia: New South Wales, Tasmania, Victoria.
NZT-12NZDT	New Zealand Standard Time, New Zealand Daylight Time.	New Zealand

If TZ is not set, time assumes Eastern Standard Time (EST5EDT).

The time differential is automatically adjusted for daylight savings time according to the values in the time and zone adjustment table (the file TZTAB.LIB.SYS).

Note

Make sure your system administrator has correctly set the hardware clock. The hardware clock must be set to Greenwich Mean Time (Universal Coordinated Time or UTC) for TIME to return the correct local time.

Function Descriptions

The FORTRAN 77 generic functions follow, in alphabetical order. The functions that do not have a generic name are listed alphabetically by specific name.

ABS Function

`ABS(`*arg*`)` is a generic function that returns the absolute value of an INTEGER*4, REAL*4, REAL*8, REAL*16, or COMPLEX*8 argument. A complex value is expressed as an ordered pair of REAL*4 or REAL*8 numbers in the form (*ar,ai*) where *ar* is the REAL*4 part and *ai* is the imaginary part. If *arg* is COMPLEX*8, `ABS(`*arg*`)` is equal to the square root of `(`*ar*`**2 + `*ai*`**2)`. For INTEGER*4, REAL*4, REAL*8, and REAL*16 arguments, the result is the same data type as the argument. For COMPLEX*8 arguments, the result is REAL*4.

Examples

Function Call	Value Returned to `a`
`a = ABS(100)`	`100`
`a = ABS(-100.0)`	`100.0`
`a = ABS(vector)`	`28.32716`, where `vector = (12.84,25.25)`
`a = ABS(-1.23451234512345D2)`	`123.451234512345`

The specific function names are ABS for REAL*4 arguments, CABS for COMPLEX*8 arguments, DABS for REAL*8 arguments,

QABS for REAL*16 arguments,

IABS and JIABS for INTEGER*4 arguments, HABS and IIABS for INTEGER*2 arguments, and ZABS and CDABS for COMPLEX*16 arguments. IABS can also be used as a generic name for HABS, and accept INTEGER*2 arguments.

ACOS Function

`ACOS(`*arg*`)` is a generic function that returns the arccosine of a REAL*4, REAL*8, or

REAL*16 argument.

The value of *arg* must be less than or equal to one. The result is expressed in radians and is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ACOS(0.0628)`	`1.5079550`
`a = ACOS(0.0)`	`1.5707964`
`a = ACOS(0.0D0)`	`1.570796326794897`

The specific function names are ACOS for REAL*4 arguments, DACOS for REAL*8 arguments, and QACOS for REAL*16 arguments.

ACOSD Function

`ACOSD(`*arg*`)` is a generic function that returns the arccosine of a REAL*4, REAL*8, or REAL*16 argument. The value of *arg* must be less than or equal to one. The result is expressed in degrees and is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ACOSD(0.2)`	`11.5370`
`a = ACOSD(1.0)`	`0.0`
`a = ACOSD(0.8D0)`	`36.86989764584401`

The specific function name is ACOSD for REAL*4 arguments, DACOSD for REAL*8 arguments, and QACOSD for REAL*16 arguments.

ACOSH Function

`ACOSH(`*arg*`)` is a generic function that returns the hyperbolic arccosine of a REAL*4, REAL*8, or REAL*16 argument.

The argument must be greater than or equal to one and less than or equal to the maximum number allowed on your system. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ACOSH(1.2)`	`0.622362`
`a = ACOSH(1.0)`	`0.0`
`a = ACOSH(1.0D0)`	`0.0`

The specific function names are ACOSH for REAL*4 arguments, DACOSH for REAL*8 arguments,

and QACOSH for REAL*16 arguments.

AINT Function

`AINT(`*arg*`)` is a generic function that truncates the fractional digits from a REAL*4, REAL*8, or REAL*16 argument. The result is the the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = AINT(324.7892)`	`324.0`
`a = AINT(324.7892D2)`	`32478.0`

The specific function names are AINT for REAL*4 arguments; DINT and DDINT for REAL*8 arguments;

and QINT for REAL*16 arguments.

If *arg* is less than one, the result is zero. If *arg* is greater than one, the result is the value with the same sign as *arg* with a magnitude that does not exceed *arg*.

ANINT Function

`ANINT(`*arg*`)` is a generic function that returns the nearest whole number. The argument can be REAL*4, REAL*8, or REAL*16. The result is `INT (`*arg* `+ 0.5)` if *arg* is positive or zero, and is `INT(`*arg* `- 0.5)` if *arg* is negative. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = ANINT(-678.44)`	`-678.0`
`a = ANINT(678.44)`	`678.0`
`a = ANINT(0.00)`	`0.0`
`a = ANINT(6.78 D1)`	`68.0`

The specific function names are ANINT for REAL*4 arguments, DNINT for REAL*8 arguments, and QNINT for REAL*16 arguments.

ASIN Function

`ASIN(`*arg*`)` is a generic function that returns the arcsine of a REAL*4, REAL*8, or REAL*16 argument. The value of *arg* must be less than or equal to one. The result is expressed in radians and is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = ASIN(0.30)`	`0.304692`
`a = ASIN(0.30D0)`	`0.3046926540153975`

The specific function names are ASIN for REAL*4 arguments, DASIN for REAL*8 arguments, and QASIN for REAL*16 arguments.

ASIND Function

`ASIND(`*arg*`)` is a generic function that returns the arcsine of a REAL*4, REAL*8, or REAL*16 argument. The value of *arg* must be less than or equal to one. The result is expressed in degrees and is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ASIND(0.30)`	`17.4576`
`a = ASIND(0.30D0)`	`17.45760312372209`

The specific function names are ASIND for REAL*4 arguments, DASIND for REAL*8 arguments, and QASIND for REAL*16 arguments.

ASINH Function

`ASINH(`*arg*`)` is a generic function that returns the hyperbolic arcsine of a REAL*4, REAL*8, or REAL*16 argument; the result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ASINH(0.30)`	`0.2956731`
`a = ASINH(0.30D0)`	`0.295673047563423`

The specific function names are ASINH for REAL*4 arguments, DASINH for REAL*8 arguments, and QASINH for REAL*16 arguments.

ATAN Function

`ATAN(`*arg*`)` is a generic function that returns the arctangent of a REAL*4, REAL*8, or REAL*16 argument. The result is expressed in radians and is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = ATAN(1.0)`	`0.7853982`
`a = ATAN(3.141592653D0)`	`1.262627255624651`

The specific function names are ATAN for REAL*4 arguments, DATAN for REAL*8 arguments, and QATAN for REAL*16 arguments.

ATAN2 Function

ATAN2(*arg1*`,`*arg2*) is a generic function that returns the arctangent of *arg1*`/`*arg2*. The arguments can be REAL*4, REAL*8, or REAL*16. The result is expressed in radians and is the same data type as the arguments. The arguments cannot both be zero.

Examples

Function Call	Value Returned to `a`
`a = ATAN2(1.0, 2.0)`	`0.4636476`
`a = ATAN2(3.141592653D0, 1.0D0)`	`1.262627255624651`

The specific function names are ATAN2 for REAL*4 arguments, DATAN2 for REAL*8 arguments,

and QATAN2 for REAL*16 arguments.

ATAND Function

`ATAND(`*arg*`)` is a generic function that returns the arctangent of a REAL*4, REAL*8, or REAL*16 argument. The result is expressed in degrees and is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = ATAND(1.0)`	`45.0`
`a = ATAND(2.5D0)`	`68.19859051364820`

The specific function names are ATAND for REAL*4 arguments, DATAND for REAL*8 arguments, and QATAND for REAL*16 arguments.

ATAN2D Function

ATAN2D(*arg1*`,`*arg2*`)` is a generic function that returns the arctangent of *arg1*`/`*arg2*. The arguments can be REAL*4, REAL*8, or REAL*16. The result is expressed in degrees and is the same data type as the arguments. The arguments cannot both be zero.

Examples

Function Call	**Value Returned to** `a`
`a = ATAN2D(1.0, 2.0)`	`26.5651`
`a = ATAN2D(2.0D0, 1.0D0)`	`63.43494882292201`

The specific function names are ATAN2D for REAL*4 arguments, DATAN2D for REAL*8 arguments, and QATAN2D for REAL*16 arguemnts.

ATANH Function

`ATANH(`*arg*`)` is a generic function that returns the hyperbolic arctangent of a REAL*4, REAL*8, or REAL*16 argument. The value of *arg* must be less than one. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = ATANH(0.30)`	`0.3095196`
`a = ATANH(0.30D0)`	`0.309519604203112`

The specific function names are ATANH for REAL*4 arguments, DATANH for REAL*8 arguments, and QATANH for REAL*16 arguments.

BADDRESS Function

`BADDRESS(`*arg*`)` is a generic function that returns the byte address of *arg* as an integer of the same size as the address. *arg* may be of any type. `BADDRESS(`*arg*`)` may not be passed as an actual argument.

Examples

Function Call	Value Returned to `i`
`i = BADDRESS(i)`	Address of integer variable `i`.
`i = BADDRESS(char)`	Address of CHARACTER*9 variable `char`.
`i = BADDRESS(log)`	Address of logical variable `log`.
`i = BADDRESS(cmplxarr)`	Address of complex array `cmplxarr`.

There are no specific function names for BADDRESS.

As an extension to the FORTRAN 77 standard, `%LOC` returns the address of *arg* the same as `BADDRESS`. *arg* can be a variable, an array element, an array, a character substring, or external procedure.

BTEST Function

`BTEST(`*arg1, arg2*`)` is a generic function that tests individual bits of storage. The arguments are INTEGER*4 and the result is LOGICAL*4. If the *arg2*th bit of *arg1* is equal to one, the result is true. If the *arg2*th bit is equal to zero, the result is false. If *arg2* is greater than or equal to the bit size of *arg1*, the result is false. Bit positions are numbered right to left, with the rightmost bit numbered zero.

Examples

Function Call	**Value Returned to** `i`
`i = BTEST(3,0)`	`.TRUE.`
`i = BTEST(0,0)`	`.FALSE.`
`i = BTEST(0,3)`	`.FALSE.`
`i = BTEST(4,1)`	`.FALSE.`

The specific function names are BTEST and BJTEST for INTEGER*4 arguments and HTEST and BITEST for INTEGER*2 arguments.

CHAR Function

`CHAR(`*i*`)` is a specific function that returns the character value in the *i*th position of the ASCII collating sequence. The argument is INTEGER*4 and the result is character.

Examples

Function Call	**Value Returned to** `c`
`c = CHAR(97)`	`'a'`
`c = CHAR(122)`	`'z'`
`c = CHAR(53)`	`'5'`

There is no generic name for this function.

CMPLX Function

`CMPLX(`*arg*`)` (or `CMPLX(`*arg1*`,`*arg2*`)`) is a specific function that performs type conversion to a COMPLEX*8 value. CMPLX can have one or two arguments.

If you specify one argument, the argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, or COMPLEX*8.

If you specify two arguments, the arguments must be the same type and both must be INTEGER*4, REAL*4, REAL*8, or REAL*16.

If only one argument is used and it is not of type COMPLEX*8, the result is COMPLEX*8, with `REAL(`*arg*`)` used as the real part and the imaginary part equal to zero. For one argument of type COMPLEX*8, the result is the same as the argument, or as much of the argument that can fit in a COMPLEX*8 variable. For two arguments, *arg1* and *arg2*, the result is COMPLEX*8, with `REAL(`*arg1*`)` used as the real part and `REAL(`*arg2*`)` used as the imaginary part.

Examples

Function Call	Value Returned to `c`
`c = CMPLX(1.0)`	`(1.0, 0.0)`
`c = CMPLX(1.0, 1.0)`	`(1.0, 1.0)`
`c = CMPLX(1, 0)`	`(1.0, 0.0)`
`c = CMPLX(3.141592653D0, 0.0D0)`	`(3.1415927, 0.0)`

There are no specific names for this function.

CONJG Function

`CONJG(`*arg*`)` is a generic function that returns the conjugate of a COMPLEX*8 or COMPLEX*16 argument. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = CONJG(var1)`	`(3.0, 0.0), where var1 = (3.0, 0.0)`
`a = CONJG(var2)`	`(3.0, -1.0), where var2 = (3.0, 1.0)`

The specific function names are CONJG for COMPLEX*8 arguments and DCONJG for COMPLEX*16 arguments.

COS Function

`COS(`*arg*`)` is a generic function that returns the cosine of a REAL*4, REAL*8,

REAL*16,

or COMPLEX*8 argument. The argument is expressed in radians. The result is the same data type as the argument.

Examples

Function Call	Value Returned to a
`a = COS(0.0)`	`1.0`
`a = COS(0.0628)`	`0.9980288`

The specific function names are COS for REAL*4 arguments, CCOS for COMPLEX*8 arguments, DCOS for REAL*8 arguments, QCOS for REAL*16 arguments, and ZCOS and CDCOS for COMPLEX*16 arguments.

COSD Function

`COSD(`*arg*`)` is a generic function that returns the cosine of a REAL*4, REAL*8, or REAL*16. The argument is expressed in degrees. The result is the same data type as the argument.

Examples

Function Call	Value Returned to a
`a = COSD(60.0)`	`.50000`
`a = COSD(0.0628D0)`	`.9999993993188778`

The specific function names are COSD for REAL*4 arguments, DCOSD for REAL*8 arguments, and QCOSD for REAL*16 arguments.

COSH Function

`COSH(`*arg*`)` is a generic function that returns the hyperbolic cosine of a REAL*4, REAL*8, or REAL*16 argument. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = COSH(1.0)`	`1.5430807`
`a = COSH(3.0)`	`10.06766`
`a = COSH(3.0D0)`;;	`10.0676619957778`

The specific function names are COSH for REAL*4 arguments, DCOSH for REAL*8 arguments,

and QCOSH for REAL*16 arguments.

DBLE Function

`DBLE(`*arg*`)` is a generic function that converts the argument to REAL*8. The argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, or COMPLEX*8. For a REAL*16 argument, the result is as much precision of *arg* as a REAL*8 item can contain. For an INTEGER*4 or REAL*4 argument, the result is as much precision of the significant part of the argument as the argument can provide. For a REAL*8 argument, the result is the argument. For a REAL*16 argument, the result is as much precision of the significant part of *arg* as a REAL*8 item can contain. For a COMPLEX*8 argument, the result is as much precision of the significant REAL*4 part of the argument as the argument can provide.

Examples

Function Call	**Value Returned to** `a`
`a = DBLE(4)`	`4.0`
`a = DBLE(4.0)`	`4.0`
`a = DBLE(4.0D2)`	`400.0`
`a = DBLE(var1)`	`4.0, where var1 = (4.00, 2)`

The specific function names are DFLOAT for INTEGER*4 arguments and DBLEQ for REAL*16 arguments.

DCMPLX Function

`DCMPLX(`*arg*`)` (or `DCMPLX(`*arg1*`,`*arg2*`)`) is a generic function that performs type conversion to a COMPLEX*16 value. DCMPLX can have one or two REAL*8 arguments. If you specify one argument, the argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, or COMPLEX*8. If you specify two arguments, the arguments must be of the same type and both must be INTEGER*4, REAL*4, REAL*8, or REAL*16.

For one argument not of type COMPLEX*8, the result is COMPLEX*16, with `DBLE(`*arg*`)` used as the real part and the imaginary part equal to zero. For one argument, of type COMPLEX*8, the result is the same as the argument. For two arguments, *arg1* and *arg2*, the result is COMPLEX*16, with DBLE(*arg1*) used as the real part and DBLE(*arg2*) used as the imaginary part.

Examples

Function Call	Value Returned to `c`
`c = DCMPLX(1.0)`	`(1.0, 0.0)`
`c = DCMPLX(1.0, 0.0)`	`(1.0, 0.0)`
`c = DCMPLX(3.141592653D0, 0.0D0)`	`(3.1415927, 0.0)`

There are no specific names for this function.

DIM Function

`DIM(`*arg1*`,`*arg2*`)` is a generic function that returns a positive difference. The arguments must be the same data type and can be INTEGER*4, REAL*4, REAL*8, or REAL*16. The result is the same type as the arguments. The result is (*arg1* - *arg2*) if *arg1* is greater than *arg2*. The result is zero if *arg1* is less than or equal to *arg2*.

Examples

Function Call	Value Returned to `a`
`a = DIM(56.9, 45.4)`	`11.5`
`a = DIM(45.4, 56.9)`	`0.0`
`a = DIM(45.4, 45.4)`	`0.0`

The specific function names are DIM for REAL*4 arguments, DDIM for REAL*8 arguments, QDIM for REAL*16 arguments, IDIM and JIDIM for INTEGER*4 arguments, and HDIM and IIDIM for INTEGER*2 arguments. IDIM can also be used as a generic name of HDIM, and accept INTEGER*2 arguments.

DNUM Function

DNUM(*arg*) is a specific function that returns the REAL*8 value represented in the character string *arg*.

Blanks are not significant in the input string.

Examples

Function Call	**Value Returned to** `d8`
`d8 = DNUM('123.5')`	`123.5D0`
`d8 = DNUM('-99.25')`	`-99.25D0`
`d8 = DNUM('327.125E75')`	`327.125D75`
`d8 = DNUM('   24  5 ')`	`245D0` (blanks are ignored)

There is no generic name for this function.

DPROD Function

DPROD(*arg1*,*arg2*) is a specific function that returns the REAL*8 product of two REAL*4 arguments (*arg1* **arg2*). The result is a REAL*8 number with none of the fractional portion lost and is equal to DBLE(*arg1*) * DBLE(*arg2*).

Examples

Function Call	**Value Returned to** `d`
`d = DPROD(2.2, 2.2)`	`4.840`
`d = DPROD(1.0, 2.0)`	`2.0`

There is no generic name for this function.

EXP Function

`EXP(`*arg*`)` is a generic function that returns an exponential result (*e**arg*). The argument can be REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = EXP(3.0)`	`20.08554`
`a = EXP(1.5D1)`	`(3269017.37247211)`
`a = EXP(var)`	`(10.85226, 16.90140)`, where `var = (3.0, 1.0)`

The specific function names are EXP for REAL*4 arguments, DEXP for REAL*8 arguments, QEXP for REAL*16 arguments, CEXP for COMPLEX*8 arguments, and ZEXP and CDEXP for COMPLEX*16 arguments.

IAND Function

`IAND(`*arg1*`,`*arg2*`)` is a generic function that returns the logical product, or bitwise AND, of two INTEGER*4 arguments. The result is INTEGER*4.

Examples

Function Call	Value Returned to `i`
`i = IAND(0, 0)`	`0`
`i = IAND(1, 0)`	`0`
`i = IAND(0, 1)`	`0`
`i = IAND(1, 1)`	`1`

The specific function names are IAND and JIAND for INTEGER*4 arguments and HIAND and IIAND for INTEGER*2 arguments.

IBCLR Function

`IBCLR(`*arg1*`,`*arg2*`)` is a generic function that returns *arg1* with the *arg2*th bit cleared (set to zero). If *arg2* is greater than or equal to the bit size of *arg1*, the result is equal to *arg1*. The arguments and result are INTEGER*4.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered zero.

Examples

Function Call	**Value Returned to** `i`
`i = IBCLR(3, 4)`	`3`
`i = IBCLR(1, 2)`	`1`
`i = IBCLR(1, 0)`	`0`

The specific function names are IBCLR and JIBCLR for INTEGER*4 arguments and HBCLR and IIBCLR for INTEGER*2 arguments.

IBITS Function

`IBITS(`*arg1, arg2, arg3*`)` is a generic function that extracts a subfield of *arg3* bits in length from *arg1*, starting with bit position *arg2* and extending left *arg3* bits. The arguments and result are INTEGER*4. The extracted bits are right-justified in the result with the remaining bits set to zero. The value of (*arg2* + *arg3*) must be less than or equal to 16 if *arg1* is INTEGER*2, or 32 if *arg1* is INTEGER*4.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered zero.

Examples

Function Call	**Value Returned to** `i`
`i = IBITS(3, 4, 8)`	`0`
`i = IBITS(16, 4, 8)`	`1`
`i = IBITS(12, 2, 2)`	`3`

The specific function names are IBITS and JIBITS for INTEGER*4 arguments and HBITS and IIBITS for INTEGER*2 arguments.

IBSET Function

`IBSET(`*arg1*`,`*arg2*`)` is a generic function that returns the value of *arg1* with the *arg2*th bit set to 1. If *arg2* is greater than or equal to the bit size of *arg1*, the result is *arg1*. The arguments and the result are INTEGER*4.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered zero.

Examples

Function Call	**Value Returned to** `i`
`i = IBSET(3, 4)`	`19`
`i = IBSET(1, 2)`	`5`
`i = IBSET(1, 0)`	`1`

The specific function names are IBSET and JIBSET for INTEGER*4 arguments and HBSET and IIBSET for INTEGER*2 arguments.

ICHAR Function

`ICHAR(`*arg*`)` is a specific function that converts a character argument to an INTEGER*4 value. The result depends on the collating position of the argument in the ASCII collating sequence. If *arg* is longer than one character, the first character is used.

Examples

Function Call	**Value Returned to** `i`
`i = ICHAR('a')`	`97`
`i = ICHAR('z')`	`122`
`i = ICHAR('5')`	`53`

There is no generic name for this function.

IEOR Function

`IEOR(`*arg1*`,`*arg2*`)` is a generic function that returns the bitwise exclusive OR of two INTEGER*4 arguments. The result is INTEGER*4.

Examples

Function Call	**Value Returned to** `i`
`i = IEOR(1, 0)`	`1`
`i = IEOR(1, 1)`	`0`
`i = IEOR(0, 0)`	`0`

An alternate generic function name is IXOR. The specific function names are IEOR and JIEOR for INTEGER*4 arguments and HIEOR and IIEOR for INTEGER*2 arguments.

IMAG Function

`IMAG(`*arg*`)` is a generic function that returns the imaginary part of a complex number. The argument can be COMPLEX*8 or COMPLEX*16. For COMPLEX*8 arguments, the result is REAL*4; for COMPLEX*16 arguments, the result is REAL*8. A complex number is expressed as an ordered pair of REAL*4 or REAL*8 numbers in the form (*ar*,*ai*), where *ar* is the REAL*4 part and *ai* is the imaginary part. The result is the REAL*4 value of *ai*.

Examples

Function Call	**Value Returned to** `a`
`a = IMAG(var1)`	`(0.00, where var1 = (25.058, 0.0)`
`a = IMAG(var2)`	`(3.5, where var2 = (25.3, 3.5)`
`a = IMAG(var3)`	`(3.5, where var3 = (25.3D0, 3.5D0)`

The specific function names are AIMAG for COMPLEX*8 arguments, and DIMAG for COMPLEX*16 arguments.

INDEX Function

`INDEX(`*arg1*`,`*arg2*`)` is a specific function that returns the location of substring *arg2* within string *arg1*. Both arguments must be character strings. If string *arg2* occurs as a substring within string *arg1*, the result is an INTEGER*4 indicating the starting position of the substring *arg2* within *arg1*. The character positions are numbered from left to right with the leftmost character numbered 1. If *arg2* does not occur as a substring, the result is zero. If *arg2* occurs more than once within *arg1*, the result is the starting position of the first occurrence. If the length of *arg1* is less than the length of *arg2*, the result is zero.

Examples

Function Call	**Value Returned to** `i`
`i = INDEX('ABCD', 'BC')`	`2`
`i = INDEX('10552', '5')`	`3`
`i = INDEX('ABC', 'XY')`	`0`
`i = INDEX('ABC', 'abc')`	`0`

There is no generic name for this function.

INT Function

`INT(`*arg*`)` is a generic function that converts data types to INTEGER*4. The argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, or COMPLEX*8; the result is INTEGER*4. If *arg2* exceeds the largest integer allowed, the result is undefined.

If *arg* is an INTEGER*4, `INT(`*arg*`)` = *arg*. If *arg* is REAL*4, REAL*8, or REAL*16 and *arg* is less than one, the result is zero. If *arg* is greater than one, the result of INT (*arg*) is the INTEGER*4 with the same sign as arg whose magnitude does not exceed *arg*. If *arg* is COMPLEX*8, the real part of *arg* is used and the result is found by applying the rules to the real part of *arg*.

Examples

Function Call	**Value Returned to `i`**
`i = INT(-3.7)`	`-3`
`i = INT(25)`	`25`
`i = INT(25.9D0)`	`25`
`i = INT(var)`	`30`, where `var = (30.57, 0.0)`

The specific function names are IINT, JINT, IFIX, and JIFIX for REAL*4 arguments, IIDINT and JIDINT for REAL*8 arguments, and IIQINT and JIQINT for REAL*16 arguments. IDINT can be used as a generic for REAL*8 arguments. IQINT can be used as a generic forREAL*16 arguments. IDINT can be used as a generic for REAL*8 arguments. IQINT can be used as a generic for REAL*16 arguments. `INT`, `IQINT`, `IFIX`, and `IDINT` behave differently when either NOSTANDARD INTRINSICS or HP9000_300 is on. See chapter 7 for more information.

INUM Function

`INUM(`*arg*`)` is a specific function that returns the INTEGER*2 value represented in the character string *arg*.

Blanks are not significant in the input string.

Examples

Function Call	Value Returned to `i2`
`i2 = INUM('123')`	`123`
`i2 = INUM('-99')`	`-99`
`i2 = INUM('32767')`	`32767`
`i2 = INUM('   24  ')`	`24` (blanks are ignored)

There is no generic name for this function

IOR Function

`IOR(`*arg1* `,`*arg2*`)` is a generic function that returns the logical (bitwise) sum (Boolean OR) of two INTEGER*4 arguments. The result is INTEGER*4.

Examples

Function Call	Value Returned to `i`
`i = IOR(1, 1)`	`1`
`i = IOR(0, 0)`	`0`
`i = IOR(1, 0)`	`1`

The specific function names are IOR and JIOR for INTEGER*4 arguments and HIOR and IIOR for INTEGER*2 arguments.

ISHFT Function

`ISHFT(`*arg1*`,`*arg2*`)` is a generic function that returns the value of *arg1* shifted by *arg2* bit positions. If *arg2* is greater than zero, the shift is to the left; if *arg2* is less than zero, the shift is to the right; if *arg2* equal zero, no shift occurs.

If *arg* is an INTEGER*2 argument and *arg2* is greater than 15 or *arg2* is less than -15, the result is zero. If *arg2* is an INTEGER*4 and *arg2* is greater than 31 or *arg2* is less than −31, the result is zero.

Bits shifted out from the left or right end are lost. Zeros are shifted in from the opposite end. The result is the same type as the arguments.

Examples

Function Call	Value Returned to `a`
`a = ISHFT(3, 4)`	`48`
`a = ISHFT(1, 4)`	`16`
`a = ISHFT(1, -4)`	`0`

The specific function names are ISHFT and JISHFT for INTEGER*4 arguments and HSHFT and IISHFT for INTEGER*2 arguments.

ISHFTC Function

`ISHFTC(`*arg1, arg2, arg3*`)` is a generic function that returns the circular shift of an INTEGER*4 argument. The result is the rightmost a *arg3* bits of *arg1* shifted circularly *arg2* places. That is, the bits shifted out of one end are shifted into the opposite end. No bits are lost.

The unshifted bits of the result are the same as the unshifted bits of the argument *arg1*. The absolute value of the argument *arg2* must be less than or equal to arg3. The argument *arg3* must be greater than or equal to one and less than or equal to 16 if *arg1* is INTEGER*2, or must be less than or equal to 32 if *arg1* is INTEGER*4. If *arg3* does not fall within this range, the results can be undefined.

Examples

Function Call	Value Returned to `i`
`i = ISHFTC(3, 4, 8)`	`48`
`i = ISHFTC(1, 4, 8)`	`16`

The specific function names are ISHFTC and JISHFTC for INTEGER*4 arguments and HSHFTC and IISHFTC for INTEGER*2 arguments.

IXOR Function

`IXOR(`*arg1*`,`*arg2*`)` is a generic function that returns the bitwise exclusive OR of two INTEGER*4 arguments. The result is INTEGER*4.

Function Call	**Value Returned to** `i`
`i = IXOR(1, 0)`	`1`
`i = IXOR(1, 1)`	`0`
`i = IXOR(0, 0)`	`0`

The alternate generic function name is IEOR. The specific function names are IEOR and JIXOR for INTEGER*4 arguments and HIEOR and IIXOR for INTEGER*2 arguments.

IZEXT

`IZEXT(`*arg*`)` is a generic function that returns a fixed-point argument of type INTEGER*2 without extending the sign bit of the argument.

In the following example, `i2` is an INTEGER*2 variable and `L1` is a LOGICAL*1 variable.

Examples

Function Call	**Value Returned to** `i2`
`i2 = IZEXT(-L1)`	`-1`
`i2 = IZEXT(L1)`	`255`, where `L1 = -1`

JNUM Function

`JNUM(`*arg*`)` is a specific function that returns the INTEGER*4 value represented in the character string *arg*.

Blanks are not significant in the input string.

Examples

Function Call	**Value Returned to** `i`
`i = JNUM('123')`	`123`
`i = JNUM('-99')`	`-99`
`i = JNUM('2000000000')`	`2000000000`
`i = JNUM('    24 ')`	`24` (blanks are ignored)

There is no generic name for this function.

LEN Function

`LEN(`*arg*`)` is a specific function that returns the length of a character string. The argument is type character and the result is an INTEGER*4 indicating the length of the argument.

Examples

Function Call	Value Returned to `i`
`i = LEN('string')`	`6`
`i = LEN('howlongami')`	`10`

There is no generic name for this function.

LGE Function

`LGE(`*arg1* , *arg2*`)` is a specific function that returns a logical result indicating whether *arg1* is lexically greater than or equal to *arg2*. The arguments are character strings. The result is true if *arg1* is equal to *arg2* or if *arg1* follows *arg2* in the ASCII collating sequence. In all other cases, the result is false. If *arg1* and *arg2* have unequal lengths the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples

Function Call	Value Returned to `i`
`i = LGE('ABC', 'BC')`	`F`
`i = LGE('BC', 'ABC')`	`T`
`i = LGE('ABC', 'ABC')`	`T`

There is no generic name for this function.

LGT Function

`LGT(`*arg1*`,`*arg2*`)` is a specific function that returns a logical result indicating whether *arg1* is lexically greater than *arg2*. The arguments are character strings. The result is true if *arg1* follows *arg2* in the ASCII collating sequence. In all other cases, the result is false. If *arg1* and *arg2* have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples

Function Call	**Value Returned to** `i`
`i = LGT('ABC', 'BC')`	`F`
`i = LGT('BC', 'ABC')`	`T`
`i = LGT('ABC', 'ABC')`	`F`

There is no generic name for this function.

LLE Function

`LLE(`*arg1*`, `*arg2*`)` is a specific function that returns a logical result indicating whether *arg1* is lexically less than or equal to *arg2*. The arguments are character strings. The result is true if *arg1* is equal to *arg2* or if *arg1* precedes *arg2* in the ASCII collating sequence. In all other cases, the result is false. If *arg1* and *arg2* have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples

Function Call	**Value Returned to** `i`
`i = LLE('ABC', 'BC')`	`T`
`i = LLE('BC', 'ABC')`	`F`
`i = LLE('ABC', 'ABC')`	`T`

There is no generic name for this function.

LLT Function

`LLT(`*arg1* , *arg2*`)` is a specific function that returns a logical result indicating whether *arg1* is lexically less than *arg2*. The arguments are character strings. The result is true if *arg1* precedes *arg2* in the ASCII collating sequence. In all other cases, the result is false. If *arg1* and *arg2* have unequal lengths, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

Examples

Function Call	**Value Returned to** `i`
`i = LLT('ABC', 'BC')`	`T`
`i = LLT('BC', 'ABC')`	`F`
`i = LLT('ABC', 'ABC')`	`F`

There is no generic name for this function.

LOG Function

`LOG(`*arg*`)` is a generic function that returns the natural logarithm (logarithm base e) of a REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16 argument; the argument must be greater than zero for REAL*4, REAL*8, and REAL*16 arguments. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = LOG(6.0)`	`1.791795`
`a = LOG(6.0D0)`	`1.791759469228055`
`a = LOG(var1)`	`(1.7917595, 0.00)`, where `var1 = (6.0D0, 0D0)`

The specific function names are ALOG for REAL*4 arguments, CLOG for COMPLEX*8 arguments, DLOG for REAL*8 arguments, QLOG for REAL*16, and ZLOG and CDLOG for COMPLEX*16 arguments.

LOG10 Function

`LOG10(`*arg*`)` is a generic function that returns the common logarithm (logarithm base 10) of a REAL*4, REAL*8, or REAL*16 argument; the argument must be greater than zero. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = LOG10(6.0)`	`0.7781513`
`a = LOG10(6.0D0)`	`0.778151250383644`

The specific function names are ALOG10 for REAL*4 arguments, DLOG10 for REAL*8 arguments, and QLOG10 for REAL*16 arguments.

MAX Function

`MAX(`*arg1,arg2, . . .* `)` is a generic function that returns the largest value from the list of arguments. The arguments can be INTEGER*4, REAL*4, REAL*8, or REAL*16. The number of arguments can vary, but there must be at least two. The result is the same data type as the arguments.

Examples

Function Call	**Value Returned to** `a`
`a = MAX(5, -2, 54, 11, 52)`	`54`
`a = MAX(5.0, 43.24, 44.1, 78.2)`	`78.2`

The function names are AMAX0, AIMAX0, and AJMAX0 for INTEGER*4 arguments with a REAL*4 result; AMAX1 for REAL*4 arguments; DMAX1 for REAL*8 arguments; QMAX1 for REAL*16 arguments; MAX0, IMAX0, and JMAX0 for INTEGER*4 arguments; and MAX1, IMAX1, and JMAX1 for REAL*4 arguments with an INTEGER*4 result.

`MAX1` behaves differently when either NOSTANDARD INTRINSICS or HP9000_300 is on.

MIN Function

`MIN(`*arg1*`,`*arg2*`[, ... ])` is a generic function that returns the smallest value from the list of arguments. The arguments can be INTEGER*4, REAL*4, REAL*8, or REAL*16; the number of arguments can vary, but there must be at least two. The result is the same data type as the arguments.

Examples

Function Call	**Value Returned to** `a`
`a = MIN(5, -2, 54, 11, 52)`	`-2`
`a = MIN(5.0, 43.24, 44.1, 78.2)`	`5.0`

The function names are AMIN0, AIMIN0, and AJMIN0 for INTEGER*4 arguments with a REAL*4 result; AMIN1 for REAL*4 arguments; DMIN1 for REAL*8 arguments; QMIN1 for REAL*16 arguments; MIN0, JMIN0, and IMIN0 for INTEGER*4 arguments; and MIN1, IMIN1, and JMIN1 for REAL*4 arguments with an INTEGER*4 result.

`MIN1` behaves differently when either NOSTANDARD INTRINSICS or HP9000_300 is on.

MOD Function

`MOD(`*arg1,arg2*`))` is a generic function that divides *arg1* by *arg2* and returns the remainder. The argument types can be INTEGER*4, REAL*4, REAL*8, or REAL*16. The result is the same data type as the arguments. If *arg2* is zero, the result is undefined.

Examples

Function Call	**Value Returned to** `a`
`a = MOD(30,13)`	`4`
`a = MOD(30.0, 13.0)`	`4.0`
`a = MOD(5, -3)`	`2`
`a = MOD(-5, 3)`	`-2`
`a = MOD(-5, -3)`	`-2`

The specific function names are MOD and JMOD for INTEGER*4 arguments, AMOD for REAL*4 arguments, DMOD for REAL*8 arguments, QMOD for REAL*16 arguments, and HMOD and IMOD for INTEGER*2 arguments.

MVBITS Subroutine

`MVBITS(`*arg1, arg2, arg3, arg4, arg5*`)` is a subroutine that moves *arg3* bits starting from position *arg2* of *arg1* to position *arg5* of *arg4*. The portion of *arg4* not affected by the movement of bits remains unchanged. All arguments are INTEGER*4 expressions, except *arg4*, which must be an INTEGER*4 variable or array element. Arguments *arg1* and *arg4* can be the same numeric storage unit. The value of *arg2* + *arg3* cannot exceed the bit length of *arg1* and the value of `(`*arg5* `+` *arg3*`)` cannot exceed the bit length of *arg4*.

Bit positions are numbered from right to left, with the rightmost (least significant) bit numbered zero. Figure B-1 shows how the MVBITS subroutine works.

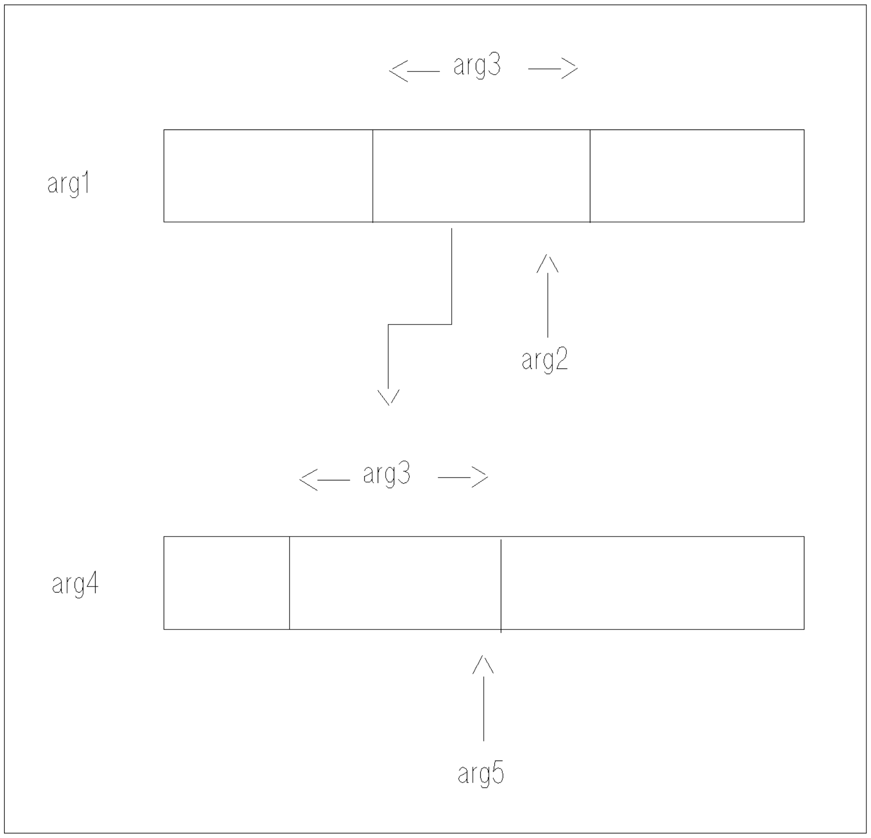


Figure B-1. MVBITS Subroutine

NINT Function

`NINT(`*arg*`)` is a generic function that returns the nearest integer. The argument can be REAL*4, REAL*8, or REAL*16; the result is INTEGER*4. If *arg* exceeds the largest integer allowed, the result is undefined. If the argument is positive or zero, the result is equal to `INT(`*arg* `+ 0.5)`. If the argument is negative, the result is equal to `INT(`*arg* `- 0.5)`.

Examples

Function Call	**Value Returned to** `i`
`i = NINT(123.456)`	`123`
`i = NINT(123.987)`	`124`
`i = NINT(123.5)`	`124`
`i = NINT(-123.456)`	`-123`
`i = NINT(-123.987)`	`-124`

The function names are NINT, ININT, and JNINT for REAL*4 arguments; IIDNNT, JIDNNT, and IDNINT for REAL*8 arguments; and IQNINT, IIQNNT, and JIQNNT for REAL*16 arguments. `NINT`, `IDNINT`, and `IQNINT` behave differently when either NOSTANDARD INTRINSICS or HP9000_300 is on. See chapter 7 for more information.

NOT Function

`NOT(`*arg*`)` is a generic function that returns the bitwise complement of an INTEGER*4 argument. The result is INTEGER*4.

Examples

Function Call	**Value Returned to** `i`
`i = NOT(1)`	`-2`
`i = NOT(0)`	`-1`
`i = NOT(5)`	`-6`
`i = NOT(-1)`	`0`

The preceding examples show the use of twos complement arithmetic.

The specific function names are NOT and JNOT for INTEGER*4 arguments and HNOT and INOT for INTEGER*2 arguments.

QEXT Function

`QEXT(`*arg*`)` is a generic function that converts the argument to REAL*16. The argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16. For an INTEGER*4, REAL*4, or REAL*8 argument, the result is as much precision of the significant part of the argument as the argument can provide. For a REAL*16 argument, the result is the argument. For a COMPLEX*8 or COMPLEX*16 argument, the result is as much precision of the significant real part of the argument as the argument can provide.

Examples

Function Call	Value Returned to `a`
`a = QEXT(4)`	`4.0`
`a = QEXT(4.0)`	`4.0`
`a = QEXT(4.0D2)`	`400.0`
`a = QEXT(var1)`	`4.0, where var1=(4.00, 2)`

The specific function name is QEXTD for REAL*8 arguments.

QNUM Function

`QNUM(`*arg*`)` is a specific function that returns the REAL*16 value represented in the character string *arg*.

Blanks are not significant in the input string.

Examples

Function Call	Value Returned to `d8`
`q16 = QNUM('123.5')`	`123.5`
`q16 = QNUM('-99.25')`	`-99.25`
`q16 = QNUM('327.125E75')`	`3.27125e+77`
`q16 = QNUM('   24  5 ')`	`245.0`

There is no generic name for this function.

QPROD Function

`QPROD(`*arg1*`,`*arg2*`)` is a specific function that returns the REAL*16 product of two REAL*8 arguments (*arg1* `*` *arg2*). The result is a REAL*16 number with none of the fractional portion lost and is equal to `QEXT(`*arg1*`) * QEXT(`*arg2*`)`.

Examples

Function Call	**Value Returned to** `q`
`q = QPROD(2.2d0, 2.2d0)`	`4.84000000000000078159700933611024`
`q = QPROD(1.0d0, 2.0d0)`	`2.0`

There is no generic name for this function.

REAL Function

`REAL(`*arg*`)` is a generic function that converts an argument to a REAL*4 number. The argument can be INTEGER*4, REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16. If *arg* is REAL*4, the result is equal to *arg*. If *arg* is INTEGER*4, REAL*8, or REAL*16 the result is as much precision of the significant part of *arg* as a REAL*4 item can contain. For a COMPLEX*8 argument (*ar,ai*), the result is *ar*. For a COMPLEX*16 argument (*ar,ai*), the result is as much significance of *ar* as a REAL*4 data item can contain.

Examples

Function Call	**Value Returned to** `r`
`r = REAL(5)`	`5.0`
`r = REAL(5.5)`	`5.5`
`r = REAL(5.55555D2)`	`555.555`
`r = REAL(var)`	`5.5`, where `var = (5.5, 5)`

The specific function names are FLOATI and FLOATJ for INTEGER*4 arguments, SNGL for REAL*8 arguments, and SNGLQ for REAL*16 arguments.

RNUM Function

`RNUM(`*arg*`)` is a specific function that returns the REAL*4 value represented in the character string *arg*.

Blanks are not significant in the input string.

Examples

Function Call	**Value Returned to** `r4`
`r4 = RNUM('123.5')`	`123.5E0`
`r4 = RNUM('-99.25')`	`-99.25E0`
`r4 = RNUM('327.125E15')`	`327.125E15`
`r4 = RNUM('   24. 5 ')`	`24.5` (blanks are ignored)

There is no generic name for this function.

SIGN Function

`SIGN(`*arg1*`,`*arg2*`)` is a generic function that transfers the sign from one numeric value to another. `SIGN(`*arg1*`,`*arg2*`)` returns the magnitude of *arg1* with the sign of *arg2*. The arguments can be INTEGER*4, REAL*4, REAL*8, or REAL*16. The result is the same data type as the arguments. The result is *arg1* if *arg2* is positive or zero, and *arg1* if *arg2* is negative.

Examples

Function Call	Value Returned to `a`
`a = SIGN(45.84, -133.0)`	`-45.84`
`a = SIGN(45.84, 133.0)`	`45.84`
`a = SIGN(-45.84, -133.0)`	`-45.84`

The specific function names are SIGN for REAL*4 arguments, DSIGN for REAL*8 arguments, QSIGN for REAL*16, JISIGN for INTEGER*4 arguments, and HSIGN and IISIGN for INTEGER*2 arguments. ISIGN can also be used as a generic function name for integer arguments, and accepts INTEGER*2 and INTEGER*4 arguments.

SIN Function

`SIN(`*arg*`)` is a generic function that returns the sine of the argument. The argument is expressed in radians and is REAL*4, REAL*8, REAL*16, or COMPLEX*8. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = SIN(0.0)`	`0.0`
`a = SIN(1.5708)`	`1.0`
`a = SIN(0.0628)`	`0.06275873`

The specific function names are SIN for REAL*4 arguments, DSIN for REAL*8 arguments, QSIN for REAL*16 arguments, CSIN for COMPLEX*8 arguments, and ZSIN and CDSIN for COMPLEX*16 arguments.

SIND Function

`SIND(`*arg*`)` is a generic function that returns the sine of the argument. The argument is expressed in degrees and is REAL*4, REAL*8, or REAL*16. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = SIND(30.0)`	`.500000`
`a = SIND(20.0)`	`.342020`
`a = SIND(45.0D0)`	`.7071067811865476`

The specific function names are SIND for REAL*4 arguments, DSIND for REAL*8 arguments, and QSIND for REAL*16 arguments.

SINH Function

`SINH(`*arg*`)` is a generic function that returns the hyperbolic sine of a REAL*4, REAL*8, or REAL*16 argument. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = SINH(0.0)`	`0.0`
`a = SINH(1.5708)`	`2.3013079`

The specific function names are SINH for REAL*4 arguments, DSINH for REAL*8 arguments, and QSINH for REAL*16 arguements.

SIZEOF Function

`SIZEOF(`*arg*`)` is a generic function that returns the number of bytes of storage used by the argument. The argument cannot be a dynamic or assumed-size array; any other argument with a valid data type is allowed, including constants and arbitrary expressions.

SQRT Function

`SQRT(`*arg*`)` is a generic function that returns the square root of a REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16 argument. The result is the same data type as the argument. The argument cannot be negative for REAL*4 and REAL*8 values.

Examples

Function Call	Value Returned to `a`
`a = SQRT(9.0)`	`3.0`
`a = SQRT(49.0D0)`	`7.0`
`a = SQRT(var)`	`(5.0, 0.0), where var = (25, 0.0)`

The specific function names are SQRT for REAL*4 arguments, CSQRT for COMPLEX*8 arguments, DSQRT for REAL*8 arguments, QSQRT for REAL*16 arguments, and ZSQRT and CDSQRT for COMPLEX*16 arguments.

TAN Function

`TAN(`*arg*`)` is a generic function that returns the tangent of the argument. The argument is expressed in radians, and its type is REAL*4, REAL*8, REAL*16, COMPLEX*8, or COMPLEX*16.

The value of *arg* must be less than or equal to the maximum number allowed on your system and not close to $(\pm(2n + 1) * \pi/2)$, where n is an INTEGER*4. The result is the same data type as the argument.

Examples

Function Call	Value Returned to `a`
`a = TAN(3.0)`	`-0.1425465`
`a = TAN(1.0)`	`1.5574077`

The specific function names are TAN for REAL*4 arguments, DTAN for REAL*8 arguments, QTAN for REAL*16, CTAN for COMPLEX*8 arguments, and ZTAN for COMPLEX*16 arguments.

TAND Function

`TAND(`*arg*`)` is a generic function that returns the tangent of the argument. The argument is expressed in degrees, and its type is REAL*4, REAL*8, or REAL*16. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = TAND(3.0)`	`0.0524078`
`a = TAND(3.0D1)`	`.5773502691896257`

The specific function names are TAND for REAL*4 arguments, DTAND for REAL*8 arguments, and QTAND for REAL*16.

TANH Function

`TANH(`*arg*`)` is a generic function that returns the hyperbolic tangent of a REAL*4, REAL*8, or REAL*16 argument. The result is the same data type as the argument.

Examples

Function Call	**Value Returned to** `a`
`a = TANH(3.0)`	`0.9950548`
`a = TANH(1.0)`	`.7615942`

The specific function names are TANH for REAL*4 arguments, DTANH for REAL*8 arguments, and QTANH for REAL*16 arguments.

ZEXT Function

`ZEXT(`*arg*`)` is a generic function that returns a fixed-point argument of the same size or larger without extending the sign bit of the argument. The size of the result depends on whether the SHORT directive is enabled. If SHORT is enabled, ZEXT behaves like IZEXT and returns an INTEGER*2 result. If SHORT is not enabled, ZEXT behaves like JZEXT and returns an INTEGER*4 result. (IZEXT and JZEXT are explained below.) `ZEXT` behaves differently when either NOSTANDARD INTRINSICS or HP9000_300 is on. See chapter 7 for more information.

If SHORT is on, `ZEXT` returns an INTEGER*2; if LONG is on, it returns an INTEGER*4.

In the following examples, `i4` is an INTEGER*4 variable and SHORT is not enabled.

Examples

Function Call	**Value Returned to** `i4`
`i4 = ZEXT(-32768i)`	`32768j*`
`i4 = ZEXT(-1)`	`65535j`

* Note that `i4 = -32768i` would assign `-32768j` to `i4.`

IZEXT is a generic function that accepts LOGICAL*1, LOGICAL*2, or INTEGER*2 arguments, and returns an INTEGER*4 result. JZEXT is a generic function that accepts LOGICAL*1, LOGICAL*2, LOGICAL*4, INTEGER*2 or INTEGER*4 arguments, and returns an INTEGER*4 result.

The specific function names are IZEXT for INTEGER*2 arguments and JZEXT for INTEGER*4 arguments. IZEXT accepts an INTEGER*2 argument and returns an INTEGER*2 result. JZEXT accepts either an INTEGER*2 or INTEGER*4 argument and returns an INTEGER*4 result.

C

FORTRAN Comparisons

This appendix makes the following comparisons:

- The HP FORTRAN 77 compiler is compared with the ANSI 77 standard by listing HP FORTRAN 77 extensions to the ANSI 77 standard.
- The HP FORTRAN 77 compiler is compared with FORTRAN 66/V.
- The HP FORTRAN 77 compiler is compared with FORTRAN 7X.

Note FORTRAN 66/V has previously been known as FORTRAN/3000.

Extensions to the Standard

HP FORTRAN 77 fully implements the ANSI 77 standard for FORTRAN. HP FORTRAN 77 also contains many extensions to this standard. This appendix categorizes and lists these extensions. Complete descriptions are given at the point in the manual where each topic is found.

MIL-STD-1753 Extensions

The HP FORTRAN 77 compiler fully implements the Military Standard Definition (MIL-STD-1753) of extensions to the ANSI 77 standard. These extensions are as follows:

- DO WHILE loops.
- INCLUDE statement (also INCLUDE directive).
- IMPLICIT NONE statement.
- The following bit manipulation intrinsic functions:

```
BTEST   IBCLR   IBSET   IOR     ISHFTC  NOT
IAND    IBITS   IEOR    ISHFT   MVBITS
```

- Octal and hexadecimal constants in DATA and PARAMETER statements. A further extension of MIL-STD-1753 is the ability to include octal and hexadecimal constants in expressions within assignments and to use them as actual parameters.
- READ and WRITE past end-of-file.

Other Extensions

These are the system dependent and all other extensions to the ANSI 77 standard.

- Block DO loops.
- Extended range DO loops.
- Label omitted in block Do loop.
- 128-bit complex data type (COMPLEX*16), as approved by the IFIP WG 2.5 Numerical Software Group.
- 8-bit integer data type (BYTE or LOGICAL*1).
- 16-bit integer data type (INTEGER*2).
- 16-bit logical data type (LOGICAL*2).
- Underscores and dollar signs in symbolic names.
- Lower case letters as part of FORTRAN character set.
- Symbolic names greater than six characters.
- Equivalence of character and noncharacter items.
- Character and noncharacter items can be mixed in same common block.
- Exclamation point (`!`) at the beginning of an embedded comment.

- Byte length specified in numeric type statements, for example, INTEGER*4. (Including the byte length in CHARACTER type statements is part of the ANSI 77 standard).
- Compiler directives.
- Integer intrinsic functions cover both two-byte (INTEGER*2) and four-byte (INTEGER*4) integers.
- Concatenation of an item of type CHARACTER*(*).
- Mixed lengths among character-typed entries.
- Unlimited number of array dimensions (the ANSI 77 standard specifies only seven).
- The logical operators—.AND., .EQV., .NEQV., .NOT., .OR., and .XOR.—can be applied to integer data to perform bit masking and bit manipulation.
- A numeric array can be used as a format specifier in an input/output statement.
- Formal parameters can be specified for a program and can be passed as values from the run string.
- Recursion is permitted.
- Hollerith, octal, and hexadecimal typeless constants.
- The letter J appended to an integer constant to explicitly specify type INTEGER*4.
- The letter I appended to an integer constant to explicitly specify type INTEGER*2.
- Logical operands can be intermixed with numeric operands.
- Length specification can be a variable enclosed in parentheses.
- A length specifier can follow the item being declared.
- Quotation marks used as string delimiters.
- Integer values can be input or output in octal or hexadecimal format.
- The SYSTEM INTRINSIC statement (from FORTRAN 66/V).
- The ON statement (from FORTRAN 66/V).
- Additional format specifications: @, K, O, Q, R, Z.
- Variable format descriptors.
- VOLATILE statement.
- List-directed I/O transfers can be made on internal files.
- Data initialization can be performed in type declaration statements by enclosing the initialization value in slashes (/ /).
- A COMMON statement can contain a name that has been initialized in a DATA statement or type declaration statement.

- A variable of type integer can be used as a character length specifier.
- Dynamic arrays.
- Optional label in an Arithmetic IF.
- A tab in column 1-6 immediately followed by a digit from 1-9, and blanks or nothing before the tab character, is a line continuation.
- Consecutive operators are allowed if the second operator is either a unary plus (`+`) or minus (`-`).
- Multi-dimensioned EQUIVALENCE.
- A CALL can have missing arguments, which are replaced by a zero passed by value.
- An optional comma (`,`) is allowed to precede the I/O list within a WRITE statement.
- Null strings are allowed in the same context where other strings are allowed.
- The use of `&` instead of `*` for alternate return arguments is allowed.
- Keyword statements: ACCEPT, DECODE, DOUBLE COMPLEX, ENCODE, NAMELIST, TYPE, VIRTUAL.
- PROGRAM statement allows the declaration of parameters for the main program unit.
- Use of noninteger expressions in computed GOTO statements.
- Allows blank commons to be initialized by block data subprograms.
- REAL*16.
- Support of user defined structure types (records).

- The following intrinsic functions are included:

%LOC	BTEST	HBITS	IISIGN	JISHFTC	QEXTD
%REF	CDABS	HBSET	IIXOR	JISIGN	QINT
%VAL	CDCOS	HDIM	IMAG	JIXOR	QLOG
ACOSD	CDEXP	HIAND	IMAX0	JMAX0	QLOG10
ACOSH	CDLOG	HIEOR	IMAX1	JMAX1	QMAX1
AIMAX0	CDSIN	HIOR	IMIN0	JMIN0	QMIN1
AIMIN0	CDSQRT	HMOD	IMIN1	JMIN1	QMOD
AJMAX0	COSD	HMVBITS	IMOD	JMOD	QNINT
AJMIN0	DACOSD	HNOT	ININT	JNINT	QNUM
ASIND	DACOSH	HSHFT	INOT	JNOT	QPROD
ASINH	DASIND	HSHFTC	INUM	JNUM	QSIGN
ATAN2D	DASINH	HSIGN	IOR	JZEXT	QSIN
ATAND	DATAN2D	HTEST	IQINT	MVBITS	QSIND
ATANH	DATAND	IAND	IQNINT	NOT	QSINH
BABS	DATANH	IBCLR	IRAND	QABS	QSQRT
BADDRESS	DBLEQ	IBITS	ISHFT	QACOS	QTAN
BBCLR	DCMPLX	IBSET	ISHFTC	QACOSD	QTAND
BBITS	DCONJG	IEOR	IXOR	QACOSH	QTANH
BBSET	DCOSD	IIABS	IZEXT	QASIN	RAND
BBTEST	DDINT	IIAND	JIABS	QASIND	RNUM
BDIM	DFLOAT	IIBCLR	JIAND	QASINH	SIND
BIAND	DFLOTI	IIBITS	JIBCLR	QATAN	SNGLQ
MIEOR	DFLOTJ	IIBSET	JIBITS	QATAN2	TAND
BIOR	DIMAG	IIDIM	JIBSET	QATAN2D	ZABS
BITEST	DNUM	IIDINT	JIDIM	QATAND	ZCOS
BIXOR	DREAL	IIDNNT	JIDINT	QATANH	ZEXP
BJTEST	DSIND	IIEOR	JIDNNT	QCOS	ZEXT
BMOD	DTAND	IIFIX	JIEOR	QCOSD	ZLOG
BMVBITS	FLOATI	IINT	JIFIX	QCOSH	ZSIN
BNOT	FLOATJ	IIOR	JINT	QDIM	ZSQRT
BSHFT	HABS	IISHFT	JIOR	QEXP	ZTAN
BSHFTC	HBCLR	IISHFTC	JISHFT	QEXT	
BSIGN					

Comparison of HP FORTRAN 77 and FORTRAN 66/V

FORTRAN 66/V is an implementation of ANSI FORTRAN (X3.9-1966) with several extensions to the standard. Listed below are some of the differences between FORTRAN 66/V and HP FORTRAN 77. This is not a complete list of differences, but most of the significant features are compared. For more information about differences between FORTRAN 66/V and HP FORTRAN 77, see the *FORTRAN 66/V to HP FORTRAN 77/V Migration Guide.*

- Free-format source code is allowed in FORTRAN 66/V, but not in HP FORTRAN 77.
- Identifiers in HP FORTRAN 77 can contain an underscore (_) or a dollar sign ($). Neither of these is allowed in FORTRAN 66/V.
- The default size of integers in FORTRAN 66/V is two bytes (INTEGER*2), while in HP FORTRAN 77 it is four bytes (INTEGER*4). INTEGER*4 is termed DOUBLE INTEGER in FORTRAN 66/V.
- The default size of logicals in FORTRAN 66/V is two bytes (LOGICAL*2). In HP FORTRAN 77 it is four bytes (LOGICAL*4). FORTRAN 66/V does not support LOGICAL*4.
- HP FORTRAN 77 has DOUBLE COMPLEX and BYTE data types.
- Both HP FORTRAN 77 and FORTRAN 66/V support character substrings; however, the notation is different in the two compilers. In HP FORTRAN 77, the size of the array and length of the string for character variables are in the opposite order from FORTRAN 66/V.
- The syntax and semantics of substring designators are different in FORTRAN 66/V and FORTRAN 77.
- Partial word designators exist in FORTRAN 66/V but not FORTRAN 77. Bit extraction is accomplished in FORTRAN 77 by the IBITS and MVBITS functions.
- The notation for octal, character, and hexadecimal constants is different in the two compilers.
- In FORTRAN 66/V, only the upper bound of an array is specified. The lower bound is always assumed to be 1. In HP FORTRAN 77, both upper and lower bounds can be given. If the lower bound is omitted, it is assumed to be 1.
- FORTRAN 66/V has a symbol table size limit of 8191 words. This restriction does not exist in HP FORTRAN 77.
- HP FORTRAN 77 provides a concatenation operator for character variables. This feature is not available in FORTRAN 66/V.
- HP FORTRAN 77 provides the additional logical operators .EQV. and .NEQV..

- The order of evaluating comparisons using the logical IF statement is different in the two compilers.
- The order for evaluating arithmetic expressions with two or more operators of the same precedence is different in the two compilers. In HP FORTRAN 77, the evaluation is right to left for exponential and left to right for all other operations.
- Mixed mode expressions are evaluated differently in the two compilers.
- The PROGRAM statement in HP FORTRAN 77 can include parameters that enable the program to access the PARM value and the INFO string of the run string. The PROGRAM statement in FORTRAN 66/V has no parameters.
- The PARAMETER statement in HP FORTRAN 77 allows the use of constant expressions. In FORTRAN 66/V, only a simple constant can be assigned.
- HP FORTRAN 77 has a SAVE statement that is not available in FORTRAN 66/V. It allows variables in a subroutine to be saved from one call of the subroutine to the next.
- In HP FORTRAN 77, the IF and DO statements are greatly expanded to provide more structured programming constructs. These statements are a superset of those provided in FORTRAN 66/V.
- HP FORTRAN 77 provides an INCLUDE statement and an $INCLUDE compiler directive that allow source from another file to be included. This feature is not available in FORTRAN 66/V, which uses MASTERFILES and NEWFILES for source control.
- HP FORTRAN 77 has a number of I/O statements not available in FORTRAN 66/V. In HP FORTRAN 77, OPEN, CLOSE, and INQUIRE statements are provided, so there is no need to call system intrinsics to do routine I/O operations.
- In HP FORTRAN 77, all I/O statements allow a status word and error label as optional parameters. In FORTRAN 66/V, the error label is allowed in some (but not all) of the I/O statements, and status words are not allowed.
- HP FORTRAN 77 does not contain the ACCEPT and DISPLAY statements, as FORTRAN 66/V does.
- There are significant differences in the format and edit descriptors of HP FORTRAN 77 and FORTRAN 66/V.
- The S edit descriptor has different meanings in the two compilers.
- In HP FORTRAN 77, functions can have empty parameter lists. This is denoted by `()`. In FORTRAN 66/V, a formal parameter has to be passed to a function for which there would otherwise be no parameter.

- These compiler directives and options are part of FORTRAN 66/V but not HP FORTRAN 77:

`CONTROL CROSSREF ALL`	`CONTROL FIXED`	`CONTROL NOSTAT`	`ERRORS`
`CONTROL CROSSREF`	`CONTROL FREE`	`CONTROL SOURCE`	`TRACE`
`CONTROL ERRORS`	`CONTROL LABEL`	`CONTROL STAT`	
`CONTROL FILE`	`CONTROL NOLABEL`	`EDIT`	

- The syntax of many compiler directives differs between FORTRAN 66/V and HP FORTRAN 77. Also, some FORTRAN 66/V compiler directives are system dependent in FORTRAN 77 and may not be available on all operating systems. See Chapter 7 for details.
- In FORTRAN 66/V, the condition code is accessed by using `.CC.` in an arithmetic IF statement. HP FORTRAN 77 provides an INTEGER*2 function, CCODE, that can be used wherever an integer expression is allowed.
- In FORTRAN 77 parentheses are required around parameters in PARAMETER statements.
- Some functions have different names in FORTRAN 66/V and FORTRAN 77 because of different default parameter types.
- While FORTRAN 66/V reserves one word for constants passed as parameters, FORTRAN 77 reserves two words. Therefore you must use caution in FORTRAN 77 when passing parameter constants from procedures in other languages.
- Logical items are implemented very differently in FORTRAN 66/V and FORTRAN 77. (However, compiler directives that eliminate this incompatibility are available. See Chapter 7 for further information.)
- Some FORTRAN 66/V functions don't exist in FORTRAN 77. See the *FORTRAN 66/V to HP FORTRAN 77/V Migration Guide* for a list of these functions.
- When FORTRAN 66/V passes a character variable to a subprogram, it passes a character pointer alone. When FORTRAN 77 passes a character variable, it passes a length parameter in addition to the character pointer. (The extra length parameter should be taken into account to avoid exceeding parameter limits in FORTRAN 77.)
- FORTRAN 66/V allows composite numbers, whereas FORTRAN 77 does not allow them.
- Debug line notation is different in FORTRAN 66/V and FORTRAN 77.

Comparison of HP FORTRAN 77 and FORTRAN 7X

Note

The *FORTRAN 77 Reference Manual* cited in this section is the reference manual for FORTRAN 7X (FORTRAN 77 for the HP 1000 computer system), not the manual you are reading now.

One of the major differences between FORTRAN 7X and HP FORTRAN 77 is that the former can be run in an ANSI 66 mode. This capability is not part of HP FORTRAN 77.

Conflicts are sometimes generated regarding program execution in the two modes of FORTRAN 7X, ANSI 66 and ANSI 77. HP FORTRAN 77 can resolve one of these ANSI 66-ANSI 77 conflicts. In FORTRAN 7X's ANSI 77 mode, DO loops can be skipped, depending on the values of the control variables; in ANSI 66 mode, all DO loops execute at least once. In HP FORTRAN 77, the ONETRIP directive can be used to specify whether to skip or execute.

Other conflicts that can be resolved in FORTRAN 7X (by specifying mode) but not in HP FORTRAN 77 are:

- Computed GOTO value out-of-bounds condition.
- Intrinsics declared in EXTERNAL and type statements (use the INTRINSIC statement instead in HP FORTRAN 77).

Refer to the *FORTRAN 77 Reference Manual* (for the HP 1000 computer system) for a description of each of the above conflicts.

The following are additional differences between FORTRAN 7X and HP FORTRAN 77:

- The EMA statement is not part of HP FORTRAN 77.
- In FORTRAN 7X a multidimensional array can be referenced by a single dimension in the EQUIVALENCE statement. (See "EQUIVALENCE Statement" in the *FORTRAN 77 Reference Manual*.) This extension is not part of HP FORTRAN 77.
- In FORTRAN 7X, the BLOCK DATA, FUNCTION, PROGRAM, and SUBROUTINE statements allow comments to be specified. This is not part of HP FORTRAN 77 (since it has no NAM record).
- In HP FORTRAN 77, instead of specifying the program type and priority in the PROGRAM statement, you can specify formal parameters and pass values from the run string.
- HP FORTRAN 77 has no LIST or NOLIST option for the INCLUDE statement or directive as in FORTRAN 7X. (Refer to the *FORTRAN 77 Reference Manual*.)

- The CALL EXIT extension is not part of HP FORTRAN 77.
- The FILES directive reserves room for the DCB in FORTRAN 7X. This is not necessary in HP FORTRAN 77. The directive is flagged with a warning message and ignored.
- The ALIAS directive in HP FORTRAN 77 does not have the DIRECT, NOABORT, and NOEMA options, but does have the parameter passing information option.
- In FORTRAN 7X, true is equal to any negative value and false to any nonnegative value. In HP FORTRAN 77/iX, the low-order bit of the high-order byte determines the logical value. See Chapter 10 for details.
- The FORTRAN intrinsics (see Appendix B) are the same in HP FORTRAN 77, except for the following:

```
PCOUNT  ISSW    EXEC    REIO
```

- The FTN control statement is not part of HP FORTRAN 77.
- Compiler invocations differ, as do compiler options and directives.
- The carriage control character for no advance in HP FORTRAN 77 is "+", as specified in the ANSI 77 standard. FORTRAN 7X uses the nonstandard "*" instead.
- In FORTRAN 7X, if the FORMAT statement specifies a record size greater than 67 words, LGBUF must be called. This restriction does not apply to HP FORTRAN 77, where nothing special is required.
- In FORTRAN 7X, the sizes of the `w`, `d`, and `n` fields in format specifications are checked at compile-time for a value greater than 2047. In HP FORTRAN 77 these fields are checked at compile time if they appear in FORMAT statements. Numeric format descriptors must not specify a field width greater than 140. Character field widths are not restricted.
- HP FORTRAN 77 allows an unlimited number of array dimensions, whereas FORTRAN 7X (and the ANSI 77 standard) allows only seven.
- The six-byte REAL data type is implemented in FORTRAN 7X, but not in HP FORTRAN 77.
- In FORTRAN 7X, as an extension to the ANSI 77 standard, unlimited continuation lines are permitted. In HP FORTRAN 77, only 99 are permitted.
- FORTRAN 7X include compatibility features that are not a part of the ANSI 77 standard. Compatibility features included in FORTRAN 7X but not HP FORTRAN 77 are as follows:
  - Extended precision type.
  - Improper array dimensioning in EQUIVALENCE statement.

  - Record number connected to unit number (earlier-style direct-access I/O).
  - Statement function in EXTERNAL statements as arguments.
  - Parentheses around simple I/O lists.
  - $ as statement separator.
  - Storage of Hollerith constants.
  - Unformatted I/O and paper tape length words.

  Refer to the *FORTRAN 77 Reference Manual* for a detailed description of each of the above features.

- The ASSIGN and the assigned GOTO statements require a 32-bit integer variable in HP FORTRAN 77. This is not a restriction in FORTRAN 7X.
- FORTRAN 7X allows repeat count for both format and edit descriptors in format statements. HP FORTRAN 77 allows only format descriptors to have repeat counts.
- In FORTRAN 7X, as an extension to the ANSI 77 standard, a DO variable can be modified.
- In FORTRAN 7X, array names may be used alone to specify the first element of the array. This is nonstandard and not allowed in HP FORTRAN 77.

D

HP Character Set

Effects of Control Key*

000-037B | 040-077B | 100-137B | 140-177B

				b_7 b_6 b_5	0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}
BITS				COLUMN								
b_4	b_3	b_2	b_1	ROW	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	'	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	;
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

32 Control Codes

Upshifted Lower case

64 Character Set

96 Character Set

128 Character Set

EXAMPLE:

The representation for the character "K" (column 4, row 11) is:

$$\begin{array}{lccccccc} & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 \\ \text{BINARY} & 1 & \underbrace{0 \quad 0 \quad 1} & & & \underbrace{0 \quad 1 \quad 1} & & \\ \text{OCTAL} & 1 & 1 & & & 3 & & \end{array}$$

LG200025_094a

*Depressing the Control key while typing an upper case letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

Hewlett-Packard Character Set for Computer Systems

This table shows HP's Implementation of ANS X3 4-1968 (USASCII) and ANS X3 32-1973. Some devices may substitute alternate characters from those shown in this chart (for example Line Drawing Set or Scandanavia font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16 bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word add the two values. For example, AB produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Mnemonic	Graphic[1]	Definition
	Left Byte	Right Byte			
0	000000	000000	NUL	$^{N}_{U}$	Null
1	000400	000001	SOH	$^{S}_{H}$	Start of Heading
2	001000	000002	STX	$^{S}_{X}$	Start of Text
3	001400	000003	ETX	$^{E}_{X}$	End of Text
4	002000	000004	EOT	$^{E}_{T}$	End of Transmission
5	002400	000005	ENQ	$^{E}_{Q}$	Enquity
6	003000	000006	ACK	$^{A}_{K}$	Acknowledge
7	003400	000007	BEL	🔔	Bell, Attention Signal
8	004000	000010	BS	$^{B}_{S}$	Backspace
9	004400	000011	HT	$^{H}_{T}$	Horizontal Tabulation
10	005000	000012	LF	$^{L}_{F}$	Line Feed
11	005400	000013	VT	$^{V}_{T}$	Vertical Tabulation
12	006000	000014	FF	$^{F}_{F}$	Form Feed
13	006400	000015	CR	$^{C}_{R}$	Carriage Return
14	007000	000016	SO	$^{S}_{O}$	Shift Out } Alternate Character Set
15	007400	000017	SI	$^{S}_{I}$	Shift In }
16	010000	000020	DLE	$^{D}_{L}$	Data Line Escape
17	010400	000021	DC1	$^{D}_{1}$	Device Control 1 (XON)
18	011000	000022	DC2	$^{D}_{2}$	Device Control 2 (Tape)
19	011400	000023	DC3	$^{D}_{3}$	Device Control 3 (XOFF)
20	012000	000024	DC4	$^{D}_{4}$	Device Control 4 ($\overline{\text{TAPE}}$)
21	012400	000025	NAK	$^{N}_{K}$	Negative Acknowledge
22	013000	000026	SYN	$^{S}_{Y}$	Synchronous Idle
23	013400	000027	ETB	$^{E}_{B}$	End of Transmission Block
24	014000	000030	CAN	$^{C}_{N}$	Cancel
25	014400	000031	EM	$^{E}_{M}$	End of Medium
26	015000	000032	SUB	$^{S}_{B}$	Substitute
27	015400	000033	ESC	$^{E}_{C}$	Escape[2]
28	016000	000034	FS	$^{F}_{S}$	File Separator
29	016400	000035	GS	$^{G}_{S}$	Group Separator
30	017000	000036	RS	$^{R}_{S}$	Record Separator
31	017400	000037	US	$^{U}_{S}$	Unit Separator
127	077400	000177	DEL	█	Delete, Rubout[3]

Decimal Value	Octal Values		Character	Definition
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent Sign
38	023000	000046	&	Ampersand, And Sign
39	023400	000047	'	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus Sign
44	026000	000054	,	Comma
45	026400	000055	-	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	} Digits, Numbers
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	350000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than Sign
61	036400	000075	=	Equals Sign
62	037000	000076	>	Greater Than Sign
63	037400	000077	?	Question Mark

LG200025_095a

Decimal Value	Octal Values		Character	Definition
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At Sign
65	040400	000101	A	
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	Uppercase Alphabet
78	047000	000116	N	Capital Letters
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	Left (opening) Bracket
92	056000	000134	\	Backslash, Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret, Circumflex, Up Arrow[4]
95	057400	000137	_ ←	Underline, Back Arrow[4]

Decimal Value	Octal Values		Character	Definition
	Left Byte	Right Byte		
96	060000	000140	‘	Grave Accent[5]
97	060400	000141	a	
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	Lowercase Letters[5]
110	067000	000156	n	
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	Left (opening) Brace[5]
124	076000	000174	|	Vertical Bar[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde [5]

Notes: [1]This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as @ or space.

[2]Escape is the first character of a special control sequence. For example, ESC followed by J clears the display on a 2640 terminal.

[3]Delete may be displayed as ___ @ or space.

[4]Normally, the caret and underline are displayed. Some devices substitute the up arrow and back arrow.

[5]Some devices upshift lowercase letters and symbols (‘ through ~) to the corresponding uppercase character (@ through ^). For example, the left brace would be converted to a left bracket.

E

Indexed Sequential Access Program

The following program uses indexed sequential access (ISAM) I/O. Its operation is described in comments within the program.

```
C***************************************************************************
C    This program shows different indexed sequential access operations.
C    This program is menu driven.  It creates an ISAM file with three
C    keys and writes some records into the file.  It then displays the
C    menu and prompts for these options:  add, read, delete, and modify
C    a record.  This program has an option to dump the entire ISAM file.
C***************************************************************************
      PROGRAM test1
      INTEGER key,phone
      CHARACTER buf*80,filename*8,keyed*5,unfo*4
      EQUIVALENCE (buf(1:4),phone)
      LOGICAL modified

       C********************************************************************
       C     field definition
       C     1:4:integer  ----------- primary key
       C     5:15: character -------- alternate key (last name)
       C     16:25: character ------- alternate key (first name)
       C     26:80: character ------- not a key (for general description)
       C********************************************************************
             filename = 'datafile'
             keyed    = 'keyed'
             unfo     = 'unfo'

         C**************************************************************
         C  Define primary key as 1:4:integer
         C  secondary key1 : 5:15:character
         C  secondary key2 : 16:25:character
         C**************************************************************
         C OPEN ISAM file as
                OPEN(10,file='DATAFILE',access='keyed',form='unfo',recl=80,
              1    key=(1:4:integer,5:15:character,16:25),err=1000)
```

```
C**********************************************************
C ADD some records
C**********************************************************
      phone =   1231111
      buf(5:15) = 'micky'
      buf(16:25)= 'mouse'
      buf(26:)  = ' DISNEYLAND'
      WRITE (10,err=1 ) buf    ! records might already be created

      phone =   1232222
      buf(5:15) = 'donald'
      buf(16:25)= 'duck'
      buf(26:)  = ' DISNEYLAND'
      WRITE (10) buf

      phone =   1233333
      buf(5:15) =  'big'
      buf(16:25)=  'bird'
      buf(26:)  = ' SESAME STREET'
      WRITE (10) buf

   C**********************************************************
   C  PRINT the menu
   C**********************************************************
   5     CONTINUE
   1     PRINT *,'                ***************'
         PRINT *,'                    M E N U'
         PRINT *,'                ***************'
         PRINT *,' '
         PRINT *,'                  1. ADD A RECORD'
         PRINT *,'                  2. READ A RECORD'
         PRINT *,'                  3. DELETE A RECORD'
         PRINT *,'                  4. MODIFY A RECORD'
         PRINT *,'                  5. MENU'
         PRINT *,'                  6. EXIT'
         PRINT *,'                  7. DUMP THE FILE'
         PRINT *,' '
         PRINT *,' '

   2     PRINT 11
   11    FORMAT('                enter your option :',$)
         READ *,i
         GOTO (100,200,300,400,500,600,700) I
         PRINT *,'invalid option, try again'
         GOTO 2

   12    FORMAT(1x,'enter phone number:',$)
   13    FORMAT(1x,'enter first name:',$)
   14    FORMAT(1x,'enter last name:',$)
   15    FORMAT(1x,'enter project name(optional):',$)
```

```
C**********************************************************
C  ADD a record
C**********************************************************
100   PRINT 12
      READ *,phone
      PRINT 13
      READ (*,'(A10)') buf(16:25)
      PRINT 14
      READ (*,'(A11)') buf(5:15)
      PRINT 15
      READ (*,'(A55)') buf(26:80)
      WRITE (10,err=101) buf
      GOTO 1
101   PRINT *,'error in reading, try again'
      GOTO 1
```

```
C***********************************************************
C  READ a record
C***********************************************************
200   PRINT *,'                        1. BY PHONE NUMBER'
      PRINT *,'                        2. BY FIRST NAME'
      PRINT *,'                        3. BY LAST NAME'
      PRINT *,' '
      PRINT *,' '
      PRINT 11
      READ *,i
      IF (I .EQ. 1) THEN
        PRINT 12
        READ *,phone
        READ (10,keyeq=phone,keyid=0,err=212,end=211,iostat=ii) buf
        PRINT *,phone,'  ',buf(5:)
      ELSEIF (I .EQ. 2) THEN
        PRINT 13
        READ (*,'(A10)') buf(16:25)
        READ (10,keyeq=buf(16:25),keyid=2,err=212,end=211,iostat=ii) buf
        PRINT *,phone,'  ',buf(5:)
      ELSEIF (I .EQ. 3) THEN
        PRINT 14
        READ (*,'(A11)') buf(5:15)
        READ (10,keyeq=buf(5:15),keyid=1,err=212,end=211,iostat=ii) buf
        PRINT *,phone,'  ',buf(5:)
      ELSE
        PRINT *,'invalid option, try again'
        GOTO 200
      ENDIF
      GOTO 1

211   PRINT *,'record does not exist, try again',ii
      GOTO 1
212   PRINT *,'error in reading :' ,ii
      GOTO 1
C***********************************************************
C  DELETE a record
C***********************************************************
300   PRINT 12
      READ *,phone
      READ (10,keyeq=phone,err=301) buf         !default primary key
      PRINT *,phone,'  ',buf(5:70)
      DELETE(10,err=302)
      GOTO 1
301   PRINT *,'error in reading the record for delete'
      GOTO 1
302   PRINT *,'error in deleting the record'
      GOTO 1
```

```
C*************************************************************
C  MODIFY a record
C*************************************************************
400   modified = .false.
      PRINT 12
      READ *,phone
      READ(10,keyeq=phone,err=402) buf   ! default is primary key
      PRINT *,phone,' ',buf(5:)

401   PRINT *,'           1. first name'
      PRINT *,'           2. last name',ii
      PRINT *,'           3. project'
      PRINT *,'           4. exit'
      PRINT *,' '
      PRINT 11
      READ *,i
      IF (I. EQ. 1) THEN
        PRINT 13
        READ (*,'(A10)') buf(16:25)
        modified = .true.
      ELSEIF (I. EQ. 2) THEN
        PRINT 14
        READ (*,'(A11)') buf(5:15)
        modified = .true.
      ELSEIF (I .EQ. 3) THEN
        PRINT 15
        READ (*,'(A55)') buf(26:80)
        modified = .true.
      ELSEIF (I .EQ. 4) THEN
        IF (modified .EQ. .true.) THEN
          REWRITE(10,err=441) buf
          PRINT *,phone,'  ',buf(5:70)
          GOTO 1
        ENDIF
      ELSE
        PRINT *,'invalid option, try again'
        GOTO 401
      ENDIF
      GOTO 401

402   PRINT *,'record does not exist, try again'
      GOTO 1
441   PRINT *,' rewriting the record failed'
      GOTO 1
```

```
                   C********   MENU
                   500   GOTO 1

                   C********  EXIT
                   600   CLOSE (10)
                         STOP

C**************************************************************
C  DUMP all the records in the ISAM file
C**************************************************************
700   phone = 0
      READ (10,keygt=phone,err=702,iostat=ii) buf  ! default is primary key
      PRINT *, phone,'  ',buf(5:76)
      DO i=1,100                                ! reading ISAM sequentailly
        READ (10,err=702,end=701,iostat=ii) buf
        PRINT *, phone,'  ',buf(5:)
      ENDDO
      GOTO 1
701   PRINT *,' number of records in the file :',(i-1),ii
      GOTO 1
702   PRINT *,' error in reading :',ii
      GOTO 1
1000  PRINT *,' open failed on isam '
      END
```

Index