HEWLETT
PACKARD

FORTRAN 77
Reference Manual

RTE-A and RTE-6/VM

Software Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 92836-90001
E1292

Printed in U.S.A. December 1992
Third Edition

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File or the Computer User's Documentation Index. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

Edition 1	Dec 1981	E1281
Update 1	Apr 1982	U0482
Update 2	Jun 1983	U0683
Reprint	Jun 1983	(Updates 1 and 2 Incorporated) I0683
Update 3	Feb 1985	U0285
Reprint	Oct 1985	(Update 3 Incorporated) I1085
Update 4	Jan 1986	U0186
Reprint	Jan 1986	(Update 4 Incorporated) I0186
Edition 2	Aug 1987	(Rev. 5000) E0887
Update 1	Jul 1990	(Rev. 5020) U0790
Edition 3	Dec 1992	(Rev. 6000) E1292

Preface

This is the reference manual of the FORTRAN 77 programming language for the HP 1000 Computer System. The compiler for the FORTRAN 77 language operates under the RTE-6/VM and RTE-A operating systems. This manual refers to both the language and the compiler as FORTRAN 77.

Note In this manual, "FORTRAN" or "FORTRAN 77" means "FORTRAN 77 for the HP 1000 Computer System."

This manual is a reference manual and is not a tutorial. The user who is familiar with any FORTRAN language can easily find syntactic and semantic information by using the index. The experienced programmer can find syntactic information in the syntax charts in Appendix G, which contains its own cross-reference index.

This manual contains the following:

Chapter 1 is an introduction to FORTRAN 77. The language and the compiler are briefly described. Source file structure is shown.

Chapter 2 describes the rudiments of the FORTRAN 77 language. The character set is identified. Keywords and symbolic names are defined. Data types are described.

Chapter 3 provides information on all the statements in the FORTRAN 77 language. The required order of statements is defined, followed by an alphabetical listing of the statements.

Chapter 4 describes in more detail the input/output statements used in FORTRAN 77. All the format descriptors are defined, with examples showing their use.

Chapter 5 discusses the file handling statements available in FORTRAN 77. These include the $FILES directive and the READ, WRITE, OPEN, CLOSE, INQUIRE, and file positioning statements.

Chapter 6 discusses procedures and block data subprograms. The procedures include function subprograms, subroutine subprograms, statement functions, and intrinsic functions.

Chapter 7 discusses using FORTRAN 77, including control statements, compiler invocation, FORTRAN 77 messages, loading and running a program, and compiler directives.

Chapter 8 discusses ANSI 66 compatibility extensions, comparing 66 mode with 77 mode.

Appendix A lists and describes object program and compilation error messages. Included are library subroutine error messages, input/output run-time error messages, and compilation error messages.

Appendix B presents tables of intrinsic functions and describes various library functions and routines.

Appendix C displays the ASCII character set used by FORTRAN 77.

Appendix D shows the internal representation of all the data formats available in FORTRAN 77.

Appendix E compares FORTRAN 77 with the ANSI 77 standard. Backward compatibility is described, as are the MIL-STD-1753 extensions. FORTRAN 77 is also compared with FORTRAN 4X.

Appendix F describes a cross-reference table.

Appendix G provides complete syntax charts in *railroad normal form*.

Appendix H explains code and data separation (CDS), static and dynamic memory allocation, and recursion. Gives restrictions on mixing CDS and non-CDS programs.

Index is a cross-reference index of all topics covered in this manual.

Additional Documentation

More information on the RTE-6/VM and RTE-A Operating Systems and related utilities can be found in the following manuals:

- *Symbolic Debug/1000 User's Manual*, part number 92860-90001
- *Getting Started with RTE-6/VM*, part number 92084-90002
- *Getting Started with RTE-A*, part number 92077-90039
- *RTE-6/VM Quick Reference Guide*, part number 92084-90003
- *RTE-A Quick Reference Guide*, part number 92077-90020
- RTE-A *System Design Manual*, part number 92077-90013
- *RTE-6/VM Loader Reference Manual*, part number 92084-90008
- *RTE-A LINK User's Manual*, part number 92077-90035
- *RTE-6/VM Terminal User's Reference Manual*, part number 92084-90004
- *RTE-A User's Reference Manual*, part number 92077-90002
- *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005
- *RTE-A Programmer's Reference Manual*, part number 92077-90007

In this manual, the last two pairs of the above listed manuals are referred to as "the appropriate system reference manual" and "the appropriate programmer's reference manual."

Conventions

The following conventions are used in this manual:

Notation — **Description**

UPPERCASE

Within syntax descriptions, characters in uppercase must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example:

```
INTEGER
```

Valid entries:	integer	Integer	INTEGER
Invalid entries:	interger	Intger	INTE_GER

italics

Within syntax descriptions, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *variable* with the name of the variable you want to declare:

COMMON *variable*

punctuation

Within syntax descriptions, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.

{ }

Within syntax descriptions, braces enclose required elements. When several elements within braces are stacked, you must select one. In the following example, you must select ON or OFF:

```
      { ON}
$LIST {OFF}
```

[ ]

Within syntax descriptions, brackets enclose optional elements. In the following example, brackets around ,OFF indicate that the option and its delimiter are optional:

$INCLUDE *filename*[,OFF]

[...]

Within syntax descriptions, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:

[,*itemname*][...]

If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:

[*itemname*][,...]

|...|

Within syntax descriptions, a horizontal ellipsis enclosed in parallel vertical lines indicates that you can select more than one element that appears within the immediately preceding pair of brackets or braces. However, each element can be selected only one time. In the following example, you must select `,A` or `,B` or `,A,B` or `,B,A`:

```
{,A}
{,B} |...|
```

If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, you must select `A` or `B` or `A,B` or `B,A`. The first element cannot be preceded by a comma:

```
{A}
{B} |,...|
```

... ⋮

Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.

△

Within examples, the space symbol △ represents a blank. In the following example, the value 123.45 is preceded by three blanks:

```
△△△123.45
```

Table of Contents

Chapter 1
Introduction to FORTRAN 77

Chapter 2
Language Elements

Chapter 3
FORTRAN 77 Statements

Chapter 4
Input/Output

Chapter 5
FORTRAN File Handling

Chapter 6
Procedures and Block Data Subprograms

Chapter 7
Using FORTRAN 77

Chapter 8
ANSI 66 Compatibility Extensions

Appendix A
Error Messages

Appendix B
Intrinsic and Library Functions

Appendix C
HP Character Set

Appendix D
Data Format in Memory

Chapter E
FORTRAN Comparisons

Chapter F
Cross-Reference Table

Appendix G
FORTRAN 77 Syntax Charts

Appendix H
CDS Usage

List of Illustrations

Tables

1

Introduction to 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, among them Hewlett-Packard's RTE FORTRAN 4X, extended the standard. To expand 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.

HP's FORTRAN 77 has many extensions to provide a more structured approach to program development and more flexibility in computing for scientific applications. In this manual, wherever such an extension is described, it is 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, *MIL-STD-1753 Military Standard FORTRAN, DOD Supplement to American National Standard X3.9-1978.* In this manual, wherever such an extension is described, it is referred to as "a MIL-STD-1753 standard extension to the ANSI 77 standard."

FORTRAN 77 is completely compatible with both ANSI 77 and the previous 1966 standard (described in the document *American National Standard Programming Language FORTRAN, ANSI X3.9-1966,* published in 1966). Because some of the ANSI 66 and ANSI 77 standards conflict, FORTRAN 77 has a compiler option to switch to either 66 or 77 mode.

In 77 mode, FORTRAN 77 is completely compatible with the ANSI 77 standard. In 66 mode, FORTRAN 77 is completely compatible with the ANSI 66 standard. All features of FORTRAN 77 are available in both modes, but some features may function differently in 66 mode and 77 mode. Chapter 8 of this manual compares features of both modes, showing where they conflict and how programs compiled in each mode behave. Appendix E compares FORTRAN 77 with RTE FORTRAN 4X, the ANSI 66 standard version of FORTRAN for the HP 1000.

The FORTRAN 77 Compiler

The FORTRAN 77 compiler constructs object language programs from source language files written according to the rules of the FORTRAN 77 language described in this manual. The FORTRAN 77 compiler described in this manual is executable under RTE-6/VM and RTE-A. The code generated by the compiler, standard binary output files, can be loaded and executed under the RTE-6/VM and RTE-A Operating Systems. Details for specifying these files are in the appropriate system reference manuals.

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. Source and object listings can be produced if specified in the FORTRAN 77 control statement or the command line (see "FORTRAN Control Statement" and "Compiler Invocation" in Chapter 7 for details on the FORTRAN 77 control statement and the command line).

Note FORTRAN 77 for the HP 1000 is invoked through the command `FTN7X`, plus a source file name and optional parameters. Because of the command form, this FORTRAN compiler is often referred to as FORTRAN 7X.

FORTRAN 77 Vocabulary

A FORTRAN 77 source file is composed of one or more program units. Each of the program units is constructed from characters grouped into lines and statements.

Figure 1-1 below shows a sample listing of a FORTRAN 77 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 FORTRAN 77 source file terms that follow Figure 1-1 refer to the figure.

```
FTN,L                                     !Optional control statement.
      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(1,'(''ENTER value-->'')')     !Prompt user.
      READ *,n                            !Enter integer limit to sum.
*     Compute sum in subprogram nfunc.
      sum = nfunc(n)                      !Invoke subprogram.
      WRITE (1,33)  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)
      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 Listing of a FORTRAN 77 Source File

FORTRAN 77 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 and segments, 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 unit, a subprogram unit, or a block data subprogram unit. (`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. (Lines 1 through 18 are a main program.)

Subprogram

A FORTRAN 77 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 (lines 21 through 27); when headed by a SUBROUTINE statement, a subroutine subprogram; and when headed by a BLOCK DATA statement, a block data subprogram. Subprograms can also be written in other languages, such as FORTRAN 4X, Pascal/1000, or Macro/1000.

Overlay

An overlay is an overlayable set of statements beginning with a PROGRAM statement that specifies type 5. (See "Alternate PROGRAM Statement" in Chapter 3.)

Line

A line is a line of characters. All characters must be from the ASCII character set. (See Appendix C for the complete ASCII character set.) The character positions in a line are called columns and are consecutively numbered 1, 2, 3, ..., 72 from left to right. (1 through 27 are lines.) Columns 73 through 80 are placed in the listing but otherwise ignored. Columns 81 and greater are completely ignored.

Initial Line

An initial line is one that is not a comment line, and contains a blank or the digit 0 in column 6. Columns 1 through 5 can contain a statement label or blanks. (Lines 2, 8, 10 through 11, 13 through 15, 17 through 18, and 21 through 27 are initial lines.)

Continuation Line

A continuation line is one that contains any characters other than a blank or the digit 0 in column 6, and contains only blanks in columns 1 through 5. A continuation line can follow only an initial line or another continuation line (unless separated by a comment line). In all cases, a statement can be continued indefinitely. (Line 16 is a continuation line.)

Directive Line

A directive line is one that contains a $ in column 1 and the text of the directive to the compiler in columns 2 through 72. A directive line cannot be continued. (See "Compiler Invocation" and "Compiler Directives" in Chapter 7.)

Statement

A statement is an initial line optionally followed by continuation lines. The statement is written in columns 7 through 72 of the lines. 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, etc. (Lines 2, 8, 10 through 11, 13 through 18, and 21 through 27 are statements.)

Control Statement A control statement is an optional directive on the first line that tells the compiler how to compile the program. See “FORTRAN Control Statement” in Chapter 7.

Comment Line A comment line is denoted by a “C” or an “*” in column 1 of a FORTRAN 77 source file. (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. (Lines 8, 10, 11, 13, 16, 23, and 26 contain end-of-line comments.) Blank lines are also treated as comment lines.

Source File Structure

FORTRAN 77 is column sensitive. The compiler control statements and directives (FTN, $EMA, $PAGE, etc.) must begin in column 1. All other FORTRAN 77 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 the digit 0 for all lines except continuation, comment, directive, and control statement lines. A “C” or “*” in column 1 denotes a comment line. Columns 73 through 80 are printed in the listing but are otherwise ignored (historically, these columns were used for sequence numbers on Hollerith cards).

Figure 3-1 in Chapter 3 lists the ordering requirements of FORTRAN 77 statements within program units.

FORTRAN Software Files and Installation

The following files are shipped with the HP 92836A FORTRAN product:

”FTN7X	Installation Guide
#FTN7X	Loader Command File
A92836	Software Numbering File
$FCLBA	RTE-A / 6/VM Compiler Library
$F7XCS	Common Compiler Modules
%F7X1	Compiler Modules Part 1
%F7X2	Compiler Modules Part 2
&FRPLS	Source for Compiler Option RPLs
%FRPLS	Relocatable for Compiler Option RPLs
%FX000	Source Message Catalogs

After you have restored the above files, read the file ”FTN7X for information on how to install FTN7X on your system.

2

Language Elements

A FORTRAN 77 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 77 provides nine data types for constants, variables, functions, and expressions. FORTRAN 77 also provides three additional constant formats, which are extensions to the ANSI 77 standard. All are described in "Data Types and Constants" below. Keywords, special characters, special symbols, symbolic names, and constant values make up the statements of a FORTRAN 77 program. This chapter describes the elements of statements.

FORTRAN 77 Character Set

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

	Blank	(	Left parenthesis
=	Equals	)	Right parenthesis
+	Plus	,	Comma
−	Minus	.	Decimal point
*	Asterisk	’	Single quotation mark (apostrophe)
/	Slash	”	(Double) quotation mark
!	Exclamation point	$	Dollar sign
_	Underscore (break)	:	Colon
@	At sign		

The exclamation point, underscore, and at sign are extensions to the ANSI 77 standard. Although the dollar sign is a standard special character, it is not valid except in character strings in a standard program. See Appendix E for a description of extensions which use the exclamation point, underscore, and at sign; see Chapter 8 for a description of the extension which uses the dollar sign.

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

In addition, any printable ASCII character can be used in a character or Hollerith constant or a comment.

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

Special Symbols

The special symbols are groups of characters that define specific operators and values. The special symbols of FORTRAN 77 are:

//	Character concatenation		
**	Exponentiation	.EQ.	Equal
.TRUE.	Logical true	.NE.	Not equal
.FALSE.	Logical false	.LT.	Less than
.NOT.	Logical negation	.LE.	Less than or equal
.AND.	Logical AND	.GT.	Greater than
.OR.	Logical OR	.GE.	Greater than or equal
.XOR.	Exclusive OR		
.EQV.	Logical equivalence (exclusive NOR)		
.NEQV.	Logical nonequivalence (exclusive OR)		

.XOR. is an extension to the ANSI 77 standard.

Keywords

Keywords are predefined FORTRAN 77 entities that identify a statement or a compiler option. Symbolic names can be identical to keywords since the interpretation of a sequence of characters is implied by the context in which it appears. The keywords of FORTRAN 77 are listed below.

Statement Keywords

ASSIGN
BACKSPACE
BLOCK DATA
CALL
CHARACTER
CLOSE
COMMON
COMPLEX
CONTINUE
DATA
DIMENSION
DO

DOUBLE COMPLEX
DOUBLE PRECISION
ELSE
ELSE IF
EMA
END
END DO
END IF
ENDFILE
ENTRY
EQUIVALENCE
EXTERNAL

FORMAT
FUNCTION
GOTO
IF
IMPLICIT
INCLUDE
INQUIRE
INTEGER
INTRINSIC
LOGICAL
OPEN
PARAMETER

PAUSE
PRINT
PROGRAM
READ
REAL
RETURN
REWIND
SAVE
STOP
SUBROUTINE
THEN
WHILE
WRITE

Compiler Directive Keywords

ALIAS
CDS
CLIMIT
ELSE
ELSEIF

EMA
ENDIF
FILES
IF
IFDEF

IFNDEF
INCLUDE
LIST
MSEG
OPTPARMS

PAGE
SET
TITLE
TRACE

Symbolic Names

Symbolic names are entities that define main program, procedure, block data subprogram, common block, named constant, or variable names. Each symbolic name consists of a sequence of characters, the first of which must be a letter. The rest can be letters, digits, or the underscore character (_). The letters can be uppercase or lowercase. The name can be any length, but only the first 16 characters are significant. Main program names may be truncated to five characters by the operating system.

Examples of Symbolic Names

```
INITIALIZATION_SUBROUTINE        REAL_VALUE
char_string                      sum_of_real_values
NumBer_of_ERRors                 error_flag
```

Notice that, because only the first 16 characters are significant, the compiler would consider INITIALIZATION_SUBROUTINE and INITIALIZATION_SUBPROGRAM to be the same name. Since uppercase and lowercase letters are not distinguishable within symbolic names, the following are equivalent:

```
result3
RESULT3
ResulT3
```

Names longer than six characters and the use of underscore or lowercase are extensions to the ANSI 77 standard.

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 or double integer, depending on compiler options. Any other letter implies type real. This default implied typing can be changed with an IMPLICIT statement or 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 since 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 77:

```
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 statements.

```
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 the language permits the above examples, using symbolic names that look like keywords is not good programming practice because it inhibits program readability.

Intrinsic Functions

Intrinsic functions are symbolic names that are predefined by FORTRAN 77. Intrinsic functions are discussed in detail in Chapter 6 and Appendix B. The intrinsic functions of FORTRAN 77 are listed in Tables 2-1 through 2-4.

If a user-defined symbolic name that is the same as a predefined symbolic name is used, any use of that name in the same program unit refers to the user-defined name. (That is, the intrinsic function of that name cannot be used in the same program unit.) Also see "EXTERNAL Statement" in Chapter 3.

Table 2-1. ANSI 77 Intrinsic Functions

ABS	ASIN	CSIN	DIM	DSQRT	INT	MINMIN0
ACOS	ATAN	CSQRT	DINT	DTAN	ISIGN	MIN1
AIMAG	ATAN2	DABS	DLOG	DTANH	LEN	MOD
AINT	CABS	DACOS	DLOG10	EXP	LGE	NINT
ALOG	CCOS	DASIN	DMAX1	FLOAT	LGT	REAL
ALOG10	CEXP	DATAN	DMIN1	IABS	LLE	SIGN
AMAX0	CHAR	DATAN2	DMOD	ICHAR	LLT	SIN
AMAX1	CLOG	DBLE	DNINT	IDIM	LOG	SINH
AMIN0	CMPLX	DCOS	DPROD	IDINT	LOG10	SNGL
AMIN1	CONJG	DCOSH	DSIGN	IDNINT	MAX	SQRT
AMOD	COS	DDIM	DSIN	IFIX	MAX0	TAN
ANINT	COSH	DEXP	DSINH	INDEX	MAX1	TANH

Table 2-2. MIL-STD-1753 Extensions

BTEST	IBCLR	IBSET	IOR	ISHFTC	MVBITS
IAND	IBITS	IEOR	ISHFT	IXOR	NOT

Table 2-3. HP Extensions

ACOSH	ATANH	DACOSH	DATANH	DEXEC	IMAG	REIO
ASINH	CTAN	DASINH	DCMPLX	EXEC	PCOUNT	XLUEX
						XREIO

Table 2-4. Compatibility Extensions

ALOGT	DATN2	DDINT	DLOGT	ISSW

Data Types

Every constant, variable, function, and 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 permissible operations on an entity of that type

The nine data types provided by FORTRAN 77 are:

- integer
- double integer
- real
- double precision
- complex
- double complex
- logical
- double logical
- character

Each is discussed below 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. With the PARAMETER statement, constants can be given symbolic names.

FORTRAN 77 provides three additional constant formats:

- Hollerith
- octal
- hexadecimal

These formats are extensions to the ANSI 77-standard. They differ from the data types in that they cannot be associated with variables, functions, or expressions. Only constants can be of these types.

The operations allowed on each of the types are discussed in "Expressions" later in this chapter. Each type statement mentioned below is discussed in detail in "Type Statement" in Chapter 3. Appendix D shows the data format in memory of each type.

Integer

The integer type defines the set of signed whole numbers in the range −32768 to 32767. Variables of type integer are stored in one 16-bit word.

A variable can be explicitly typed as integer by specifying it in an INTEGER type statement (depending on compiler options) or in an INTEGER*2 type statement, or implicitly in an IMPLICIT statement. If not explicitly typed, a symbolic name with a first letter of I, J, K, L, M, or N is type integer.

Integer constants consist of an optional plus (+) or minus (−) sign followed by one or more digits (0 through 9). Whole numbers outside the integer range are represented as double integers.

When an integer constant has an I suffix, the constant is 16 bits long, regardless of compiler options.

Examples

```
0        -638      123I (even with the J option)
45       -32767
```

Double Integer

The double integer type defines the set of signed whole numbers in the range −2147483648 to 2147483647. Variables of this type are stored in two 16-bit words.

A variable can be explicitly typed as double integer by specifying it in an INTEGER*4 type statement, or it can be implicitly typed in an IMPLICIT statement. When the J compiler option is used, any symbolic name declared in an INTEGER type statement or with a first letter of I, J, K, L, M, or N implies type double integer.

Double integer constants consist of an optional plus (+) or minus (−) sign followed by one or more digits (0 through 9). Only whole numbers that are outside the range of integer but within the range −2147483648 to 2147483647 are represented as double integer constants. Constants outside this range cause compile-time errors. Numbers outside the range assigned to, or read into, variables cause an overflow or underflow condition. The compiler does not detect these, and erroneous results can be produced. It is the programmer's responsibility to check for overflow and underflow.

When an integer constant has a J suffix, the constant is 32 bits long, regardless of the I and J compiler options.

Examples

```
-99526          32768          123J
2147483647      -3 (with the J option)
```

Real

The real type defines a set of real numbers. Real values, often called *floating-point*, fall in these ranges:

$-1.70141 \times 10^{+38}$ to $-1.469368 \times 10^{-39}$

0.0

1.469368×10^{-39} to $1.70141 \times 10^{+38}$

Entities of type real are stored in two 16-bit words and have an accuracy of approximately 6.6 to 6.9 decimal digits (that is, one part in $10^{6.6}$ to $10^{6.9}$).

A variable can be explicitly typed as real by specifying it in a REAL or REAL*4 type statement. If not explicitly declared in a type statement, a symbolic name with a first letter of A through H or O through Z is type real unless otherwise specified in an IMPLICIT statement.

Real constants contain a decimal point, an exponent, or both. They can have a leading plus (+) or minus (−) sign.

Syntax

sn.n
s.n
sn.
*sn.*E*se*
*sn.n*E*se*
*sn*E*se*
*s.n*E*se*

where:

n is a string of digits.

s is the optional sign.

e is the exponent, which must be an integer.

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

`3.4E-4` = 3.4 x 10^{-4} = `.00034`

`42.E2` = 42 x 10^{2} = `4200.`

Examples

```
8.5               5.E+04

-.6               2E-15

3.                .18181E-2

3.14159E2
```

Double Precision

The double precision type defines a set of real numbers. Double precision values have the following range:

$-1.70141183460469232 \times 10^{+38}$ to $-1.46936793852785946 \times 10^{-39}$

0.0

$1.46936793852785938 \times 10^{-39}$ to $1.70141183460469227 \times 10^{+38}$

Entities of type double precision are represented in four 16-bit words and have an accuracy of approximately 16.3 to 16.6 decimal digits (that is, one part in $10^{16.3}$ to $10^{16.6}$).

A variable can be explicitly typed as double precision by specifying it in a REAL*8 or DOUBLE PRECISION type statement, or it can be implicitly typed in an IMPLICIT statement

Double precision constants contain an optional decimal point and an exponent. They can have a leading plus (+) or minus (−) sign. The exponent is specified with the letter D.

The syntax for the double complex type is the same as the syntax for the real type, except that double complex uses a D in the exponent part. The D is required.

Examples

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

Complex

The complex type defines a set of complex numbers. The representation of a complex entity is an ordered pair of real values. The first of the pair represents the real part of the complex value, and the second represents the imaginary part. Each part has the same degree of accuracy as for a real value. Values of type complex are represented in four consecutive 16-bit words.

A variable can be explicitly typed as complex by specifying it in a COMPLEX or COMPLEX*8 type statement, or it can be implicitly typed in an IMPLICIT statement.

The form of a complex constant is an ordered pair of real or integer constants separated by a comma and surrounded by parentheses.

Examples

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

Double Complex

The double complex type is an extension to the ANSI 77 standard. It defines a set of complex numbers. The representation of a double complex value is an ordered pair of double precision values. The first of the pair represents the real part of the double complex value, and the second represents the imaginary part. Each part has the same degree of accuracy as for a double precision value. Values of type double complex are represented in eight consecutive 16-bit words.

A variable can be explicitly typed as double complex by specifying it in a COMPLEX*16 or DOUBLE COMPLEX type statement, or it can be implicitly typed in an IMPLICIT statement.

The form of a double complex constant is an ordered pair of constants separated by a comma and surrounded by parentheses. One of the pair of constants must be double precision, while the other can be double precision, real, or integer.

Examples

```
(1.56792456774D-24,-9.74375486354D-21)
(0,5.99537D5)
(-153D-12,4.66257)
```

Logical

An entity of the logical type can assume only the values true or false. Entities of type logical are represented in one 16-bit word. The values true and false are represented internally by, respectively, a 1 or a 0 in the most significant bit (bit 15) of the word. The lower 15 bits of a logical value are not defined, and therefore each can be 0 or 1.

A variable can be explicitly typed as logical by specifying it in a LOGICAL (depending on compiler options) or LOGICAL*2 type statement, or it can be implicitly typed in an IMPLICIT statement.

The forms and values of a logical constant are:

Form	Value
.TRUE.	true
.FALSE.	false

The periods must be included as shown when specifying a logical constant.

Double Logical

The double logical type is similar to the logical type except that entities of type logical are represented in two 16-bit words. Only the most significant bit of the first word is used.

A variable can be explicitly typed as double logical by specifying it in a LOGICAL*4 type statement or, with the J compiler option specified, in a LOGICAL type statement, or it can be implicitly typed in an IMPLICIT statement.

The double logical type is included for alignment of data in common blocks and equivalenced data. Otherwise double logical is identical to the logical type.

Character

The character type is used to represent a string of characters. The string can consist of any characters in the 8-bit ASCII character set. Most nonprintable characters can be included in a string, but it is recommended that nonprintable characters be specified with the CHAR intrinsic function and concatenated to a string. The CHAR intrinsic function is shown in Appendix B. The blank character is valid and significant in a character value. Lowercase characters are not identical to their uppercase equivalents in character values.

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

Entities of type character are stored two characters per 16-bit word, with each character occupying 1 byte (8 bits). If the last character is in the left byte of a word, the right byte (that is, the right 8 bits) is not part of the entity. The first character does not necessarily start in the left byte of a word; that is, character values are not word-aligned.

A variable can be explicitly typed as character by specifying it in a CHARACTER type statement, or it can be implicitly typed in an IMPLICIT statement.

The form of a character constant is a single quotation mark (apostrophe, ') followed by a nonempty string of characters followed by a single quotation mark.

If an apostrophe or single quotation mark is included in a string delimited by single quotation marks, it must be written twice to distinguish it from the delimiting characters.

The length of a character constant is the number of characters between the delimiting single quotation marks (which are not counted). Double apostrophes count as one character. The length of a character constant must be greater than 0.

As an extension to the ANSI 77 standard, a one-character constant may be specified as CHAR (*n*), where *n* is an integer constant.

Examples

```
   'Input the next item'
   'EXPECTING A "1" OR A "2"'
   'That''s life!'
   'FILE1:SU:-48'
   'Item #1 =>'
```

Note Character strings are not initialized. You can initialize them using a DATA statement or an assignment statement.

Hollerith Constants

Hollerith constants are an extension to the ANSI 77 standard. They are a special format for ASCII characters and are stored as numeric values. A Hollerith constant consists of an integer specifying the number of characters (including blanks), followed by the letter H and the character string.

Hollerith constants are stored as the following types:

Number of Characters	Compiler Option	Type
1–2	I	Integer
1–2	J	Double Integer
3–4	I or J	Double Integer
3–4	I	Real
5–8	I or J	Double Precision

(The table does not apply in 66 mode.)

A Hollerith constant with greater than eight characters is legal only as an actual argument in a CALL statement or function reference, or in a DATA or FORMAT statement.

Examples

```
2H$$            6H&PROGA
8HA STRING      3H12A
12HReport Title 7HQU'OTED
```

Note Since Hollerith constants offer no advantages over character constants and are less flexible, they should be avoided.

Octal Constants

Octal constants are an extension to the ANSI 77 standard. They are a special format of octal values that are stored as integers or double integers, according to the rules described for those types. There are two formats for octal constants, the B-form and the O-form.

Syntax

*sn*B   the B-form

o'*n*'   the O-form

where

n   is an integer constant containing only the digits 0 through 7.

s   is the optional sign and can be used only in the B-form.

The B-form of octal constants can be used anywhere an integer or double integer constant can be used. The O-form adheres to the MIL-STD-1753 standard extensions to the ANSI 77 standard. These constants can appear in DATA statements only. With either form, if the constant fits into 16 bits and the list item is double integer, the first word is 0, regardless of the sign of the second word.

Examples

```
400B    O'2137'

100000B O'177777'
```

Hexadecimal Constants

Hexadecimal (base 16) constants are a MIL-STD-1753 extension to the ANSI 77 standard.

Syntax

Z'*n*'

where

n   is a constant containing the digits 0 through 9 and hexadecimal digits A through F or a through f (representing the decimal equivalents of 10 through 15).

Hexadecimal constants are a form of hexadecimal values that are stored as integers or double integers, according to the rules of those types. These constants can appear in DATA statements only.

If the constant fits into 16 bits and the list item is double integer, the first word is 0, regardless of the second word.

Examples

```
Z'F9A1'      Z'AB2'

Z'2782'      Z'FFFF'
```

Variables

A variable is a symbolic name that represents a data element whose value can be changed during program execution by assignment statements, READ statements, and so forth. Each variable can represent only one type of value: integer, double integer, real, double precision, complex, logical, double complex, double logical, or character.

A variable name can represent only one value (this is a simple variable), or it can represent a collection of values (this is an array). Individual elements in an array are referenced by the array name followed by a subscript (this is a subscripted variable).

See "Symbolic Names" above for a description of valid variable names.

Simple Variables

A simple variable is used for processing 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           sum_of_values
voltage         ERROR_FLAG1
Final_Score     array3_element(i,j)
i               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 the array. To designate exactly one data value, or element, of the array, follow the array name with one or more subscripts. A group of values arranged in a single dimension 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. The maximum number of dimensions is seven.

Array Declarators

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

Syntax

name(d1,d2,d3,...)

where

name is the symbolic name of the array.

d is a dimension declarator. There must be one dimension declarator for each dimension in the array. The syntax of a dimension declarator is:

n

or

m:n

where

m is the lower dimension bound.

n is the upper dimension bound.

If only the upper dimension bound is specified, the value of the lower dimension bound is 1. The value of either dimension bound can be positive, negative, or 0; 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 in which all constants, symbolic names of constants, and variables are of type integer or double integer. These expressions 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.

Note Using variables or asterisks in a dimension declarator is limited to declarators of formal arguments to subprograms. This is discussed in detail in Chapter 6.

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)...list(10)`.

A complete array declarator for a particular array can be specified only once 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 only (without dimensions or subscripts) can 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.

Subscripts

Subscripts designate a specific element of an array. An array element reference (a 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. Thus, 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`. The compiler does not generate an error if a subscript is used that is outside its declared lower and upper bounds, but the results are unpredictable.

A subscript can be an integer expression. The expression can contain subscripted variables and function references.

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 the array `chess_board`.
`arr(i+4,j-2)`	Subscripts `i+4` and `j-2` are expressions that represent specific elements of the array `arr` when evaluated.
`i((3*x+1)/4)`	If `x`=3.6, the expression evaluates to 2.95, which truncates to integer 2. Thus, the subscripted variable is `i(2)`.

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 number of words 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. Integer and logical arrays store each element of an array in a single 16-bit word; double integer, real, and double logical arrays store each element in two words; double precision arrays and complex arrays store each element in four words; double complex arrays store each element in eight words; while character arrays use half a word (1 byte) for each character. A one-dimensional array is stored as a linear list. Arrays of more than one dimension are stored in *column major order*, with the first subscript varying 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)`

Character Substrings

A character substring is a contiguous portion of a character variable. The form of a substring is:

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

or

a(*s1*[,*s2*]...) ([*first*]:[*last*])

where:

name is a character variable name.

a(*s1*[,*s2*]...) is a character array element name.

first is an integer expression that specifies the leftmost position of the substring; the default value is 1.

last is an integer expression that specifies the rightmost position of the substring; the default value is the length of the variable or array element.

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

$$1 \leq \mathit{first} \leq \mathit{last} \leq \mathit{len}$$

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

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

Expressions

An expression can be a constant, a simple or subscripted variable, a function reference, a substring, or combinations of these entities, joined by arithmetic, character, logical, or relational operators. There are four types of expressions:

- Arithmetic
- Character
- Relational
- Logical

Arithmetic expressions return a single value of type integer, double integer, real, double precision, complex, or double complex. Character expressions return character values. Relational and logical expressions evaluate to either true or false (that is, to a logical value).

Arithmetic Expressions

Arithmetic expressions perform arithmetic operations. An arithmetic expression can consist of a single operand, or it can consist of one or more operands together with arithmetic operators, parentheses, or both. An operand in an arithmetic expression can be an arithmetic constant, the symbolic name of an arithmetic constant, a variable, an array element reference, or a function reference. The arithmetic operators are:

+	Addition; unary plus (positive or plus sign)
–	Subtraction; unary minus (negation or minus sign)
*	Multiplication
/	Division
**	Exponentiation

A unary operator is one that affects one operand only. For example, the unary minus (also called a minus sign, or sign of negation) is used to designate that the expression following it is to be negated.

The following are valid arithmetic expressions:

```
a                num(i)
-4. + z          a**2
3.145            c**4)*d
SQRT(r + d)      total + sum_of_values
arr(5,2)*45.5    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; a*b must be used.

Division by a constant 0 will cause a compilation warning. Division by a variable containing the value 0 will produce an invalid result. There is no run-time check for division by zero. It is the programmer's responsibility to check for this condition.

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:

**	Exponentiation	highest
* /	Multiplication and division	
+ −	Addition and subtraction; unary plus and minus	lowest

Evaluation of operations within parentheses occurs 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 determine the value of the expression.

If an expression contains two or more operators of the same precedence, the order of evaluation is determined by the following rules:

Two or more exponentiation operations are evaluated from right to left.

Multiplication and division or addition and subtraction are evaluated from left to right.

Examples

2**3**a

Evaluation occurs as follows:

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

2***op1* is evaluated.

a/b*c

Evaluation occurs as follows:

a/b is evaluated to form *op1*.

*op1**c is evaluated.

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

Evaluation occurs as follows:

`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 be used to control the order in which the parts of an expression are evaluated. 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.

Examples

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

Evaluation occurs as follows:

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

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

*op2***`d` is evaluated.

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

Evaluation occurs as follows:

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.

Two arithmetic operators cannot appear together unless one of the operators is enclosed in parentheses. For example, `a*(-3)` is allowed, but `a*-3` is not.

Expressions with Mixed Operands

Integer, double integer, real, double precision, complex, and double complex operands can be intermixed freely in an arithmetic expression. 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 operand type in the expression. Operand types rank from highest to lowest in the following order:

Double Complex	Highest
Complex	
Double Precision	
Real	
Double Integer	
Integer	Lowest

Exceptions: If the two operands have types double precision and complex, the result is double complex.

If the second operand of exponentiation (**) is of type integer or double integer, the result has the type of the first operand.

Note The precision of a subexpression is independent of any subsequent use of the value obtained from the subexpression. For example, if the I compiler option is used, `P = NINT (32768)` and `P = 1000*1000` both cause overflows, even if `P` is double integer. Any overflow is an error, and its results are unpredictable.

The compiler may evaluate certain expressions in a higher precision and thereby avoid an overflow. However, programs should not depend on this happening.

The conversion precedence for mixed type arithmetic expressions with the operators +, −, *, /, and ** is shown in Table 2-5. For example, in the expression `a*b-i/j`, if `a` and `b` are real variables and `i` and `j` are integer variables, then `a` is multiplied by `b` to form *op1*; `i` is divided by `j` with integer division, converted to real, and subtracted from *op1*. The result is an expression of type real.

Table 2-5. Conversion of Mixed-Type Operands

op2

		Integer	Double Integer	Real	Double Precision	Complex	Double Complex
	Integer	I	DI	R	DP	C	DC
	Double Integer	DI	DI	R	DP	C	DC
op1	**Real**	R	R	R	DP	C	DC
	Double Precision	DP	DP	DP	DP	DC	DC
	Complex	C	C	C	DC	C	DC
	Double Complex	DC	DC	DC	DC	DC	DC

where:	I	=	Integer
	DI	=	Double Integer (INTEGER*4)
	R	=	Real
	DP	=	Double Precision
	C	=	Complex
	DC	=	Double Complex

When any value is raised to an integer power, the operation is performed by repeated multiplications. When any value is raised to a power that does not have an integer type, the operation is performed by logarithms and exponentiation. Therefore, a negative number raised to a real or double precision power results in an operation undefined (UN) error, even if the power is a whole number such as 2.0.

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. The exponentiation operator (**) is not allowed unless the exponent is of type integer. Note that variable, array element, and function references are not allowed.

Character Expressions

A character expression is used to express a character string. 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, or a character function reference. More complicated character expressions can be formed by using two or more character operands together with the character operator and parentheses. The character operator is:

// Concatenation

The interpretation of the expression formed with the character operator is:

c1 // *c2* Concatenate *c1* with *c2*

The result of a concatenation operation is a character string whose value is the value of *c1* contatenated 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'
file_name//'::'//crn
char (33B)//'H'//char(33B)//'J'
```

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. Note that variable, array element, substring, and function references are not allowed (except the special use of CHAR (*n*) as a character constant).

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

where:

*op*1 and *op2* are either both arithmetic expressions or both character expressions.

relop is a relational operator.

The relational operators are:

.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).

Arithmetic Relational Expressions

Arithmetic expressions as operands in a relational expression are evaluated according to the rules governing arithmetic expressions (defined above). If the expressions are of different types, the one with the lower rank is converted to the higher ranking type as specified in Table 2-2. 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.                i + j .GE. z + 1
a + b - c .LT. num         o .GT. p
```

Expressions of type complex or double complex 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.

Hollerith data of types integer and double integer can be used with all six relational operators. Hollerith data of other types may not compare correctly, because of the behavior of floating-point data (see Chapter 8).

Character Relational Expressions

Character relational expressions are used to compare two operands, each of which is a character expression. The character expressions are first 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 the two operands is determined by the greater of these two characters. Thus, the ranking of the operands is determined only by the first character position at which the two operands differ. If there is no such position, then 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. See Appendix C for the ASCII collating sequence.

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

Examples

```
IMPLICIT CHARACTER*6 (a-n)      ! All variables beginning with the
                                ! letters a-n are of this type.
'the'.LT.'there'
'MAY 23'.GT.'MAY 21'
name .LE. 'PETERSEN'
char_str1 .GE. char_str2
first .EQ. a_string(2:8) // 'COD'
```

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 together with logical operators, parentheses, or both. An operand in a logical expression can be a logical constant, the symbolic name of a logical constant, a logical variable, a logical array element reference, a logical function reference, or a relational expression. The logical operators are:

.NOT.	Logical negation (unary)
.AND.	Logical AND
.OR.	Inclusive OR
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence (exclusive OR)

The unary operator .NOT. takes 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 are 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 .EQV. operator returns a value of true if the logical operands on either side of the .EQV. operator are both true or both false.

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

Table 2-6 is a truth table for the logical operators.

Table 2-6. Truth Table for Logical Operators

a	*b*	`.NOT.` *a*	`.NOT.` *b*	*a* `.AND.` *b*	*a* `.OR.` *b*	*a* `.NEQV.` *b*	*a* `.EQV.` *b*
True	True	False	False	True	True	False	True
True	False	False	True	False	True	True	False
False	True	True	False	False	True	True	False
False	False	True	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. lowest

Thus, .NOT. operations are performed before all other operations; .EQV. and .NEQV. operations are performed after all other operations. If there is more than one operator of the same precedence, evaluation occurs from left to right.

Examples

`a .OR. b .AND. C`

Evaluation occurs as follow:

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

`a .OR.` *lop1* is evaluated.

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

Evaluation occurs 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.

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

Evaluation occurs as follows:

`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.

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

Evaluation occurs as follows:

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

`a .AND.` *lop1* is evaluated.

As shown in the last example above, 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 here, 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 and double integer operands to perform bit masking operations. You must be aware of the internal binary representations of the data to use the masking operators with predictable results. (See Appendix C for details on data representation in memory.)

A complete truth table is shown in Table 2-7 (a duplicate of Table 2-6 with true = 1 and false = 0). A bit by bit comparison is done of the operands (i and j in Table 2-7), and the corresponding bit of the expression's result is set according to the truth table.

Care must be taken when using bit masking in the same expression as relational operators. The expression:

```
   (status .AND. mask .NE. 0)
```

is evaluated as:

```
   (status .AND. (mask.NE. 0))
```

which is incorrect. The above expression should be written as:

```
   ((status .AND. mask) .NE.)
```

or

```
   (IAND (status,mask) .NE. 0)
```

Note that FORTRAN 77 also supplies these bit masking operations and other bit manipulation operations as intrinsic functions. These are described in Appendix B. The FO RTRAN intrinsic functions comply with the MIL-STD-1753 extensions to the ANSI 77 standard.

Table 2-7. Truth Table for Masking Operators

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

Examples

.AND. returns the logical product of two operands:

op1	0111111111111111 (32767_{10})
op2	0001011001011001 (5721_{10})
result	0001011001011001 (5721_{10})

.NEQV. returns the symmetric difference of two operands:

op1	0000000011111111 (255_{10})
op2	0001011001011001 (5721_{10})
result	0001011010100110 (5798_{10})

Comments in FORTRAN 77

FORTRAN 77 uses two types of comments: comment lines and embedded comments. A comment line is denoted by a C or * in column 1, or a blank line in a source file. 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. Only comment lines beginning with an asterisk (*) are included in mixed listings.

An exclamation point (!) after a statement on the same line indicates the beginning of an embedded comment, unless the exclamation point is part of a character or Hollerith constant. The compiler ignores the exclamation point and any text following it; that is, it treats them as blanks. This use of the exclamation point is a FORTRAN 77 extension to the ANSI 77 standard.

3

FORTRAN 77 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. Each of the statements is then described in detail, in alphabetical order. Each statement description includes statement syntax, applicable rules, and examples.

A FORTRAN statement has the following general form:

[label] statement

The label is used to identify 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, 300, or 30 0`	identical
`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 0 or a blank in column 6. Columns 1 through 5 of a continuation line must be blank. Each statement can have an unlimited number of continuation lines.

A FORTRAN statement can be one of two types: executable or nonexecutable. Executable statements specify an action that the program is to take. Nonexecutable statements contain information such as the characteristics of operands, types 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	
Arithmetic IF Statement	END IF Statement
ASSIGN Statement	ENDFILE Statement
Assignment Statement	GOTO Statement
BACKSPACE Statement	INQUIRE Statement
Block IF Statement	Logical IF Statement
CALL Statement	OPEN Statement
CLOSE Statement	PAUSE Statement
CONTINUE Statement	PRINT Statement
DO Statement	READ Statement
DO WHILE Statement	RETURN Statement
ELSE Statement	REWIND Statement
ELSE IF Statement	STOP Statement
END Statement	WRITE Statement
END DO Statement	
Nonexecutable Statements	
BLOCK DATA Statement	FUNCTION Statement
COMMON Statement	IMPLICIT Statement
COMPLEX Statement	INTRINSIC Statement
DATA Statement	PARAMETER Statement
DIMENSION Statement	PROGRAM Statement
EMA Statement	SAVE Statement
ENTRY Statement	Statement Function Statement
EQUIVALENCE Statement	SUBROUTINE Statement
EXTERNAL Statement	Type Statement
FORMAT Statement	

Statement Categories

Executable and nonexecutable statements can be further grouped into seven functional categories:

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

The statements belonging to each of these categories are shown in Table 3-2.

Table 3-2. Classification of Statements

Statement	Description
Program Unit Statements	
BLOCK DATA	Identifies the program unit as a block data subprogram.
END	Specifies the end of a program.
ENTRY	Provides an alternate entry into a function or subroutine.
FUNCTION	Identifies the program unit as a function subprogram.
PROGRAM	Identifies the program unit as a main program.
RETURN	Transfers control from a subprogram back to the calling program.
Statement Function	Defines a one-statement function.
SUBROUTINE	Defines a program unit as a subroutine subprogram.
Specification Statements	
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.
EMA	Declares local variables, formal agruments, or both to be in EMA.
EQUIVALENCE	Associates variables so that they share the same place in memory.
EXTERNAL	Identifies subprogram names used as actual arguments, nonintrinsics, or both.
IMPLICIT	Specifies the type associated with the first letter of a symbolic name.
INTRINSIC	Identifies intrinsic function names used as actual arguments.
PARAMETER	Defines named constants.
SAVE	Causes the value of an entity to be retained after execution of a RETURN or END statement in a subprogram.
TYPE	Assigns an explicit type to a variable.
Value Assignment Statements	
ASSIGN	Assigns a value to a variable used in a GOTO or FORMAT statement label.
Assignment	Assigns values to variables at execution time.
DATA	Assigns initial values to variables before execution.

Table 3-2. Classification of Statements (continued)

Statement	Description
Control Statements	
CALL	Transfers control to an external procedure.
CONTINUE	Causes execution to continue; has no effect of itself.
DO	Causes a group of statements to be executed a specified number of times.
DO WHILE	Causes repeated execution of a group of statements while a condition is true.
GOTO	Transfers control to a specified statement.
Arithmetic IF	Transfers control based on a condition.
Logical IF	Conditionally executes a statement based on a logical value.
Block IF	Executes optional groups of statements based on one or more conditions.
Input/Output Statements	
BACKSPACE	Positions a file at the previous record.
CLOSE	Terminates access to a file.
ENDFILE	Writes an end-of-file mark.
FORMAT	Describes how data records are to be read or written.
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.
WRITE	Transfers data out.
Program Halt Statements	
PAUSE	Causes a program suspension.
STOP	Terminates program execution.

Order of Statements

Each of the following statements can appear only as the first statement in a program unit: PROGRAM, SUBROUTINE, FUNCTION, and BLOCK DATA.

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

- FORMAT and ENTRY statements can appear anywhere.
- All specification statements must precede all DATA statements, statement function statements, and executable statements.
- All statement function statements must precede all executable statements.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements. 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.

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td rowspan="2">PARAMETER Statement</td><td>IMPLICIT Statement</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

Figure 3-1. Required Order of Statements

ASSIGN Statement

The ASSIGN statement associates a statement label with an integer variable.

Syntax

```
ASSIGN label TO variable
```

where

label is a statement label.

variable is a simple integer variable.

Example	**Notes**
`ASSIGN 10 TO label1`	The statement label `10` is assigned to the variable `label1`.

The statement label can only refer to an executable statement or a FORMAT statement. The variable defined as a label can then be used in an assigned GOTO statement or as the format specifier in an input/output statement.

Example **Notes**

```
    ASSIGN 20 TO last1
      .
      .
    GOTO last1
```

The statement label `20` is assigned 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, k1, ave1
```

The statement label `100` is assigned to the variable `form1`. The label is that of a FORMAT statement.

A variable must be defined with a statement label value when referenced in an assigned GOTO statement or 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.

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.

Assignment Statement

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

- Arithmetic
- Logical
- Character

Arithmetic Assignment Statement

Syntax

var = *expr*

where:

var is a variable or array element of one of the following types:

INTEGER or INTEGER*2
INTEGER*4
REAL or REAL*4
DOUBLE PRECISION or REAL*8
COMPLEX
DOUBLE COMPLEX or COMPLEX*16

expr is an arithmetic expression.

Examples	**Notes**
`total = sub total + 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.

If the type of the variable on the left of the equals 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 below, followed by examples.

Note The expression is evaluated before the variable preceding the equals sign is considered. If the expression overflows, the value is incorrect even if the variable before the equals sign could have held the larger result.

In Table 3-3, *k* and *n* represent all possible combinations of byte sizes for the particular data type. For example, if *var* is INTEGER**k* and expr is REAL**n*, *k* is 2 or 4, while *n* is 4 or 8. This represents the four possible combinations of byte sizes.

Table 3-3. Type Conversion Rules for Arithmetic Assignment Statements of the Form var=expr

Var Type	Expr Type	Rules
a. INTEGER*_k_	INTEGER*_n_	If _k_≥_n_, assign INTEGER*k. If _k_<_n_, assign least significant word to _var._ See NOTE 1.
b. INTEGER*_k_	REAL*_n_	Truncation.
c. INTEGER*_k_	COMPLEX*8	Real part is REAL*4. Apply rule b to real part. Imaginary part is not used.
d. INTEGER*_k_	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._ See NOTE 3.
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 as real part; imaginary part = 0.
j. COMPLEX*16	INTEGER or REAL	Convert to REAL*8 by rule e or f and assign as real part; imaginary part = 0.
k. COMPLEX*_k_	COMPLEX*_n_	Apply rule f to real and imaginary parts independently.

NOTES:

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 the interpretation of the least significant word, and therefore incorrect.

2. When converting from INTEGER*4 to REAL*4, the precision can be lost because REAL*4 holds only 23 significant bits and INTEGER*4 holds 31 significant bits. If this occurs, the value is truncated and the least significant bits are lost.

3. When converting to a higher precision, the additional bits are set to zero. Converting a lower-precision constant to a higher precision does not necessarily yield the value of that constant written in the higher precision; for example, `DBLE(1.3)` does not equal `1.3DO`; the least significant 32 bits are different.

Table 3-4 below gives some examples of type conversions.

Table 3-4. Examples of Type Conversions for Arithmetic Assignment Statements of the Form var=expr

Rule	Var Type	Var Value	Expr Value	Expr Type
a	INTEGER*4	542	542	INTEGER*2
a	INTEGER*2	See NOTE 2 for Table 3-3.	86420	INTEGER*4
b	INTEGER*2	3	3.842	REAL*4
c	INTEGER*2	502	(5.0297E2,1.27E−5)	COMPLEX*8
d	INTEGER*4	−48170	(−4.817D4,1.0096D7)	COMPLEX*16
e	REAL*4	59.	59	INTEGER*2
f	REAL*8	10.D+09 See NOTE 3 for Table 3-3.	10.E+09	REAL*4
f	REAL*4	1.70141E+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
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.) See NOTE 4 for Table 3-3.	14.23E−17	REAL*4

Logical Assignment Statement

Syntax

lvar = *lexp*

where:

lvar is a variable or array element of type logical.

lexp is a logical or relational expression.

Examples

```
LOGICAL logl
i = 10
logl = i .EQ.  10
```

```
LOGICAL log_res, flag_set
num = 100
flag_set = .TRUE.
log_res = NUM .GT. 200 .AND. flag_set
```

Notes

`logl` is assigned the value true because `i` equals 10.

`logical_res` is assigned the value false because num is not greater than 200.

Character Assignment Statement

Syntax:

cvar = *cexp*

where:

cvar is a variable, array element, or substring of type character.

cexp is a character expression.

Examples	**Notes**

`CHARACTER*10 employee_name`
`employee_name = 'EMILIE'`
	The variable `employee_name` is assigned the value `EMILIE`ΔΔΔΔ.

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

`CHARACTER address*20`
`address(1:4) = '1645'`
`address(6:13) = 'First St.'`
	The first through fourth characters of the variable `address` are assigned the value `1645` and the sixth through 13th are assigned the value `First St.` The fifth character is undefined.

`CHARACTER name*6`
`name ='MURRAY'`
`   :`
	The variable `name` is assigned the character string `MURRAY`.

`CHARACTER*4 color(6), k`
`k = 'blue'`	

`color(5) = k`
`   :`
	The fifth element of the array `color` is assigned the character string `blue`.

`color(4) = 'G' // name(4:6)`	The fourth element of the array `color` is assigned the character string `GRAY`.

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. The information that appears on the left and right sides of the assignment statement must not overlap.

BACKSPACE Statement

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

Syntax

```
          { unit}
BACKSPACE {([UNIT=]unit[,IOSTAT=ios][,ERR = label])}
```

where

unit is an integer expression (zero or positive) specifying the unit number of a sequential file.

ios is an integer variable for error code return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement in the same program unit as the BACKSPACE statement. If an error occurs during execution of the BACKSPACE statement, control transfers to the specified statement rather than aborting the program.

If the prefix `UNIT=` is omitted, *unit* must be the first parameter. Otherwise the order of parameters is flexible.

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 the variable `j`.

If the file is positioned at beginning-of-information (BOI), a BACKSPACE statement has no effect upon the file.

The BACKSPACE statement is allowed on files connected for direct access, but should be avoided to preserve program portability. Backspacing over records written using list-directed formatting can cause unpredictable results because the number of records produced by list-directed output is difficult to predict.

BLOCK DATA Statement

The BLOCK DATA statement is the first statement in a block data subprogram.

Syntax

BLOCK DATA [*name*] [,*comment*]

where:

name is an optional subprogram name.

comment is a string of zero to 86 characters. This is an extension to the ANSI 77 standard.

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

As an extension to the ANSI 77 standard, data in common blocks can be initialized in any program unit; block data subprograms are required only in programs that must conform completely to the ANSI 77 standard.

See Chapter 6 for more information on block data subprograms.

CALL Statement

The CALL statement references and transfers control to a subroutine.

Syntax

CALL *name* [([*arg1*[,*arg2*[,*arg3*...]]])]

where:

name is the name of the subroutine being referenced.

arg is an actual argument or an asterisk followed by a label, where the asterisk indicates an alternate return and the label is a statement label of an executable statement in the same program unit as the CALL statement.

Examples	Notes
`CALL print_forms(top,lh,rh)`	The subroutine `print_forms` is called. Three arguments are passed.
`CALL exit`	The subroutine `exit` is called. No arguments are passed.

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

When a CALL statement is executed, any expressions in the actual argument list are evaluated, then control passes to the subroutine. Upon return from the subroutine, execution continues with the statement following the CALL statement for a normal return. When an alternate return is taken, execution continues with the statement label in the actual argument list that corresponds to the return ordinal specified in the subroutine's RETURN statement. Subroutine subprograms, referencing subroutines, and alternate returns from a subroutine are all discussed in detail in Chapter 6.

CHARACTER Statement

See "Type Statement" later in this chapter for the syntax of CHARACTER and all other type statements.

CLOSE Statement

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

Syntax

```
CLOSE ([UNIT=]unit[,IOSTAT=ios][,ERR=label][,STATUS=stat])
```

where:

unit is an integer expression specifying the unit number of the file.

ios is an integer variable for error code return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement in the same program unit as the CLOSE statement. If an error occurs during execution of the CLOSE statement, control transfers to the specified statement rather than aborting the program.

stat is a character expression that determines the disposition of the file; *stat* is one of the following:

`'KEEP'` The file continues to exist after execution of the CLOSE statement.

`'DELETE'` The file does not exist after execution of the CLOSE statement.

If `STATUS` = *stat* is not specified, the assumed value is 'KEEP'. The STATUS specifier has no effect on scratch files, because scratch files are always deleted on CLOSE or at normal program termination.

If the prefix `UNIT=` is omitted, *unit* must be the first parameter. Otherwise the order of parameters if flexible.

Examples	**Notes**
`CLOSE (10)`	The file connected to unit 10 is disconnected. The file continues to exist.
`CLOSE (UNIT=6,STATUS='DELETE')`	The file connected to unit 6 is disconnected. The file no longer exists.
`CLOSE (5,IOSTAT=io_error,ERR=100)`	The file connected to unit 5 is disconnected and kept. If an error occurs, control transfers to statement 100, and the error code is stored in the variable `io_error`.

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.

The CLOSE statement is discussed in detail in Chapter 5.

COMMON Statement

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

Syntax

COMMON [/[*blockname1*]/] *list1* [[,]/[*blockname2*]/ *list2*[,] . . .]

where:

blockname is the name of a labeled common block. Each omitted *blockname* specifies blank common.

list is one or more simple variables, array names, or array declarators.

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; and d is placed in common block c.

In each COMMON statement, the variables following a block name are declared to be in common block *blockname*. If the first block name is omitted, all variables that appear in the first list are specified to be in blank common. Alternately, the appearance of two slashes with no block name between them declares the variables that follow to be in blank common.

The following data items must not appear in a COMMON statement:

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

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

Any 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.

Example	**Notes**
`INTEGER*2 b(3)`	The common block `blk1` uses 9 words of storage,
`COMMON /blk1/b,arr(3)`	`b` uses 3 (1 word per element), and `arr` uses 6 (2 words per element).

Common block storage is allocated at load time. Storage is not local to any one program unit. Data space is allocated as follows within the common block for arrays `b` and `arr` in the preceding example:

Word	Common Block
1	`b(1)`
2	`b(2)`
3	`b(3)`
4	`arr(1)`
5	
6	`arr(2)`
7	
8	`arr(3)`
9	

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 by name, type, or number of elements with those of any other program unit. The only consideration is the size of the common blocks referenced by the different program units. The size of an unlabeled (blank) common block can differ between program units, but a labeled common block should be the same size in all program units. For example, assume that the following COMMON statement appears in program unit 1:

```
COMMON/blocka/i(4),j(6),alpha,sam
CHARACTER*4 alpha
```

and the COMMON statement

```
COMMON/blocka/geo,m(10),india,jack
```

appears in program unit 2. `blocka` is the same size (14 words) in both program units. Thus, referencing `i(4)` in program unit 1 is equivalent to referencing `m(2)` in program unit 2, because both variables refer to the same word of the labeled common block. The correspondence between the variables in common in the two program units is shown in the following table:

Program 1 Reference	Common Block Word Number	Program 2 Reference
i(1)	1	geo
i(2)	2	
i(3)	3	m(1)
i(4)	4	m(2)
j(1)	5	m(3)
j(2)	6	m(4)
j(3)	7	m(5)
j(4)	8	m(6)
j(5)	9	m(7)
j(6)	10	m(8)
alpha(1:2)	11	m(9)
alpha(3:4)	12	m(10)
sam	13	india
	14	jack

When common blocks in SSGA or system-labeled common are used, the NOALLOCATE option of the $ALIAS directive is required. This causes the common block size to be defined in the block data subprogram. The block data subprogram is then required, and should precede other references to the common block in the load. See Chapter 7 for a description of the NOALLOCATE option.

The following example shows an unlabeled common block in program unit 2 that is a different size from the common block referenced in program unit 1:

```
Program unit 1: COMMON i(12)

Program unit 2: COMMON law(7)
```

The correspondence between the variables in common in the two program units is:

Program 1 Reference	Common Block Word Number	Program 2 Reference
i(1)	1	law(1)
i(2)	2	law(2)
i(3)	3	law(3)
i(4)	4	law(4)
i(5)	5	law(5)
i(6)	6	law(6)
i(7)	7	law(7)
i(8)	8	Unused
i(9)	9	Unused
i(10)	10	Unused
i(11)	11	Unused
i(12)	12	Unused

The ANSI 77 standard specifies that in a common block all items must be either character or noncharacter. As an extension to the ANSI 77 standard, FORTRAN 77 allows a mixture. However, noncharacter variables must begin on even byte addresses. If they do not, a warning is issued and the odd byte is skipped.

Some linkers require that when more than one labeled common block of the same name appear in a program unit, the largest must be loaded first. The largest blank common block must appear in the first program unit loaded.

COMPLEX Statement

See “Type Statement” later in this chapter for the syntax of COMPLEX and all other type statements.

COMPLEX*8 Statement

See “Type Statement” later in this chapter for the syntax of COMPLEX*8 and all other type statements.

COMPLEX*16 Statement

See “Type Statement” later in this chapter for the syntax of COMPLEX*16 and all other type statements.

CONTINUE Statement

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

Syntax

```
CONTINUE
```

Example

```
   DO 20 i = 1,10
10 x = x + 1
   y = SQRT(x)
   PRINT *,y
   IF (x .LT.  25.) GOTO 20
   GOTO 10
20 CONTINUE
```

Notes

Because the last statement in the loop is a GOTO statement, a CONTINUE statement is used to terminate the loop.

The CONTINUE statement should always be written with a label. CONTINUE is used to mark a point in the program where a label is needed but where you do not want to associate the label with any specific action. CONTINUE statements are useful with GOTO statements because they allow you to add statements either before or after the statement label during program development.

In earlier versions of FORTRAN, the CONTINUE statement was often used as the last statement in a labeled DO loop that otherwise would end in a prohibited statement such as a GOTO. The END DO statement now serves this purpose. If a CONTINUE statement is used elsewhere in a program, or if it is not labeled, the statement performs no function and control passes to the next statement.

DATA Statement

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

Syntax

DATA *varlist1*/*conlist1*/[[,]*varlist2*/*conlist2*/[,] . . .]

where:

varlist is one or more simple variable names, array names, array element names, substring names, or implied DO loops. For syntax and detailed information on implied DO loops, refer to "Implied DO Loops" under "DO Statement" below.

conlist is the list of constants to be assigned to the corresponding items in *varlist*. The form of an item in *conlist* is:

[*num**] *con*

where:

num is an integer or named constant; the default is 1.

* is the repeat specifier.

con is a constant or named constant that is repeated *num* times.

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.

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

`PARAMETER (init_val = -1)`
`DIMENSIONm(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.

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

The number of items in the constant list must agree with the number of variables in the variable list. If the list of variables contains an array name without a subscript, one constant must be specified for each element of that array. The elements of the array are considered to be in column major order. Each subscript in an array element in the variable list must be an integer or double integer constant expression.

Constants are assigned to their corresponding variables in a DATA statement according to the rules of the assignment statement, except that constants have the same precision as their corresponding variables.

Examples	**Notes**

`DOUBLE PRECISION d,e`
`DATA d/1.23/`
`e = 1.23`
	`d` is not equal to `e` because the least significant bits of `e` are zero. `1.23` is converted in an assignment statement to REAL*4.

`INTEGER*4 j,k`
`DATA j/100000B/`
`k = 100000B`
	`j` is not equal to `k` because `100000B` is converted to a 32-bit integer in the DATA statement (upper bits = zero) and to a 16-bit integer in the assignment statement, which is then sign-extended.

If a constant has a type of character or logical, the corresponding variable must be of the same type. If a constant is of any numeric type (integer, double integer, real, double precision, complex, or double complex), the corresponding variable can be of any numeric type.

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 constant is blank-filled on the right. If the constant is longer than the variable, the constant is truncated, losing characters from the right.

Any variable can be initialized in a DATA statement except:

- A variable that is a formal argument to a function or subroutine
- A variable in blank common within a BLOCK DATA subprogram

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 are equivalenced, only one can appear in a DATA statement.

Each subscript or substring expression in a DATA statement must be an integer constant expression, except that implied DO loop variables may be used in subscript and substring expressions.

DATA statements can be placed anywhere after specification statements in a program unit. Compile time is shortened if all DATA statements immediately follow the last specification statement (with no intervening statement function definitions or executable statements).

DATA statements can be used to assign initial values to variables placed in the EMA area. However, this is only supported on RTE-A. See “EMA Statement” section in this chapter for more information on the EMA area. If any EMA variables appear in DATA statements in a program, the RTE-A linker will change the program from an EMA program to a VMA program without warning. See the *RTE-A Programmer’s Reference Manual*, part number 92077-90007, for more information on EMA and VMA programming. Initialized VMA is not supported on the RTE-6/VM Operating System.

The use of character constants to initialize non-character data is a compatibility extension; see Chapter 8.

DIMENSION Statement

The DIMENSION statement defines the dimensions and bounds of arrays.

Syntax

```
DIMENSION name1(bounds1) [,name2(bounds2), . . . ]
```

where:

name(*bounds*) is an array declarator. (See "Array Declarators" in Chapter 2.)

name is the symbolic name of an array.

bounds is a dimension declarator. There must be one dimension declarator for each dimension in the array. The syntax of a dimension declarator is:

```
[n:]m
```

where:

n is the lower dimension bound.
m is the upper dimension bound.

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.

An array can have up to seven dimensions.

Examples	**Notes**

`INTEGER*2 arr1`
`DIMENSION arr1(-3:1,4)`
	In this example, the type statement specifies `arr1` as type integer; the name of the array, not the complete array declarator, appears. The DIMENSION statement causes 20 words of memory to be allocated for the array arr1. An equivalent declaration would be `INTEGER*2 arr1(-3:1,4)`.

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

DO Statement

DO statements come at the beginning of DO loops. A DO loop is a group of statements that is executed repeatedly zero or more times, or a list within one statement that is executed a specified number of times. There are four kinds of DO loops:

- 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.

DO Loop Execution

When a DO statement is executed, the following sequence occurs:

1. *init*, *limit*, and *step* are evaluated; *index* is set to *init*. (See the following sections for definitions of these terms.) If necessary, *init*, *limit*, and *step* are converted to the same type as *index*.
2. The trip count, or the number of times the loop executes unless abnormally terminated (for example, by a GOTO statement), is computed as:

   INT((*limit* – *init* + *step*)/*step*)
3. If the trip count 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 when *init* is less than *limit* and *step* is negative.
4. The range of the loop is executed.
5. *index* is incremented by the value of *step*. If the loop has not yet been executed the number of times computed for the trip count, the range of the loop is executed again.

Within the range of a DO loop, modification of *index*, *init*, *limit*, or *step* does not affect the number of iterations of the loop, because this value is established when the loop is entered. Changing the value of *step* does not affect the incrementing of *index*; *index* is incremented by the original value of *step*.

Example

```
   DO 10 i = 1,10,2
      WRITE (1,'(''i ='',I2)')i
      i = i-2
10 CONTINUE
```

Notes

Modification of `i` in this example does not result in an infinite loop. The loop executes five times, printing `i = 1` each time.

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, that is, the value that the variable would have had on the next iteration. Upon abnormal exit from the DO loop, the control variable retains its value at the time of exit.

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.

Syntax

DO [*label* [,]] *index* = *init*,*limit* [,*step*]

where

label is the statement label of an executable statement. In a labeled DO loop, this statement must follow the DO statement in the sequence of statements within the same program unit as the DO statement (see "Labeled DO Loops" below).

As an extension to the ANSI 77 standard, the label can be omitted, in which case an END DO statement terminates the loop (see "Block DO Loops" below).

index is a simple variable that controls the loop. Note: The index must not be an array element.

init is an expression that is the initial value given to *index* at the start of the execution of the DO statement.

limit is an expression that is the termination value for *index*.

step is an 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 1. *step* should not equal 0.

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 in the following:

Example

```
   DO 10 i = 1,3,.1
      WRITE (1,*) i
10 CONTINUE
```

Notes

This program is intended to increment i by 10ths. Instead, a warning is generated when the program is compiled. When .1 is converted to type integer, it becomes 0, but the increment must not be 0.

If *index* is a 1-word integer, *limit* and *init* can have any 1-word integer values, and the value *limit-init* can be as large as 65536. This is the maximum number of times the loop can execute. If *index* is a double integer, the value *limit-init* must not exceed 2147483647.

Labeled DO Loops

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. The terminating statement must not be one of the following:

- An unconditional GOTO statement
- An assigned GOTO statement
- An arithmetic IF statement
- Any of the four statements associated with the block IF statement:

  - An IF THEN statement
  - An ELSE statement
  - An ELSE IF statement
  - An ENDIF statement
- A RETURN statement
- A STOP statement
- An END statement
- Another DO statement
- A DO WHILE 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. This terminating END DO statement must have a label that matches the label of the DO statement. (A DO loop terminated with an unlabeled END DO statement is a block DO loop, described below.)

Examples **Notes**

```
    DO 100 i = 1,10
     ⋮
100 CONTINUE
```

Labeled DO loop. 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
```

Labeled DO loop. 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
```

Labeled DO loop. The group of statements terminating with the one labeled `300` is repeated 11 times. Athough this loop ends with an END DO statement, it is not considered a block DO loop. Notice that the label in the DO statement corresponds with the one in the END DO statement.

Block DO Loops

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" below), 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.)

Implied DO Loops

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.

Implied DO Loops in Input/Output Statements

Syntax

(*list*, *index* = *init*, *limit* [, *step*])

where

list is an input/output list. It can contain other implied DO loops.

index, init,
limit, and *step* have the same meaning as in labeled and block DO loops.

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, output, or initialized. 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 the same way as for DO loops. For example, the effect of the following statement:

```
PRINT *, (a, i = 1,3)
```

is to write the value of a three times. If a = 35.6, the output consists of one record as follows:

```
35.6 35.6 35.6
```

If the list of an implied DO loop contains several simple variables, each of the variables in the list is input or output for each pass through the loop. For example, the statement

```
READ *, (a, b, c, j = 1,2)
```

is equivalent to

```
READ *, a, b, c, a, b, c
```

An implied DO loop can also transmit arrays and array elements. For example,

```
DIMENSION b(10)
PRINT *, (b(i), i = 1,10)
```

results in the array b being written in the following order:

```
b(1) b(2) b(3) b(4) b(5) b(6) b(7) b(8) b(9) b(10)
```

If an unsubscripted array name is used in the list, the entire array is transmitted. For example, the result of the following statements:

```
DIMENSION x(3)
PRINT *, (x, i = 1,2)
```

is to write the elements of array x twice, as follows:

```
x(1) x(2) x(3) x(1) x(2) x(3)
```

The list can contain expressions that use the index value. For example:

```
DIMENSION a(10)
PRINT *, (i*2, a(i*2), i = 1,5)
```

Implied DO loops can also be nested. The form of a nested implied DO loop in an input/output statement is:

```
(((list,index1=init1,limit1,step1),index2=init2,limit2,step2)
...indexn=initn,limitn,stepn)
```

The expressions *init1*, *limit1*, and *step1* can use the current value of the outer indices *index2* through *indexn*, the expressions *init2*, *limit2*, and *step2* can use the current value of the outer indices *index3* through *indexn*, and so forth.

Nested implied DO loops follow the same rules as other nested DO loops. For example, the statement

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

produces the following output:

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

The first, or nested DO loop, is satisfied once for each execution of the outer loop. (See "Nesting DO Loops" below.)

Implied DO loops are useful for controlling the order in which arrays are output, that is, for outputting an array in row-major, as opposed to column-major, order (to produce a more natural listing). The next two example statements print an array a, dimensioned as a(2,3), with values

$$\boxed{\begin{matrix}1 & 3 & 5\\ 2 & 4 & 6\end{matrix}}$$

The statement

```
WRITE (1,'(3I2)') a
```

prints the array in column-major order, yielding

```
1 2 3
4 5 6
```

and the statement

```
WRITE (1,'(3I2)') ((a(i,j), j = 1,3), i = 1,2)
```

prints the array in row-major order, yielding

```
1 3 5
2 4 6
```

Implied DO loops in input/output statements are not used only with arrays. The following statement prints a table of degrees and the sine of each, in steps of 10 degrees.

```
WRITE (1,'(F4.0,F9.5)') (i,SIN(i*3.14159/180.), i = 0, 360, 10)
```

Implied DO Loops in DATA Statements

The format of a DATA statement containing an implied DO loop is:

`DATA` (*dlist*, *index* = *init*, *limit* [,*step*]) / *clist* /

where

dlist is a list of array element names and implied DO loops.

index, init, limit, and *step* have the same meaning as in labeled and block DO loops.

clist is the list of constants to be assigned to the corresponding items in *dlist*.

init, limit, and *step* are arithmetic expressions, just as in implied DO loops in input/output statements; they must be integer expressions containing only constants and indices of outer loops. *index, init, limit*, and *step* must all be integer.

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 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 to generate a full variable list.

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.

This example initializes parts of a one-dimensional character array:

```
CHARACTER char_array(5)*5
DATA ((char_array(i) (1:i), i = 1,10) /15*'x'/
```

The following example initializes a square array to the identity matrix (its main diagonal is 1s, and the rest of the array is 0s). It uses a WRITE statement with an implied DO loop to output the array in row order.

Example

```
DIMENSION id_array(10,10)
DATA ((id_array(i,j), j = i+1,10), i = 1,9)      /45*0/  ! upper
DATA (id_array(i,i), i = 1,10)                   /10*1/  ! diagonal
DATA ((id_array(i,j), i = j+1,10), j = 1,9)      /45*0/  ! lower
WRITE(1,'(10I2)') ((id_array(i,j), j = 1,10), i = 1,10)
END
```

The program produces this output:

```
1 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

DO WHILE Statement

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.

Syntax

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

where:

label is the statement label of an executable statement

logical_expression is an expression that produces a value of true or false.

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 this way. 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 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. (See "Ranges of DO Loops" below.)

Examples **Notes**

```
DO WHILE (i .NE. 999)
   READ(1,33) 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
```

Repeatedly increments `index` while the condition of the DO WHILE statement is true.

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, you should always use a separate terminating statement for each loop.)

Here is an example in which the terminating statement of the innermost loop occurs before the last statement of the preceding loop. The two outer loops have the same terminating statement.

```
     DO 100 i = 1,10
        DO 100 j = 1,10
          sum = 0
          DO 90 K=1,10
90             sum = sum + a(i,k) * b(k,j)
          c (i,j) = sum
100 CONTINUE
```

Control passes to the statement following statement 100 only after all iterations of all three loops are executed.

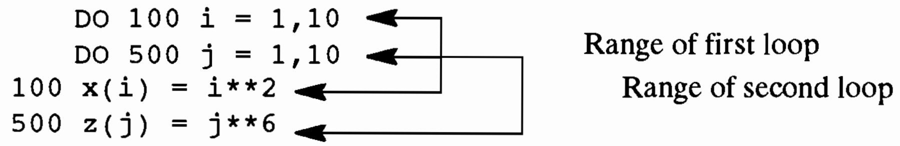
In a block DO or DO WHILE loop, each level must be terminated with a separate END DO statement.

Example	Notes

`DO WHILE (x .GE. 0)`
`   ⋮`
`   DO WHILE (y .LT. 10)`
`   ⋮`
`   END DO`
`END DO`
	One END DO statement is required for each level.

DO loops can be nested to as many levels as desired as long as the range of statements in any DO loop does not overlap the range of the preceding loop.

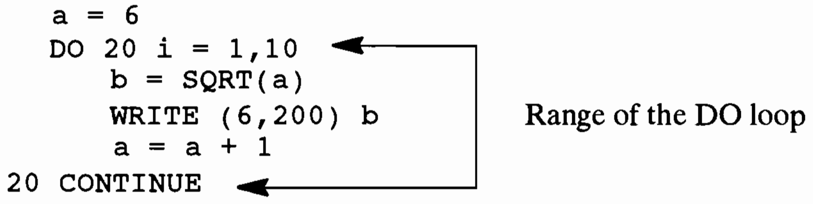
This example shows an illegal construction, one in which the ranges of two loops overlap.



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

Ranges of DO Loops

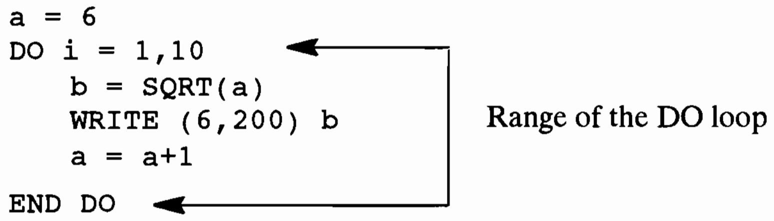
The range of a DO loop is defined as the first statement following the DO statement up to and including the terminal statement referenced by *label*, or, in the absence of *label*, up to and including the END DO statement. This example shows the range of a labeled DO loop:



```
  a = 6
  DO 20 i = 1,10
     b = SQRT(a)
     WRITE (6,200) b
     a = a + 1
20 CONTINUE
```

Range of the DO loop

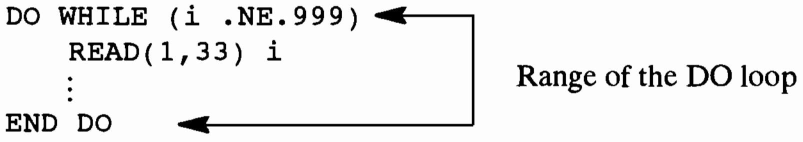
This example shows the range of a block DO loop. It produces the same results as the preceding example.



```
a = 6
DO i = 1,10
   b = SQRT(a)
   WRITE (6,200) b
   a = a+1
END DO
```

Range of the DO loop

This example shows the range of a DO WHILE loop:



```
DO WHILE (i .NE.999)
   READ(1,33) i
   :
END DO
```

Range of the DO loop

A DO loop can be exited at any time. Normal exit occurs when the DO loop is completed and execution continues with the statement following the termination statement of the loop. A DO loop can be exited abnormally with, for example, a GOTO statement, which transfers control out of the loop.

This example searches a list for a keyword. If the keyword is found, control passes out of the DO loop to the statement labeled 60. If the keyword is not found, the loop terminates normally and then the program executes the STOP statement.

```
   DO 50 i = 1,n
50 IF (list(i) .EQ. keyword) GOTO 60
   STOP 'Not found.'
60 PRINT *, 'Match at', i
```

As an extension to the ANSI 77 standard, the range of a DO loop can be extended by passing control out of and then back into the range.

Control can be passed into the range of the loop only if a transfer out of the same loop previously occurred. This is an extension to the ANSI 77 standard, and causes the compiler to generate a warning.

Example

```
    DO 50 i = 1,10,J*2
      GOTO 70
20    x = y*v + r
50  END DO
    GOTO 100

70   v = ban + 6
    GOTO 20
100 CONTINUE
```

Notes

This is a legal transfer out of the range of a DO loop and back into the same range.

After the DO loop in the above example is satisfied, statement 60 is executed, passing control to statement 100. The example shows the awkwardness of transferring into the range of a DO loop without executing the DO statement. The capability is included only for compatibility with other versions of FORTRAN, and should not be used in new programs.

DOUBLE COMPLEX Statement

See "Type Statement" later in this chapter for the syntax of DOUBLE COMPLEX and all other type statements.

DOUBLE PRECISION Statement

See "Type Statement" later in this chapter for the syntax of DOUBLE PRECISION and all other type statements.

ELSE Statement

See "BLOCK IF Statement" under "IF Statement" later in this chapter for information on the ELSE statement.

ELSE IF Statement

See "BLOCK IF Statement" under "IF Statement" later in this chapter for information on the ELSE IF statement.

EMA Statement

The EMA (Extended Memory Area) statement is an extension to the ANSI 77 standard. It declares that local variables or formal arguments are in EMA or VMA (Virtual Memory Area). EMA and VMA may not be supported under some operating systems. For more information, refer to the appropriate programmer's reference manual.

Syntax

EMA *v1,v2,...,vn*

where

v is a simple variable or array that is a formal argument or local variable.

Example / **Notes**

```
PROGRAM main
EMA large_array, x, y
DIMENSION large_array(75000)
```

The EMA statement declares the items large_array, x, and y to be in EMA.

```
CALL sub1 (x,y)
     ⋮
END
```

sub1 is called and is passed the two EMA variables x and y.

```
SUBROUTINE sub1 (a,b)
EMA a,b
     ⋮
END
```

In sub1, the EMA statement contains the two formal arguments a and b corresponding to x and y above.

Arrays and single variables can be put in EMA through the EMA statement. Like other local variables, EMA variables are unique to the program unit that declares them.

Because variables in EMA are accessed by a different mechanism from those not in EMA, you must specify which formal arguments are EMA arguments. The default type for formal arguments is non-EMA. Under the E compiler option, all formal arguments are EMA. (See Table 7-1 for a discussion of the E option.)

Character variables cannot be in EMA.

DATA statements can be used to assign initial values to variables placed in the EMA area. However, this is only supported on RTE-A. If any EMA variables appear in DATA statements in a program, the RTE-A linker will change the program from an EMA program to a VMA program without warning. See the *RTE-A Programmer's Reference Manual*, part number 92077-90007, for more information on EMA and VMA programming. Initialized VMA is not supported on the RTE-6/VM Operating System.

See "$EMA Directive" in Chapter 7 for a complete example of the use of EMA.

Notes on Using EMA

Character variables and arrays cannot be put in EMA, and common blocks in EMA cannot contain character items. Variables in EMA cannot be equivalenced to character variables either directly or indirectly.

Although any variable can be declared to be in EMA, you should restrict EMA usage to those arrays that require a large amount of storage. Because references to EMA variables take longer than references to local variables, restricting EMA usage decreases execution time.

The addressing modes of actual and formal arguments must match (that is, they must be both EMA or non-EMA). If they do not match, an incorrect address is used. The effect is similar to accessing an array with a subscript of unknown value. Therefore, do not pass a non-EMA variable to a subroutine expecting an EMA argument, or vice versa.

END Statement

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

Syntax

```
   END
```

Example

```
PROGRAM xtest
READ (5,*) a,b
IF (a .LT. b) a = b
PRINT (6,*) a, b
END
```

Notes

The END statement terminates program `xtest`.

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, execution of the program terminates.

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

END DO Statement

See “DO Statement” earlier in this chapter for information on the END DO statement.

ENDFILE Statement

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

Syntax

```
         {unit}
ENDFILE {([unit=]unit[,IOSTAT=ios][,ERR=label])}
```

where

unit is the unit number of a sequential file.

ios is an integer variable or integer array element name for error code return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement. It is in the same program unit as the ENDFILE statement. If an error occurs during execution of the ENDFILE statement, control transfers to the specified statement rather than aborting the program.

If the prefix `UNIT=` is omitted, *unit* must be the first parameter. Otherwise the order of parameters is flexible.

An end-of-file record can occur only as the last record of a disk file. After execution of an ENDFILE statement, the file is positioned beyond the end-of-file record.

Some devices (magnetic tape units, for example) can have multiple end-of-file records, with or without intervening data records.

An end-of-file record cannot be written to a direct access file.

Examples	**Notes**
`ENDFILE 10`	An end-of-file record is written to the file connected to unit 10.
`ENDFILE (UNIT=5,IOSTAT=j,ERR=100)`	An end-of-file record is written to the file connected to unit 5. If an error occurs, control transfers to statement 100, and the error code is stored in the variable `j`.

END IF Statement

See "IF Statement" later in this chapter for information on the END IF statement.

ENTRY Statement

The ENTRY statement provides an alternate name, argument list, and starting point for a function or subroutine.

Syntax

```
ENTRY name [([arg1,arg2,arg3,...])]
```

where:

name is the name of an external function and subroutine.

arg is a formal argument. *arg* can be a variable name, an array name, a formal procedure name, or an asterisk. An asterisk is permitted only in a subroutine.

Example

Notes

```
SUBROUTINE linka (d,i,f)
     ⋮
ENTRY search (table,f)
```

ENTRY defines an alternate entry point into subroutine `linka`.

```
CHARACTER*10 FUNCTION compose (word,sent,para)

     ⋮
CHARACTER*15 punctuation
ENTRY punctuation (document)

RETURN
END
```

The ENTRY statement provides an alternate way of entering the function `compose`. Because this function is of type character, the ENTRY statement must also specify a character name.

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 statement. However, for each call to the subprogram through a given entry point, only the formal arguments of that entry point can be used.

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 name in an ENTRY statement can appear in a type statement in a function subprogram.

Character and noncharacter ENTRY statements cannot be mixed in a function subprogram. If an entry name in a function subprogram is of type character, each entry name and the name of the function subprogram must be of type character, and vice versa. In a function subprogram the ENTRY statement name cannot appear as a variable in any statement before the ENTRY statement, except a type 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 considered a nonexecutable statement. If control falls into an ENTRY statement, the statement is treated as an unlabeled CONTINUE statement; that is, control falls through to the next statement.

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, SUBROUTINE, or another ENTRY statement. An entry name must not appear in an EXTERNAL statement.

Here is an example that creates a stack. It shows the use of ENTRY to group the definition of a data structure together with the code that accesses it, a technique known as *encapsulation*.

```
      SUBROUTINE push(value)
      PARAMETER (size = 100)
      IMPLICIT INTEGER (a-z)
      DIMENSION stack(size)
      DATA top /0/

C     Push entry

      IF (top .EQ. size) STOP 'Stack overflow'
      top = top + 1
      stack(top) = value
      RETURN

C     Pop entry

      ENTRY pop(value)
      IF (top .EQ. 0) STOP 'Stack underflow'
      value = stack(top)
      top = top - 1
      RETURN
      END
```

Here are examples of CALL statements that might be associated with the preceding example:

```
DO i=1,LEN(string)
   CALL push (ICHAR (string (i:i)))
END DO


DO i=1,LEN(string)
   CALL pop (j)
   string (i:i) = CHAR(j)
END DO
```

Both examples reverse the characters in `string`.

EQUIVALENCE Statement

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

Syntax

```
EQUIVALENCE (list1)[,(list2), . . . ]
```

where:

list is two or more simple variables, array elements, array names, or character substrings. All items in each list entry 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.

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

The EQUIVALENCE statement can be used to conserve storage. For example, arrays that are manipulated at different times in the same program can be equivalenced. Thus, the same storage space is used for each array.

The types of equivalenced data items can differ. The EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If an array and a simple variable are equivalenced, the array does not have the characteristics of a simple variable and the simple variable does not have the characteristics of an array. The array and the simple variable only share the same storage space.

Use caution when equivalencing data types of different sizes, because the EQUIVALENCE statement specifies that each data item in a list has the same first storage unit. For example, if an integer and a real value are equivalenced, the integer value shares the same space as the most significant word of the 2-word real value.

Equivalence of Array Elements

Array elements can be equivalenced to elements of a different array or to simple variables. For example:

```
DIMENSION a(3), c(5)
EQUIVALENCE (a(2), c(4))
```

specifies that array element a(2) shares the same storage space as array element c(4). This implies that:

- a(1) shares storage space with c(3), and a(3) shares storage space with c(5).
- No equivalence occurs outside the bounds of any of the arrays.

The storage space for the above two arrays is shown in the following table:

Array a	Storage Space Word Number	Array c
	1	c(1)
	2	c(2)
a(1)	3	c(3)
a(2)	4	c(4)
a(3)	5	c(5)

Array elements are equivalenced on the basis of storage elements. If the arrays are not of the same type, they may not line up element by element. For example, the statements:

```
DIMENSION a(2), ibar(4)
EQUIVALENCE (a(1), ibar(1))
```

where a is of type REAL*4 and ibar is of type INTEGER*2, produce the following storage space allocation:

Array a	Storage Space Word Number	Array ibar
a(1)	1	ibar(1)
	2	ibar(2)
a(2)	3	ibar(3)
	4	ibar(4)

If only an array name appears in an EQUIVALENCE statement, it has the same effect as using an array element name that specifies the first element of the array. Specifying EQUIVALENCE (a,ibar) instead of EQUIVALENCE (a(1),ibar(1)) in the above example would produce the same results.

When equivalencing array elements with other array elements or simple variables, do not specify that the same storage space be occupied by more than one element of the same array. The following example is illegal because it specifies the same storage space for a(1) and a(2):

```
DIMENSION a(2)
EQUIVALENCE (a(1),b),(a(2),b)
```

An EQUIVALENCE statement must not specify that consecutive array elements are to be noncontiguous. For example:

```
REAL a(2), r(3)
EQUIVALENCE (a(1), r(1)), (a(2), r(3))
```

is prohibited because the EQUIVALENCE statement specifies that r is noncontiguous.

Equivalence Between Arrays of Different Dimensions

To determine equivalence between arrays of different dimensions, FORTRAN contains an internal array successor function that views all elements of an array in linear sequence. Each array is stored as if it were a one-dimensional array. Array elements are stored in ascending sequential order. The first index varies the fastest, then the second, then the third, and so on.

Examples	**Notes**
`i(-2:4)`	The elements of `i` are stored in the following order: `i(-2)  i(-1)  i(0)  i(1)  i(2)  i(3)  i(4)`
`t(2,3)`	The elements of `t` are stored in the following order: `t(1,1)  t(2,1)  t(1,2)  t(2,2)  t(1,3)  t(2,3)`

`k(2,2,3)`	The elements of `k` are stored in the following order (as read left to right, top to bottom):
	`k(1,1,1)  k(2,1,1)  k(1,2,1)  k(2,2,1)`
	`k(1,1,2)  k(2,1,2)  k(1,2,2)  k(2,2,2)`
	`k(1,1,3)  k(2,1,3)  k(1,2,3)  k(2,2,3)`

The number of words each element occupies depends on the type of the array. For example, these statements:

```
DIMENSION a(2,2), i(4)
EQUIVALENCE (a(2,1), i(2))
```

produce the following storage space allocation:

Array a	**Storage Space** **Word Number**	**Array i**
`a(1,1)`	1	
	2	`i(1)`
`a(2,1)`	3	`i(2)`
	4	`i(3)`
`a(1,2)`	5	`i(4)`
	6	
`a(2,2)`	7	
	8	

In the EQUIVALENCE statement, a multidimensional array can be referenced by a single dimension that specifies the array element relative to its linear position in storage. This is an extension to the ANSI 77 standard. These statements:

```
DIMENSION a(2,2), i(4)
EQUIVALENCE (a(3), i(4))
```

produce the same storage space allocation as the previous example.

Equivalence of Character Variables

As an extension to the ANSI 77 standard, character and noncharacter data items can be equivalenced.

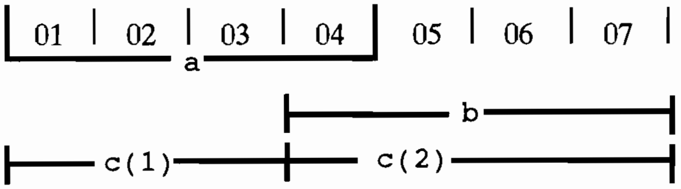
If equivalencing of character and noncharacter data forces a noncharacter data item to begin on an odd byte address, a compilation error occurs.

Equivalenced data items do not have to be the same length. An EQUIVALENCE statement specifies that the storage sequences of the character data items whose names are specified in the list have the same first character storage unit. This causes the association of the data items in the list and can cause association of other data items. Any adjacent characters in the associated data items can also have the same character storage unit and thus can also be associated.

In the example:

```
CHARACTER a*4, b*4, c(2)*3
EQUIVELENCE (a,c(1)), (b,c(2))
```

the association of a, b, and c can be illustrated this way:



Equivalence in Common Blocks

Data elements can be put into a common block by specifying them as equivalent to data elements mentioned in a COMMON statement. If one element of an array is equivalenced to a data element within a common block, the whole array is placed in the common block and equivalence is maintained for storage units preceding and following the data element in common. The common block is always extended when it is necessary to fit an equivalenced array into the common block. No array can be equivalenced into a common block, however, if storage elements would have to be prefixed to the common block to contain the entire array. Equivalences cannot insert storage into the middle of the common block or rearrange storage within the block. Because the elements in a common block are stored contiguously according to the order in which they are mentioned in the COMMON statement, two elements in common cannot be equivalenced. In the following example, array i is in a common block and array element j(2) is equivalent to i(3):

```
DIMENSION i(6), j(6)
COMMON i EQUIVALENCE (i(3),j(2))
```

The common block is extended to accommodate array `j` as follows:

Array i	**Common Block** **Word Number**	**Array j**
`i(1)`	1	`j(0)*`
`i(2)`	2	`j(1)`
`i(3)`	3	`j(2)`
`i(4)`	4	`j(3)`
`i(5)`	5	`j(4)`
`i(6)`	6	`j(5)`
`i(7)*`	7	`j(6)`

* Because FORTRAN does not check bounds on dimensional arrays, `j(0)` coincides with `i(1)`, and `i(7)` references the same storage space as `j(6)`.

The equivalence set up by the following example is not allowed:

```
DIMENSION i(6), j(6)
COMMON i
EQUIVALENCE (i(1), j(2))
```

To set array `j` into the common block, an extra word must be inserted in front of the common block. Element `j(1)` would be stored in front of the common block; thus, `EQUIVALENCE (i(1), j(2))` is not allowed.

EXTERNAL Statement

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.

Syntax

```
EXTERNAL proc1 [,proc2, ... ]
```

where:

proc is the name of a subprogram. Each name can appear only once.

Examples

```
EXTERNAL b1
CALL sub(a,b1,c)
      ⋮
END
```

Notes

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

```
SUBROUTINE sub(x,y,z)
z = y(z)
RETURN
END
```

The reference to `y` causes `b1` to be called.

The EXTERNAL statement allows you to use 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.

As an extension to the ANSI 77 standard, statement function names can appear in an EXTERNAL statement.

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 the program.

Refer to Appendix E for information on compatibility of the EXTERNAL statement with the ANSI 66 standard.

Also see the section “INTRINSIC Statement.”

FORMAT Statement

The FORMAT statement describes the position and type of data fields in an ASCII data record.

Syntax

label `FORMAT(`*des1*`[,`*des2*`, . . . ])`

where:

label is a statement label.

des is a format or edit descriptor.

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

The allowed format descriptors are summarized in Table 3-5. The edit descriptors are summarized in Table 3-6. See Chapter 4 for a more detailed explanation of the FORMAT statement.

Table 3-5. Format Descriptors

Descriptor	Data Type
A*w*	Character or Hollerith
R*w*	Character or Hollerith
D*w*.*d*[Ee]	Real, Double Precision, Complex
E*w*.*d*[Ee]	Real, Double Precision, Complex
F*w*.*d*	Real, Double Precision, Complex
G*w*.*d*[Ee]	Real, Double Precision, Complex
I*w*[.*m*]	Integer
K*w*	Octal
@*w*	Octal
O*w*	Octal
L*w*	Logical

where: *w* is an integer specifying the field width.
d is an integer specifying the number of digits to the right of the decimal point.
e is an integer specifying the number of digits in the exponent.
m is an optional integer specifying the minimum number of digits on output.

Table 3-6. Edit Descriptors

Descriptor	Function
BN	Ignore blanks
BZ	Treat blanks as zeros
*n*H	Hollerith editing
T*c*	Skip to column *c*
*n*X	Skip *n* positions
TR*c*	Skip *c* positions to the right
TL*c*	Skip *c* positions to the left
/	Begin new record
:	Terminate format if list empty
'...'	Literal editing
"..."	Literal editing
S	Processor determines sign output; same as SS on HP 1000
SP	Output optional plus signs
SS	Inhibit optional plus sign output

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

```
  WRITE (1,'("Average is ",I5)') iaverage
```

However, single quotation marks are preferred.

FUNCTION Statement

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

Syntax

```
[type] FUNCTION name ([arg1, arg2, . . . ])[,comment]
```

where:

type is the type of the function; *type* can be one of the following:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
DOUBLE COMPLEX
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER[**len*]

The **n* in the above type specifications is optional.

In CHARACTER [**len*], *len* is the length of the formal argument (*arg*) of the character function. The default value is 1. For an array, the length specified is for each element of the array. *len* can be one of the following:

- An unsigned integer constant.
- An integer constant expression with a positive value. The integer expression must be enclosed in parentheses. Example: (−3+4).
- An asterisk enclosed in parentheses (*).

name is the name of the function (if *type* is not specified, the name is typed the same way as a variable).

arg is a formal argument of the function.

comment is up to 86 characters passed to the linker. This feature is an extension to the ANSI 77 standard.

Examples	**Notes**
`FUNCTION comp()`	Defines the function `comp`.
`INTEGER FUNCTION timex(a,b,k)`	Defines an integer function, `timex,` with three arguments.
`CHARACTER*6 FUNCTION namex(l)`	Defines a character function, `namex`, with one argument; the function result is six characters long.

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

- Variables
- Array names
- Subprogram 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.

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*(*), the function assumes the length declared for it by the calling program. Each calling program may declare a different length.

Function subprograms are discussed in detail in "Function Subprograms" in Chapter 6.

GOTO Statement

The GOTO statement transfers control to a labeled statement in the same program unit. There are three kinds of GOTO statements:

- Unconditional GOTO
- Computed GOTO
- Assigned GOTO

The statement can be written GOTO or GO TO, because blanks are significant only in character and Hollerith constants.

Unconditional GOTO Statement

The unconditional GOTO statement transfers control to the specified statement.

Syntax

`GOTO` *label*

where:

label is the label of an executable statement.

Example	**Notes**
`GOTO 20`	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.

Computed GOTO Statement

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

Syntax

`GOTO (`*label1*`,` *label2*`, . . . )[,]`*exp*

where:

label is the label of an executable statement. The same label can appear more than once.

exp is an integer, real, or double precision expression.

The use of noninteger expressions is an extension to the ANSI 77 standard.

In a CDS program, a computed GOTO statement cannot have more than 255 labels.

The computed GOTO statement passes control to one of several labeled statements, depending on the result of an evaluation. *exp* is evaluated and truncated to an integer value (the index). The index is used to select 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 of the list of labels. If the index value is 2, the second label in the list is used, and so on. If *exp* evaluates to less than 1, or to a value greater than the number of labels in the label list, control passes to the statement following the computed GOTO.

Examples **Notes**

```
a = 3
GOTO (30,60,50,100) a
```

Because a has a value of 3, control passes to statement 50.

```
b = 1.5
z = 1
GOTO (10,20,40,40) b + z
```

Because `INT(b + z) = 2`, control passes to statement 20.

Assigned GOTO Statement

The assigned GOTO statement transfers control to the statement whose label was most recently assigned to the variable in the GOTO statement by an ASSIGN statement.

Syntax

GOTO *ivar*[[,](*label1*, *label2*, . . .)]

where

ivar is a simple integer variable.

label is the label of an executable statement. The list of labels is optional.

Examples **Notes**

```
ASSIGN 10 TO age
GOTO age
```

Control transfers to statement 10 when the GOTO statement is executed.

```
ASSIGN 100 TO time
GOTO time (90,100,150)
```

Control transfers to statement 100 when the GOTO statement is executed.

ivar must be given a label value 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 *i*var.

The list of labels gives the possible label values that *ivar* might assume. Although the list of labels is not used, it should be provided as good programming practice. Programs with undocumented assigned GOTO statements are very difficult to understand.

IF Statement

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

- Arithmetic IF
- Logical IF
- Block IF

An arithmetic IF statement transfers control to one of three labeled statements depending on whether an expression evaluates to a negative, zero, or positive value. A logical IF statement causes a statement to be executed if an evaluated expression is true. A block IF causes one of two blocks of statements to be executed depending on the value of one or more logical expressions.

Each kind of IF statement is described in detail in the following sections.

Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three statements. The form of the arithmetic IF statement is:

Syntax

`IF` (*exp*) *labeln,labelz,labelp*

where

exp is an arithmetic expression of any type except complex or double complex.

label is the statement label of an executable statement.

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

When an arithmetic statement is executed, *exp* is evaluated. If the resulting value is negative, control passes to the statement whose label is *labeln*. If the value is 0, control passes to the statement whose label is *labelz*. If the value is positive, control passes to the statement whose label is *labelp*.

If the value of the expression exceeds the range of the expression, an overflow condition occurs. Because overflow conditions are not detected by the compiler, an arithmetic IF statement can make an erroneous transfer. Overflow conditions can often be avoided with logical IF statements.

Examples	**Notes**

`testa = 0.`
`IF (test) 50,100,50`
	Because `testa` equals 0, 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.

`IF (timex) 60,60,60`	Control passes to statement 60 regardless of the value of `timex`.

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

As illustrated in the above examples, two of the labels in the label list can be the same; control branches to one of two possible statements. In fact, all of the labels in the list can be the same. If all are the same, control branches to the statement bearing this label, regardless of the results of the evaluation.

If two of the labels are the same and one of the three labels is the label of the next statement, the statement should be changed to a logical IF or a block IF for improved readability. For example, this arithmetic IF statement:

```
    IF (exp) 30,40,30
40  ...
```

is the same as this logical IF statement:

```
    IF (exp  .NE. 0) GOTO 30
```

and this arithmetic IF statement:

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

is the same as this block IF statement:

```
    IF (exp) .LE. 0) THEN
    ...
    END IF
```

Logical IF Statement

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

Syntax

`IF` (*exp*) *statement*

where

exp is a logical expression.

statement is any executable statement other than a DO, END, or block IF statement, or another logical IF statement.

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.

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

Block IF Statement

The block IF statement is an extension of the logical IF statement, allowing one of two blocks of statements to be executed depending on the true or false value of a logical expression.

Syntax

```
IF (exp) THEN
          ⋮
     statement 1
     statement 2
          ⋮
[ELSE [IF(exp) THEN]
          ⋮
     statement 3
     statement 4
          ⋮
[ELSE
          ⋮
     statement 5
     statement 6
          ⋮
     statement n]]
          ⋮
ENDIF
```

where:

exp is a logical expression.

statement is any executable statement or a format statement.

`END IF` terminates the block IF. One END IF is required for each IF...THEN statement (see the third example below).

One block IF statement can contain any number of ELSE IF sub-blocks, but only one ELSE sub-block. IF blocks can be nested to any level desired.

Examples

Notes

```
x = y
IF (x .EQ. y) THEN
      x = x+1
END IF
```

Because `x = y`, the value of `x` is replaced by the value of `x+1`. Note that this is equivalent to the 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
END IF
```

If `x` < 0, one block of code is executed; if `x` ≥ 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
     END IF
ELSE
     n = i
     k = n(i)
END IF
```

This example demonstrates nesting of IF blocks using the construct:

```
IF (exp1) THEN
     ⋮
IF (exp2) THEN
     ⋮
     ELSE IF (exp3) THEN
     ⋮
     END IF
ELSE
  ⋮
END IF
```

If the logical expression is true, the block of statements between THEN and ELSE is executed (if there is no ELSE statement, the block of statements between THEN and END IF is executed). If the logical expression is false, the block of statements between ELSE and END IF is executed (if no ELSE block exists, control passes to the statement following the END IF statement).

Using ELSE IF does not change the nesting level of the IF block. (The term *nesting level* refers to the number of enclosing IF blocks; the nesting level is equal to the number of preceding IF ...THEN statements minus the number of preceding END IF statements.) The nesting level must be equal to 0 at the end of each program unit. An IF THEN statement increases the nesting level by one, while the END IF statement decreases the nesting level by one.

Example **Nesting Level**

```
                                         0
IF (exp1) THEN                           1
    ⋮                                    ⋮
   IF (exp2) THEN                        2
    ⋮                                    ⋮
   ELSE  IF (exp3) THEN                  2
   ELSE                                  2
    ⋮                                    ⋮
   ENDIF                                 1
    ⋮                                    ⋮
ELSE  IF (exp4) THEN                     1
    ⋮                                    ⋮
ENDIF                                    0
```

The FORTRAN 77 compiler can perform conditional compilation within block IF statements. If the expression following the IF in an IF block contains only constants and named constants (specified in a PARAMETER statement) and does not contain any character data, then the compiler can determine which section of code need not be generated: either the code following the THEN in the IF block or the code following the corresponding ELSE or ELSE IF (if specified). This form of conditional compilation can be used only in the executable portion of a program. It cannot be used in declarations.

Example **Code Generated**

```
INTEGER system, rte4b, rte110, rte120          Yes
PARAMETER (rte4b=4, rte110=110, rte120=120)    Yes
PARAMETER (system=rte4b)                              Yes
    :                                                 Yes
IF (system .EQ.  rte4b) THEN                          Yes
    CALL EXEC (23,5HD.RTR,...)                        Yes
ELSE IF (system .EQ. rte110) THEN                     No
    :                                                 No
ELSE IF (system .EQ. rte120) THEN                     No
    :                                                 No
ELSE                                                  No
    PRINT '("UNKNOWN SYSTEM")'                        No
ENDIF                                                 Yes
```

IMPLICIT Statement

The IMPLICIT statement overrides or confirms the type associated with the first letter of a variable name.

Syntax

```
            {NONE}
IMPLICIT    {type(range1,[range2, ...])[,type(range1,[range2, ...])]}
```

where

type is the type to be associated with the corresponding letter or list of letters in *range*; *type* can be one of the following:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
DOUBLE COMPLEX
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER[**len*]

In CHARACTER [**len*], *len* is the length of the character items, and is one of the following:

- An unsigned integer constant.
- An integer constant expression enclosed in parentheses with a positive value.

If *len* is not specified, the default value is 1.

range is either a single letter or a range of letters (for example, a-z or i-n) to be associated with the specified type. Writing a range of letters has the same effect as writing a list of single letters.

Example	**Notes**
`IMPLICIT COMPLEX(i,j,k),INTEGER(a-c)`	All variables and function names beginning with i, j, or k are of type COMPLEX; data items beginning with a, b, or c are of type INTEGER.

An IMPLICIT statement specifies a 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; it does not change the type of any intrinsic functions.

The IMPLICIT statement itself can be overridden for specific names when these names are used in a type statement. For example, `IMPLICIT INTEGER (a)` specifies that symbolic names starting with the letter `a` are type integer. A type statement such as `REAL able`, however, indicates that the variable `able` is type real, thus overriding the IMPLICIT statement.

Uppercase and lowercase letters are equivalent in arguments to the IMPLICIT statement. Thus `IMPLICIT INTEGER (Q)` and `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.

As a MIL-STD-1753 extension to the ANSI 77 standard, if IMPLICIT NONE is specified, inherent typing is disabled and all simple variables, arrays, named constants, function subprograms, entry names, and statement functions (but not intrinsic functions) must be explicitly typed. The IMPLICIT NONE statement must be the only IMPLICIT statement in the program unit. The types of intrinsic functions are not affected.

Within the specification statements of a program unit, IMPLICIT statements must precede all other specification statements, except PARAMETER statements.

The same letter must not appear as a single letter, or be included in a range of letters, more than once in all of the IMPLICIT statements in a program unit.

INCLUDE Statement

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 logical unit (LU). When EOF is read from this file or LU, the compiler continues processing at the line following the INCLUDE statement.

Syntax

```
                    {[,LIST]}
   INCLUDE file_name {[,NOLIST]}
```

where:

file_name is either the disk file name or an LU number. *file_name* may be quoted or unquoted. Unquoted file names must not contain '=' or '('.

`LIST` causes the current lines to be listed. `LIST` is the default.

`NOLIST` causes lines not to be listed.

If the $LIST directive is OFF, lines are not listed regardless of `LIST` and `NOLIST`.

INCLUDE statements cannot be continued. INCLUDEs can be nested nonrecursively; that is, an INCLUDE cannot mention an active include file.

If the file name does not specify a directory, the directory containing the source file is searched first, then the working directory.

Examples

```
INCLUDE specs

INCLUDE '../includes/grahics.inc',NOLIST
```

After an INCLUDE statement is executed, the LIST status is restored to what it was just before the INCLUDE was encountered, even if LIST or NOLIST is specified and even if the LIST directive appears in the included file.

Line numbering within the listing of an included file begins with 1. These line numbers are suffixed with "+" (for example, `153+`). The original line numbering of the source resumes after the included file listing.

If an interactive LU is specified for the file name, information is included interactively from that LU, in which case the prompt listed at the LU is "+".

INCLUDE is also used as a compiler directive. The $INCLUDE compiler directive is useful when the file name contains special characters that might be misinterpreted by the compiler. (See "$INCLUDE Directive" in Chapter 7.)

INQUIRE Statement

The INQUIRE statement provides information about selected properties of a particular file or unit number.

Syntax

INQUIRE ([[UNIT=]*unit*][,FILE=*name*][,IOSTAT=*ios*][,ERR=*label*]...*other specs*)

where:

unit is the unit number of a sequential file.

name is a character expression containing the file name of the file to be inquired about. See Chapter 5 for more information about file names.

ios is an integer variable or integer array element name for error code return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement in the same program unit as the INQUIRE statement. If an error occurs during execution of the INQUIRE statement, control transfers to the specified statement rather than aborting the program.

Either the `UNIT` or `FILE` keyword specifier must be present in the keyword list, but not both. If the prefix `UNIT=` is omitted and *unit* is present, *unit* must be the first parameter in the list. Otherwise the order of parameters is flexible.

Table 3-7 below describes each of the specifiers used with the INQUIRE statement.

Most of the information described in Table 3-7 is assigned through the OPEN statement, which is described in "OPEN Statement" below.

See "INQUIRE Statement" in Chapter 5 for an example of the INQUIRE statement.

In Table 3-7, when a variable is specified as undefined, its value may or may not be changed from its previous value by the INQUIRE statement. In any event, the value is meaningless if the file or unit either does not exist or cannot be accessed at the time the INQUIRE statement is executed.

Table 3-7. INQUIRE Statement Specifiers

Specifier	Restrictions	Description
`FILE=`*name*	Character expression	Specifies file name for inquiry by file *name.* **Example:** `FILE='OUTPUT'`
`UNIT=`*unit*	Integer expression	Specifies unit number for inquiry by *unit.* **Example:** `UNIT=i`
`IOSTAT=`*ios*	Integer variable or array element	*ios*=0 if no error; *ios*=positive value if error condition exists. **Example:** `IOSTAT=j`
`ERR=`*label*	Statement number	Control transfers to specified executable statement if error condition on named file or unit exists. **Example:** `ERR=99`
`EXIST=`*ex*	Logical variable or array element	*ex*=.TRUE. if named file exists and is accessible; *ex*=.FALSE. otherwise. **Example:** `EXIST=lext`
`OPENED=`*od*	Logical variable or array element	*od*=.TRUE. if named file or unit has been opened by an OPEN statement in this program; *od*=.FALSE. otherwise. **Example:** `OPENED=lopn`
`NUMBER=`*num*	Integer variable or array element	*num* is the FORTRAN logical unit number of the external named file; if no unit is connected to the named file, *num* is undefined. **Example:** `NUMBER=n`
`NAMED=`*nmd*	Logical variable or array element	*nmd*=.TRUE. if specified unit is not a scratch file; *nmd*=.FALSE. otherwise. **Example:** `NAMED=lnam`
`NAME=`*fn*	Character variable, array element, or substring	*fn* is returned as the external name of the specified unit; if the file has no name or is not connected, *fn* is undefined. **Example:** `NAME=nam`
`USE=`*use*	Character variable, array element, or sub-string	*use* returns EXCLUSIVE if not connected for shared use, NONEXCLUSIVE if connected for shared use, and not defined if the file is not connected. **Example:** `USE=use`
`ACCESS=`*acc*	Character variable, array element, or substring	*acc* returns SEQUENTIAL if the unit or file is connected for sequential access, DIRECT if the unit or file is connected for direct access, and is undefined if the unit or file is not connected. **Example:** `ACCESS=acc`
`SEQUENTIAL=`*seq*	Character variable, array element or substring	*seq* returns YES if connected for sequential access, NO if connected for direct access, and undefined if the processor is unable to determine the access type. **Example:** `SEQUENTIAL=seq`

Table 3-7. INQUIRE Statement Specifiers (continued)

Specifier	Restrictions	Description
`DIRECT=`*dir*	Character variable, array element, or substring	*dir* returns YES if connected for direct access, NO if connected for sequential access, and undefined if the processor is not able to determine the access type. **Example:** `DIRECT=dir.`
`FORM=`*fm*	Character variable, array element, or substring	*fm* returns FORMATTED or UNFORMATTED depending upon the format specified when the unit was connected; *fm* is undefined if the unit is not connected. **Example:** `FORM=for`
`FORMATTED=`*fmt*	Character variable, array element, or substring	*fmt* returns FORMATTED or UNFORMATTED depending upon the format specified when the unit was connected; *fmt* is undefined if the unit is not connected. **Example:** `FORM=for`
`UNFORMATTED=`*unf*	Character variable, array element, or substring	*unf* returns YES or NO depending upon whether the unit was connected for unformatted data transfer; *unf* is undefined if the processor is unable to determine the form of data transfer. **Example:** `UNFORMATTED=iunf`
`RECL=`*rcl*	Integer variable or array element	*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. **Example:** `RECL=irec`
`NEXTREC=`*nr*	Integer variable or array element	*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, or is an LU, or has an indeterminate status, *nr* is undefined. If the file can contain more than 32767 records, *nr* should be double integer. **Example:** `NEXTREC=nrec`
`BLANK=`*blnk*	Character variable, array element, or substring	*blnk* contains 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. **Example:** `BLANK=iblk`

Table 3-7. INQUIRE Statement Specifiers (continued)

Specifier	Restrictions	Description
MAXREC=*mrec*	Integer variable or array element	*mrec* returns the largest record number that currently exists in a direct access file. If the file is extendable or already has extents, the returned value may indicate the amount of disk space allocated for the file. If used in an inquiry of a sequential file, *mrec* is the number of 128-word blocks in the file (not including extents). If the file can contain more than 32767 records, *mrec* should be double integer. MAXREC is an extension to the ANSI 77 standard. **Example:** MAXREC=mr
NODE=*node*	Integer variable or array element	*node* is set to the node number to which the unit is connected. (If at the local node, *node*=−1.) When the CI file system is used, the node number is also returned in the file name. NODE is an extension to the ANSI 77 standard. **Example:** NODE=5003

INTEGER Statement

See "Type Statement" later in this chapter for the syntax of INTEGER and all other type statements.

INTEGER*2 Statement

See "Type Statement" later in this chapter for the syntax of INTEGER*2 and all other type statements.

INTEGER*4 Statement

See "Type Statement" later in this chapter for the syntax of INTEGER*4 and all other type statements.

INTRINSIC Statement

The INTRINSIC statement identifies a name as representing an intrinsic function and permits the name to be used as an actual argument.

Syntax

`INTRINSIC` *fun1*`[`*,fun2*`,...]`

where:

fun is the name of an intrinsic function. Each name can appear only once.

Example	**Notes**

`INTRINSIC SIN,TAN`
`CALL MATH(SIN,TAN)`
	The INTRINSIC statement informs the compiler that `SIN` and `TAN` are intrinsics.

The INTRINSIC statement provides a means of using intrinsics as actual arguments. The INTRINSIC statement informs the compiler that these names are intrinsic names and not variable names. Whenever an intrinsic name is passed as an actual argument, it must be placed in an INTRINSIC statement in the calling program.

The names of intrinsic functions for type conversion (INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR), for logical relationships (LGE, LGT, LLE, LLT), and for choosing the largest or smallest value (MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, AMIN0, MIN1) must not be used as actual arguments.

Generic names must not be used in an INTRINSIC statement.

A name must not appear in both an EXTERNAL and an INTRINSIC statement in the same program unit.

LOGICAL Statement

See "Type Statement" later in this chapter for the syntax of LOGICAL and all other type statements.

LOGICAL*2 Statement

See "Type Statement" later in this chapter for the syntax of LOGICAL*2 and all other type statements.

LOGICAL*4 Statement

See "Type Statement" later in this chapter for the syntax of LOGICAL*4 and all other type statements.

OPEN Statement

The OPEN statement establishes a connection between a unit number and a file; it also establishes or verifies properties of a file.

Syntax

OPEN([UNIT=]*unit*[,FILE=*name*][,IOSTAT=*ios*][,ERR=*label*]...*other specs*)

where

unit is the unit number for the file.

name is a character expression containing the file name of the file to be connected. See Chapter 5 for more information about file names.

name can also hold the ASCII representation of a system LU (logical unit).

ios is an integer variable or integer array element name for error return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement in the same program unit as the OPEN statement. If an error occurs during the execution of the OPEN statement, control transfers to the specified statement rather than aborting the program.

The UNIT specifier is required in the keyword list. If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. The other specifers can be in any order. At most one each of the other specifics can appear in the keyword list.

Table 3-8 describes each of the specifiers used with the OPEN statement.

See Chapter 5 for additional details and examples.

Table 3-8. OPEN Statement Specifiers

Specifier	Restrictions	Description
FILE=*name*	Character expression	Specifies file name, which can be the ASCII representation of an LU identifying a nondisk device (value less than 64).
UNIT=*unit*	Integer expression	Specifies unit number.
IOSTAT=*ios*	Integer variable or array element	*ios*=0 if no error; *ios*=positive value if error condition exists.
ERR=*label*	Statement number	Control is transferred to specified executable statement if error encountered during OPEN.
USE=*use*	Character expression	Specifies file or unit for EXCLUSIVE (default) or NONEXCLUSIVE use. As an extension to the ANSI 77 standard, USE can also specify UPDATE mode. (See Note 3.)
STATUS=*sta*	Character expression	Specifies file as OLD, NEW, SCRATCH, or UNKNOWN (default). (See Note 1.)
ACCESS=*acc*	Character expression	Specifies file access to be DIRECT, SEQUENTIAL (default), BLOCKS (which forces to type 1, an extension to the ANSI 77 standard) or APPEND (which is like SEQUENTIAL, but file is positioned at the end). (See Note 2.)
FORM=*fm*	Character expression	Specifies data format to be FORMATTED or UNFORMATTED. Included for compatibility only, has no effect on the file.
RECL=*rcl*	Integer expression	Specifies record length for direct access. Length is measured in characters (bytes). If *rcl*>120, you must use LGBUF to allocate a larger record buffer. Note that the LGBUF size argument is in words, not bytes.
BLANK=*blnk*	Character expression	Specifies treatment of blanks within numbers on input. If BLANK='NULL' (default), blanks are ignored; if BLANK='ZERO', blanks are treated as zeros.
MAXREC=*mrec*	Integer expression	Specifies the size, in records, of a direct access file when no explicit length is given in the file name (ignored otherwise.) The number of records can be greater than 32767, in which case *mrec* must be double integer. MAXREC is an extension to the ANSI 77 standard. **Example:** MAXREC=40000.

Table 3-8. OPEN Statement Specifiers (continued)

Specifier	Restrictions	Description
BUFSIZ=*bufs*	Integer expression	Specifies the number of 128-word blocks of disk buffer for this file. This parameter overrides the default (the third argument in a $FILES directive). You must use the fourth argument in a $FILES directive to allocate sufficient blocks for all files which will be open at any given time. Also see: the $FILES directive; the NFIOB library function. BUFSIZ is ignored for system LU opens (nondisk devices), DS, and redundant OPEN statements. BUFSIZ is an extension to the ANSI 77 standard. The LGBUF library function is completely unrelated to BUFSIZ. It allocates the record buffer shared by all READ and WRITE statements.
NODE=*node*	Integer expression	Specifies the DS node number at which the connection is to be made. In the FMGR file system, this specifier is legal only if DS is included in the $FILES directive. This specifier is ignored if the node number is specified in the file name. NODE is an extension to the ANSI 77 standard.

Notes for Table 3-8:

Note 1

If	=	then the	and
STATUS	'OLD' 'NEW' 'SCRATCH'	FILE= parameter is required FILE= parameter is required FILE= parameter must not be present	the file must exist. the file must not exist. a scratch file is created.
If	**=**	**and if the**	**then the**
STATUS	'UNKNOWN'	FILE= parameter is present and is not a system LU FILE= parameter is not present or if the parameter is a system LU	file named is created if it does not already exist. the nondisk unit is connected to the unit specified.

Note 2

If	=	then the	and the
ACCESS	'SEQUENTIAL' or 'APPEND'	RECL= parameter must not be present and the file must be of type 0, 3, 4, or 5	file is opened for sequential access.
	'DIRECT'	RECL= parameter is required and the file must be of type 1 or 2	file is opened for direct access.
	'BLOCKS'	RECL= parameter is required and must be 256, and the file must be of type 1 or greater.	file is opened for direct access with 256-byte records (forced to type 1).

Note 3

If	=	then
USE	'UPDATE'	this mode allows you to write to sequential files that are randomly positioned using FPOSN. When this mode is used, the file is opened for shared access. There is no provision for exclusive access with update. (See the discussion on update mode for FMP OPEN calls in the appropriate programmer's reference manual.) USE='UPDATE' is not necessary for direct access files.

Examples	Notes
` OPEN (100,FILE='/mlib/file10/mycode.ex')`	The CI file system file is connected to unit 100.

` OPEN (10,FILE='INV::JW',NODE=15,`
`+ACCESS='SEQUENTIAL',ERR=100,IOSTAT=ios)`
	The file `INV` on CRN `JW` on node number 15 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`. If the FMGR file system is used, DS must have been specified in the $FILES directive.

` OPEN (ACCESS='DIRECT',UNIT=4,RECL=50,`
`+FORM='FORMATTED',FILE=next1)`
	The character variable 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.
	Note: The file must be type 2 for this OPEN statement to succeed.

` OPEN (6,FILE='1')`	The system logical unit 1 is connected to FORTRAN unit 6.

Under older file systems, file names have a restricted form and must be named in uppercase letters. See your system reference manual for details.

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=`*name* option 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 the value of the `BLANK=` option. 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 causes an error.

PARAMETER Statement

The PARAMETER statement is used to define named constants. After the definition of a name in a PARAMETER statement, subsequent uses of the name are treated as if the constant was used. When a character variable is defined as a constant in a parameter statement, the variable name cannot be used in a substring expression in subsequent statements within the program.

Syntax

`PARAMETER` (*cname1=cexp*1[,*cname2=cexp2,....*])

where:

cname is a symbolic name that represents a constant. This name cannot appear in any statement before the PARAMETER statement, except a type statement.

cexp is a constant expression.

Examples

```
PARAMETER (minval=-10,maxval=50)

PARAMETER (debug=.TRUE.)

PARAMETER (file='WELCOM')

PARAMETER (clear=char(33B)//'H'//char(33B)//'J')
```

If the symbolic name *cname* is of type integer, real, double precision, complex, or double complex, the corresponding expression *cexp* must be an arithmetic constant expression. If the symbolic name *cname* is of type character, the corresponding expression *cexp* must be a character constant expression. If *cname* is of type logical, *cexp* must be logical or relational constant expression. Character constant expressions can include `CHAR`(*const*) as a value; *const* must be an integer constant or named constant.

Each *cname* is the symbolic name of a constant that becomes defined with the value determined from the expression *cexp. cexp* must appear on the right of the equal sign, in accordance with the rules for assignment statements. If exponentiation (**) is used, the exponent must be of type integer.

Any symbolic name of a constant that appears in an expression must be defined in a previous PARAMETER statement in the same program unit or earlier in the same PARAMETER statement.

A symbolic name of a constant must not be defined more than once in a program unit.

Even though the constant is represented by a name, that name may not be used in the context where a variable is required. For example, it may not appear on the left side of an assignment or in a substring expression.

Examples

```
      PARAMETER (nbytes=4)
      REAL*(nbytes) a,b,c

      PARAMETER (lower=0, upper=7)
      DIMENSION a (lower:upper)
      DO 10 i=lower,upper
      a (i) = 1.0

   10 CONTINUE

      PARAMETER (pi=3.14159)
        .
        .
        .
      area = pi * (radius**2)
```

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. Its type must not be changed by subsequent statements, including IMPLICIT statements. If a symbolic name of type CHARACTER*(*) is defined in a PARAMETER statement, its length becomes the length of the expression assigned to it.

Once such a symbolic name is defined, that name can appear in that program unit in any subsequent statement as a value in an expression or in a DATA statement. A symbolic name of a constant is not recognized in a character or Hollerith constant or within a FORMAT statement.

A symbolic name in a PARAMETER statement can identify only the corresponding constant in that program unit.

PAUSE Statement

The PAUSE statement causes a temporary pause in program execution.

Syntax

```
PAUSE [n]
```

where

n is an unsigned integer constant (one to five digits) or a character constant.

Examples	**Notes**
`PAUSE 7777`	The message "`PAUSE 7777`" is displayed at the terminal.
`PAUSE 'ENTER GO to Continue'`	The message "`Enter GO to Continue`" is displayed at the terminal.
`PAUSE`	Nothing is displayed at the terminal.

The PAUSE statement suspends execution of a program. When the program suspends, what is printed at the terminal depends on whether digits, characters, or nothing has been specified in the PAUSE statement.

If digits are used, the message PAUSE <*digits*> is displayed at the terminal. If a character expression is used, the character value is displayed at the terminal. If nothing is used after PAUSE, nothing is displayed.

After displaying the appropriate message, the PAUSE statement causes a blank line and a suspend message to be displayed. The suspend message has the form:

```
<prg> Suspended
```

where:

<*prg*> is the five-character program name.

At this point the program is suspended. The operator must enter the system command GO to resume operation.

PRINT Statement

The PRINT statement transfers data from memory to the standard output unit.

Syntax

`PRINT` *fmt*[,*outlist*]

where

fmt is 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 specifying the text of a FORMAT statement
- An asterisk

outlist is a list that specifies the data to be transferred. See the description of *outlist* under "WRITE Statement" for further information.

Examples	**Notes**
`PRINT 10,num,des`	Prints `num` and `des` according to FORMAT statement 10.
`PRINT *,'x=',x`	Prints `'x='` and x according to list-directed formatting.

`ASSIGN 100 TO fmt`
`PRINT fmt,rat,cat`
	Prints `rat` and `cat` according to FORMAT statement 100.

`PRINT '(4I3)',i,j,k*2,330`	Prints `i, j, k*2` and the constant `330` according to the format specification in the PRINT statement itself.

` PRINT 100`
`100 FORMAT ("End of report")`
	Prints the Hollerith constant in the FORMAT statement.

` PRINT '(" x SIN(x) COS(x)"//(I2,2F7.3))',`
`+(i,SIN(i/57.3),COS(i/57.3),i=0,360,5)`
	Prints a literal heading and 73 rows of values as indicated by the implied DO in the input/output list.

The PRINT statement is used only for transferring data from memory to the standard output unit. See Chapter 4 for a more detailed discussion of the PRINT statement.

PROGRAM Statement

The PROGRAM statement defines the name of the main program. As an extension to the ANSI 77 standard, the PROGRAM statement can also be used to access strings passed from the RUN command.

Syntax

```
PROGRAM name [(p1[,p2...pn])]
```

where:

name is the name of the program.

p1,...pn are the names of character variables. These variables must appear in CHARACTER statements or be typed by IMPLICIT statements. They cannot be in common.

The string parameters in the RUN command are copied into *p1,...pn* when the program starts. Null parameters are not copied, and the corresponding variables are uninitialized unless they appear in DATA statements. If a parameter is not null and the variable is defined in a DATA statement, the parameter value takes precedence. Therefore, DATA statements can be used to set default values.

Using the parameter variables is equivalent to using

```
CALL FPARM (p1,...pn)
```

as the first executable statement. See Appendix B for more details on FPARM.

Alternate PROGRAM Statement

The alternate PROGRAM statement also defines the name and, optionally, the type, priority, and time values of the main program in which it appears.

Syntax for RTE-6/VM

```
PROGRAM name[([type,pri,res,mult,hr,min,sec,msec]),comment]
```

Syntax for RTE-A

```
PROGRAM name[([type,pri]),comment]
```

where:

name is the name of the program (and its entry point). If *name* is longer than five characters, the linker may truncate it to five characters.

comment is up to 86 characters passed to the linker.

See the appropriate programmer's reference manual for the definitions of *type, pri, res, mult, hr, min, sec,* and *msec*.

The parameters *type, pri, res, mult, hr, min, sec, msec*, and *comment* are all extensions to the ANSI 77 standard. These parameters are for compatibility with older programs, and should not be used in new programs.

Examples	**Notes**
`PROGRAM prog1(3,90)`	Specifies `prog1` as the name of the program and as type 3 with a priority of 90.
`PROGRAM main`	Specifies `main` as the name of the program.
`PROGRAM proga(,90)`	ILLEGAL. If any parameter is given, all preceding parameters must appear.
`PROGRAM prog2(),Bubble Sort`	To pass comments to the loader, you must specify both parentheses before the comments.

The program statement must be the first noncomment statement in a module, except for certain compiler directives (see Chapter 7 for details).

In the absence of a PROGRAM statement, the program name defaults to "`FTN.`". The type defaults to 4, the priority defaults to 99, and the time defaults to 0.

READ Statement

The READ statement transfers data from a file to program variables. There are two kinds of READ statements:

- READ from the standard input unit
- READ from file

The READ from the standard input unit statement complements the PRINT statement. The READ from file statement complements the WRITE statement. See Chapter 4 for a more detailed discussion of the READ statement.

READ from the Standard Input Unit Statement

The READ from the standard input unit statement transfers data from a unit designated to be the standard input unit (usually the user's terminal).

Syntax

READ *fmt* [, *inlist*]

where:

fmt is 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 specifying the text of a FORMAT statement
- An asterisk(*)

inlist is a list of variables that specify where the data is to be transferred. Each item in *inlist* 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

Functions used in subscripts, substrings, and implied DO loops must not contain any READ, WRITE, or PRINT statements.

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,hl`
	Reads the values of al and hl 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 from File Statement

The READ from file statement transfers data from a file to memory.

Syntax

```
READ([UNIT=]unit[,[FMT=]fmt][,IOSTAT=ios][,ERR=errlabel][,END=endlabel]
    [,REC=rn][,ZBUF=ibuf][,ZLEN=ilen])[inlist]
```

where;

unit is the unit number for the file; *unit* is required. *unit* can be one of the following:

- An arithmetic expression of type integer
- An asterisk (*)
- A qualified external (*unit:sec*[*:ter*])
- A character variable, array element, substring, or array name

An asterisk indicates that the standard input device is to be used.

fmt is 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 specifying the text of a FORMAT statement
- An asterisk (*)

If *fmt* is not present, the access is unformatted.

ios is an integer variable or integer array element name for error return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

errlabel is the statement label of an executable statement. If an error occurs during execution of the READ statement, control transfers to the specified statement rather than aborting the program.

endlabel is the statement label of an executable statement. If an end-of-file is encountered in a sequential file during execution of the READ statement, control transfers to the specified statement. In this case, the variable *ios* is set to −1.

rn specifies the number of the record in a direct access file.

ibuf is an integer variable, array name, or array element name passed as the Z-buffer in a device input request.

ilen is an integer expression that specifies the size of ZBUF (in words).

inlist is a list that specifies the data to be transferred. The items in *inlist* 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

If the `UNIT=` prefix is omitted, *unit* must be the first item in the list. 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. Except for these requirements, the order of parameters is flexible.

Examples	**Notes**
`READ (7,10)a,b,c`	The values of `a, b,` and `c` are read from the file connected to unit 7 according to FORMAT statement 10.

`ASSIGN 4 TO num`
`READ (UNIT=3,ERR=50,FMT=num)z`
	The value of `z` is read from the file connected to unit 3 according to FORMAT statement 4. If an error occurs, control transfers to statement 50.

`READ (10)x`	The value of `x` is read from the file connected to unit 10. Because *fmt* is omitted, the data is unformatted.
`READ (10,FMT=*,END=60)b`	The value of `b` is read from the file connected to unit 10 according to list-directed formatting. If an EOF is encountered, control passes to statement 60.
`READ (2,'(I3)',REC=10)i`	The value of `i` is read from the 10th record of the direct access file connected to unit 2 according to the format specification in the READ statement itself.

A READ from file statement must contain a unit number and at most one of each of the other options. Note that `REC=`*rn* cannot appear with `END=`*endlabel*, `UNIT=*`, nor with `ZBUF,ZLEN`, or secondary and/or tertiary addresses. If `REC=`*rn* appears, the unit must be connected for direct access.

ZBUF and ZLEN are extensions to the ANSI 77 standard. They are used to pass control information to the I/O driver. Also as an extension, secondary or tertiary addresses or both can be passed to the I/O driver by qualifying the `UNIT=` specifier, as in

`UNIT=`*unit*`:`*sec*`[:`*ter*`]`

where:

sec and *ter* are integer expressions.

For further information on reading from a file, see Chapter 5, "FORTRAN File Handling."

REAL Statement

See "Type Statement" later in this chapter for the syntax of REAL and all other type statements.

REAL*4 Statement

See "Type Statement" later in this chapter for the syntax of REAL*4 and all other type statements.

REAL*8 Statement

See "Type Statement" later in this chapter for the syntax of REAL*8 and all other type statements.

RETURN Statement

The RETURN statement transfers control from a subprogram back to the calling program unit.

Syntax

```
RETURN [rtnnum]
```

where:

rtnnum is an integer expression specifying the alternate return ordinal. Normally control is returned from a subroutine to the calling program unit at the statement following the CALL statement. The alternate return statement allows return to the calling program unit at one of a list of labeled executable statements supplied in the CALL statement.

Example

```
    PROGRAM main
       ⋮
    CALL matrx (*10,m,*20,n,k,*30)
       ⋮
10 ...
       ⋮
20 ...
       ⋮
30 ...
       ⋮
    END SUBROUTINE matrx(m,n,k,*,*,*)
       ⋮
    k=2
       ⋮
    RETURN k
    END
```

Notes

The CALL statement specifies three possible return labels, plus the normal return point (the statement following the CALL statement).

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 $1 \leq$ k ≤ 3, control returns to the statement following the CALL statement.

When the RETURN statement occurs in a subroutine and no alternate return is specified, control returns to the first executable statement following the CALL statement that referenced 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 optional integer expression takes on the values:

$1 \leq rtnnum \leq n$

where:

rtnnum identifies the ordinal of the statement label in the actual argument list of the CALL statement. Values outside the range indicate the ordinary return.

n is the number of alternate returns specified by the number of asterisks in the CALL statement.

The asterisks in the SUBROUTINE statement are used for documentation purposes. There should be the same number of asterisks as there are statement labels in the CALL statement.

If *rtnnum* is an integer constant, the value of *rtnnum* must be less than or equal to the number of asterisks in the SUBROUTINE statement. If the constant *rtnnum* exceeds the number of asterisks in the SUBROUTINE statement, a compiler warning is generated, but load and execution are not affected.

Similarly, a warning is generated if any alternate returns are specified and no asterisks appear in the SUBROUTINE statement.

When the value of *rtnnum* is not in the range 1 ≤ *rtnnum* ≤ n, control returns to the statement following the CALL statement. If *rtnum* is a variable or expression, only one asterisk is required in the SUBROUTINE statement, although good programming practice dictates that the number of asterisks in the SUBROUTINE statement always matches the number of labels in the CALL statement.

A CALL and subroutine with alternate returns is equivalent to a computed GOTO and a function:

```
CALL sub(a,b,*10,*20)
   ⋮
SUBROUTINE sub(x,y,*,*)
   ⋮
RETURN k
```

is equivalent to

```
GOTO (10,20), fsub(a,b)
   ⋮
INTEGER FUNCTION fsub(x,y)
   ⋮
fsub = k
RETURN
```

REWIND Statement

The REWIND statement positions a sequential file or device at beginning-of-information.

Syntax

```
           {unit}
REWIND     {([UNIT=]unit[,IOSTAT=ios][,ERR=label])}
```

where:

unit is the unit number of a sequential file or device.

ios is an integer variable or integer array element name for error return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement. If an error occurs during execution of the REWIND statement, control transfers to the specified statement rather than aborting the program.

If the `UNIT=` prefix is omitted, *unit* must be the first parameter in the list. Otherwise the order of parameters is flexible.

Examples	**Notes**
`REWIND 10`	The file connected to unit 10 is positioned at beginning-of-information.
`REWIND (UNIT=5, IOSTAT=j, ERR=100)`	The file connected to unit 5 is positioned at beginning-of-information. If an error occurs, control transfers to statement 100 and the error code is returned in `j`.

If the file is already positioned at beginning-of-information, a REWIND statement has no effect upon the file.

The REWIND statement can be used on direct access files, although it should be avoided to preserve program portability.

SAVE Statement

The SAVE statement causes the values of specified variables in a subprogram to be preserved after execution of a RETURN or END statement.

Syntax

SAVE [*var1*,*var2*,/*com*/,*var3*,...]

where:

var is a simple variable name or an array name.

com is a common block name. The common block name must be preceded and followed by a slash.

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 value of all of the variables in the subroutine `fixit`.

The following items must not be specified 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 appropriate variables in the subprogram are saved.

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 specified in a SAVE statement in one subprogram, it must be specified in a SAVE statement in each subprogram where the common block appears, including the main program.

A SAVE statement is unnecessary in a main program, except when it is required for consistency with saved common blocks in subprograms as described above.

Statement Function Statement

The statement function statement defines a one-statement function.

Syntax

name([*parm1*, *parm2*, . . .])=*exp*

where:

name is the user-specified name of the function.

parm is a formal argument.

exp is an arithmetic, logical, relational, or character expression.

Examples	**Notes**
`disp(a,b,c)=a + b*c`	Defines a statement function `disp` with three arguments.
`tim(t1)=t1/2 + b`	Defines the statement function `tim` with one formal argument. The value of `b` is the current program value of `b` when the function is invoked.

A statement function is a user-defined, single-statement computation that applies only to the program unit where it is defined. A statement function statement can appear only after the specification statements and before the first executable statement of the program unit.

Formal arguments must be simple variables; they can be typed in preceding type statements if necessary. Actual arguments must agree in number, order, and type with their corresponding formal agruments.

The expression defines the actual computational procedure that derives the value. When the statement function is referenced, this value is computed and returned as the function result. The expression must be an arithmetic, logical, relational, 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 type of the name of the function. That is, 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 used in an arithmetic statement function need not be the same type as the function name. (For example, the expression can be type integer even though the function name is type real.) The expression's value is converted to the statement function type before the value is returned.

Statement functions can reference other statement functions or function subprograms. Statement functions cannot contain calls to themselves. They can contain indirect recursive calls, but a function subprogram must be in the call chain; otherwise the recursion could never terminate.

The values of any formal arguments in the expression are supplied at the time the statement function is referenced. All other expression elements are local to the program unit containing the reference. These expression elements derive their values from statements in the containing program unit.

See Chapter 6 for further information about actual and formal arguments and calling a statement function.

STOP Statement

The STOP statement terminates program execution.

Syntax

```
STOP [n]
```

where:

n is an unsigned integer constant (one to five digits) or a character expression.

Examples	**Notes**
`STOP 7777`	The message "`STOP 7777`" is displayed at the terminal.
`STOP 'This is the end!'`	The message "`This is the end!`" is displayed at the terminal.
`STOP`	Nothing is displayed at the terminal.

The STOP statement is used to terminate program execution before the end of the program unit.

Sample Program	**Notes**
`   READ *,a,b`	
`   IF (a .LT. b) STOP 56789`	If `a` is less than `b`, execution terminates.
`10 b = b-1`	
`   IF (b .EQ. a) STOP 'All done'`	When `b` equals `a`, execution terminates.
`   GOTO 10`	
`   END`	

When execution is terminated by a STOP statement, the message sent to the terminal depends on whether digits, characters, or nothing has been specified with the STOP statement. If digits are used, the message `STOP` <*digits*> is displayed at the terminal. If a character expression is used, the character value is displayed. If nothing is used after STOP, nothing is displayed.

As an extension to the ANSI 77 standard, a CALL EXIT statement performs the same function as the STOP statement with no message. However, STOP is preferred.

SUBROUTINE Statement

The SUBROUTINE statement identifies a program unit as a subroutine subprogram.

Syntax

SUBROUTINE *name* [([*arg1*,*arg2*,...][*,...]....)][,*comment*]

where:

name	is the name of the subroutine.
arg	is a formal argument of the subroutine.
*,...	indicates one or more alternate returns.
comment	is text of up to 86 characters passed to the linker. *comment* is an extension to the ANSI 77 standard.

Examples	**Notes**
`SUBROUTINE add`	Begins a subroutine named `add.`
`SUBROUTINE sub(z,i,d,*,*,*)`	Begins a subroutine named `sub` with three arguments and three alternate return points.

The formal arguments in a SUBROUTINE statement can be variables, array names, or subprogram names. They must be of the same type and structure as the actual arguments that are passed to the subroutine.

One or more alternate returns can be specified by asterisks in the SUBROUTINE statement. Alternate returns are described in “RETURN Statement” above.

See Chapter 6 for more information on subroutine subprograms.

THEN Statement

See “Block IF Statement” under “IF Statement” earlier in this chapter for information on the THEN statement.

Type Statement

The type statement is a specification statement that assigns an explicit type to symbolic names that would otherwise have their type implicitly determined by the first letter of their names.

Syntax

type name1[*,name2,* . . .]

where:

type is the type to be associated with the specified variables; *type* can be one of the following:

INTEGER
INTEGER*2
INTEGER*4
REAL
REAL*4
REAL*8
DOUBLE PRECISION
COMPLEX
COMPLEX*8
COMPLEX*16
DOUBLE COMPLEX
LOGICAL
LOGICAL*2
LOGICAL*4
CHARACTER[**len*]

The **n* in the above type specifications is optional.

In `CHARACTER [`**len*`]`, *len* specifies the length of character variables not having their own length specifications; the default value is 1. *len* can be one of the following:

1. An unsigned integer constant.
2. An integer constant expression with a positive value. The integer expression must be enclosed in parentheses, and cannot contain variable names. Example: `(-3 + 4)`.
3. An asterisk enclosed in parentheses (*).

name is a simple variable name, an array name, an array declarator, a named constant, or a function name. When type is CHARACTER, name can have **len* as a suffix designating its length (see the examples that follow). Each name can appear in a type statement only once.

Examples	**Notes**
`INTEGER run,time`	The variables `run` and `time` are single-word integers.
`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.
`INTEGER*4 rn,hours(4,5)`	The variable `rn` and each element of the two-dimensional array `hours` are double integers.

`REAL item`
`DIMENSION item(2,3,5)`
	`item` is a three-dimensional real array.

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

`SUBROUTINE sub (var1 var2, var3, var 4)`	
`CHARACTER*(*) var1`	The variable `var1` is defined as being of type character and as having the same length as the variable `var1` in the calling program.
`CHARACTER*10 var2(*)`	`var2` is a character array with a length of 10; the number of array elements is assumed-size.
`CHARACTER var2(*)*10` . . .	Same as the preceding line.
`CHARACTER *(*) var3(10)`	`var3` is a character array of the same length as the actual argument; it has 10 array elements.
`CHARACTER var3(10)*(*)`	Same as the preceding line.
`CHARACTER*(*) var4(*)`	`var4` is a character array of the same length as the actual argument; the number of array elements is assumed-size.
`CHARACTER var4(*)*(*)`	Same as the preceding line.

If an array declarator is specified in a type statement, the declarator for that array must not be used 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 CHARACTER*(*) form can be used only for named constants, formal arguments, function subprograms, and ENTRY statements.

WHILE Statement

See "DO Statement" earlier in this chapter for information on the WHILE statement.

WRITE Statement

The WRITE statement transfers data from memory to a file.

Syntax

```
WRITE ([UNIT=]unit[,FMT=]fmt[,IOSTAT=ios][,ERR=label][,REC=rn]
   [,ZBUF=ibuf][,ZLEN=ilen]) [outlist]
```

where:

unit is the unit number for the file; *unit* is required. *unit* can be an asterisk, in which case the write is done to the standard output device.

fmt is the format designator. It 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 specifying the text of a FORMAT statement
- An asterisk (*)

ios is an integer variable or integer array element name for error return (see Appendix A for IOSTAT error codes). *ios* is set to zero if no error occurs.

label is the statement label of an executable statement. If an error occurs during execution of the WRITE statement, control transfers to the specified statement rather than aborting the program.

rn specifies the number of the record in a direct access file.

ibuf is an integer array name or array element name passed as the z-buffer in a device output call.

ilen is an integer expression that specifies the size of ZBUF.

outlist is a list that specifies the data to be transferred. Each item in *outlist* must be one of the following:

- A variable name
- An array element name
- An array name
- A substring
- An expression
- An implied DO loop

For syntax and detailed information on implied DO loops, see “Implied DO Loops” under “DO Statement” above.

If *outlist* contains a function reference, that function must not contain any READ, WRITE, or PRINT statements. If *outlist* contains a character expression using concatenation (//), the operands must not have implied lengths (through use of *).

If the prefix `UNIT=` is omitted, *unit* must be the first item in the list. 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. Apart from these requirements the order if flexible.

Example	**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,IOTAT=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.

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

If `REC=`*rn* appears, the file must be connected for direct access, while if `REC=`*rn* does not appear in a WRITE to a direct file, the current position of the file is used. The `END=` specifier cannot appear in a WRITE statement.

ZBUF and ZLEN are extensions to the ANSI 77 standard. They are used to pass control information to the I/O driver. Also as an extension, secondary or tertiary addresses or both can alternately be passed to the I/O driver by qualifying the `UNIT=` specifier, as in:

`UNIT=`*unit*`:`*sec*`[ :`*ter*`]`

where *sec* and *ter* are integer expressions.

The HP-IB driver cannot distinguish between a zero secondary address and a missing secondary address. To ensure that the driver understands that you really do want to send a secondary address of 0, use 400B instead; this sets bit 8 in the address field. Because secondary addressing uses only the low 5 bits, the driver will then properly use secondary address 0. If you prefer, you may set bit 8 on any secondary address.

See Chapters 4 and 5 for more information on the WRITE statement.

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 three types of input/output:

- Formatted input/output
- Unformatted input/output
- List-directed input/output

For each type of input/output, there are one or more input statements and corresponding output statements.

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 a character expression in the input/output statement itself.

Formatted Input

Formatted input is specified by the following input statements:

Syntax

```
     {fmt,list}
READ {(unit, fmt,...optional keywords)  list}
```

where:

fmt is 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 specifying the text of a FORMAT statement.

unit is the unit number of the file (see Chapter 5 for further information).

list is the list of variables that specifies where the data is to be transferred. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" in Chapter 3. If *list* is omitted, the file pointer is positioned at the next record without data transfer.

See "READ Statement" in Chapter 3 for more specific syntax and information on the optional keywords.

The first READ statement syntax shown above (*fmt,list*) is used for transferring information from the standard input unit. The second READ statement syntax is used for transferring data from a file or device. (Files and READ statement options are discussed in Chapter 5.)

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 agree with the list. If the record is shorter than the format specification, the record is treated as if blanks were added to the end to match the format specification. If the list is longer than the format specification, the file skips to the next record and reads it using part or all of the format specification again. This process continues until the list is satisfied. 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. This is because any subsequent READ statement reads values from the next record, unless the file pointer is repositioned before the next READ.

Array names in the list represent all the elements in the array. Values are transferred to the array elements according to the standard array storage order (see Chapter 2).

Formatted Output

Formatted output is specified by the following output statements:

Syntax:

PRINT *fmt,list*

WRITE (*unit, fmt,...optional keywords*) *list*

where:

fmt is 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 specifying the text of a FORMAT statement.

unit is the unit number of the file (see Chapter 5 for further information).

list is a list of variables or expressions that specifies the data to be transferred; if *list* is omitted, a blank line is written. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" in Chapter 3.

See Chapter 3 for more specific syntax and information on the optional keywords.

The PRINT statement is used for transferring information to the standard output unit. The WRITE statement is used for transferring information to disk files or to output devices. (Files and WRITE statement options are discussed in Chapter 5.)

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.

The first character of the output record is always considered to be a carriage control character for devices that recognize carriage control. The ANSI 77 standard carriage control characters are listed in Table 4-1. (Some HP printers do not conform to the standard.)

Some older HP printers do not conform to the ANSI 77 standard. The standard requires that any paper advance be done before printing. However, for an unbuffered printer, this generally forces the printer to run at half speed. This is because the printer must wait for the paper to advance before printing the line. By performing the advance after printing, older printers can overlap the paper advance with the transmission of the next line from the computer. Newer printers with larger buffers print at full speed regardless of when the paper advance is done.

The carriage control characters for older printers are listed in Table 4-2.

Note that the asterisk (*) rather than the plus sign (+) is the suppress paper advance character. When a standard printer is used to print two lines of text on the same line, the first line must have a + and the second line must have a blank. When some older printers are used, the first line must have a blank and the second line must have an *. Other combinations are also different; for example, with some older printers, it is not possible to overprint the first line on a page if you reached that position using the carriage control character 1.

Table 4-1. Carriage Control Characters

Character	Vertical Spacing Before Printing
Blank	Advance one line (single space).
0	Advance two lines (double space).
1	Advance to first line of next page (page eject).
*	Do not advance (for overprinting).
?	(Any other character; not defined in the standard.) Device dependent; usually the same as a blank.

Table 4-2. Non-Standard Carriage Control

Character	Printer Action
Blank	Print line and then advance one line (single space).
0	Advance one line, print line, advance one line (double space).
1	Advance to top of form, print line, advance one line (page eject).
*	Print line; do not advance. (See note below.)
?	(Any other character.) Device dependent; usually the same as a blank.
Note: Some printers advance one line as a side effect of printing and thus cannot implement the * feature.	

Format Specifications

A format specification is a list of format descriptors and edit descriptors. The format descriptors describe how the data is converted between internal form and ASCII, and edit descriptors specify editing information.

Format specifications can be given in FORMAT statements or 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. The form of the FORMAT statement is:

label `FORMAT` *format_specification*

Example **Notes**

```
   READ(10,10)a,i,d,e
10 FORMAT(A2,I3,D8.2,F12.2)
```

The format specification

`(A2,I3,D8.2,F12.2)`

corresponds to the variables `a`, `i`, `d`, and `e` in the READ statement. List element `a` corresponds to the format descriptor `A2`, `i` correspond to `I3`, `d` corresponds to D8.2, and `e` corresponds to `F12.2`.

A FORMAT statement can be referenced by several input/output statements. Ensure that each variable in the input/output list corresponds with the appropriate format descriptors 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(UNIT = 4,'(A3,3X,F10.2)')a,z
```

The variables `a` and `z` are read according to the format specification (`A3,3X,F10.2`).

```
CHARACTER a*5
DATA a/'(3I3)'/
PRINT a,i,j,k
```

The three integers, `i`, `j`, and `k`, are printed according to the format specification (`3I3`).

```
WRITE(1,'(F10.2)')d
```

The variable `d` is written as a fixed-point number according to the format specification `(F10.2)`.

To print a single quotation mark within single quotation mark edit descriptors, you must use two consecutive single quotation marks. If the format is contained in an input/output statement, each pair of consecutive single quotation marks must, in turn, be represented by two single quotation marks. For further information see “Character” in Chapter 2.

Examples	**Notes**
`WRITE(6,'(3X,'' THIS IS THE END'')')`	Writes the following record: `ΔΔΔΔTHIS IS THE END` (Each Δ represents a blank.)
`WRITE (1,'(''Ain''''t it true!'')')`	Writes the following on the display screen: `Ain't it true!`
`WRITE (1,'(“Ain''t it true!”)')`	Writes the same as the previous example. (Note that double quotation marks can be used in input/output and FORMAT statements, but single quotation marks are preferred.)

Format and Edit Descriptors

A list of format and edit descriptors makes up a format specification. The format descriptors describe how the data appears, and 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. The format descriptor `3F12.3` specifies three real numbers, each with a field width of 12 and three digits to the right of the decimal point. A PRINT statement referencing this format specification could be of the form:

```
    PRINT 10,item,a,b,c
10  FORMAT (......)
```

Figure 4-1 shows the output data as it might appear in the output record with the field widths indicated.

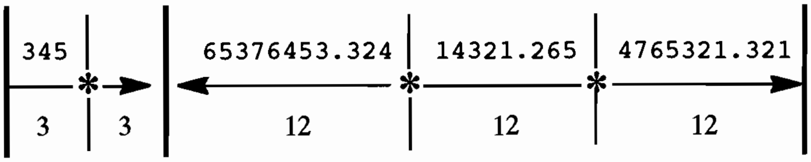


Figure 4-1. Output Data

The descriptors in a format specification must be separated by a comma except before and after a slash (/) edit descriptor, a colon edit descriptor, a literal edit descriptor, or a scaling (P) 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. These two are equivalent:

```
3I2,F4.0,/I5,F12.6
```

and

```
3I2,F4.0/I5,F12.6
```

Format descriptors and edit descriptors (except H and P) can be preceded by a repeat specification (such as 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. For example, the information shown on the input record in Figure 4-2 below could be represented in the following FORMAT statement:

```
10 FORMAT (I3,F7.4,3(F7.2,I3),F12.4)
```

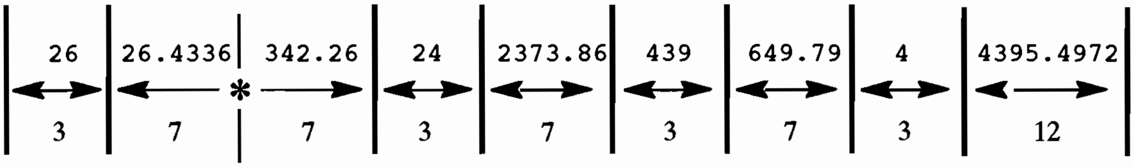


Figure 4-2. Input Data

A READ statement corresponding with the FORMAT statement above could be

```
READ 10,i,a,b,j,d,k,e,m,f
```

The READ statement would read values for `i` and `a`, then repeat the parenthetical statement (`F7.2,I3`) three times to read values for `b` and `j`, `d` and `k`, and `e` and `m`, and, finally, read a value for `f`. Nesting of format and edit descriptors is limited to five levels.

If a FORMAT statement specifies a record size larger than 134 bytes (67 words), the LGBUF routine must be called to provide a larger input/output buffer. (For more information on LGBUF, refer to “Input/Output Library Interface Functions” in Chapter 6.) If LGBUF is not used, an input/output run-time error 496 is generated.

The format descriptors are summarized in Table 4-3 and the edit descriptors in Table 4-4. A detailed explanation of the descriptors follows the tables.

Table 4-3. Format Descriptors

Data Type	Format Descriptor	Additional Explanation
Integer and Double Integer	I*w*[.*m*]	
Real, Double Precision, Complex, and Double Complex	F*w.d*	Fixed-point format descriptor
Real, Double Precision, Complex, and Double Complex	D*w.d*[E*e*] E*w.d*[E*e*]	Floating-point format descriptor
Real, Double Precision, Complex, and Double Complex	G*w.d*[E*e*]	Fixed- or floating-point format descriptors
Integer	K*w* @*w* O*w*	Octal
Logical	L*w*	
Character	A[*w*]	Character data is left-justified in memory and right-justified in external format*
Character	R[*w*]	Character data is right-justified in memory and external format*

* See Table 4-5 for exact format.

Table 4-4. Edit Descriptors

Descriptor	Function
BN	Ignore blanks
BZ	Treat blanks as zeros
*n*X	Skip *n* positions to the right
T*c*	Skip to column *c*
TR*c*	Skip *c* positions to the right
TL*c*	Skip *c* positions to the left
/	Begin new record
:	Terminate format if remaining list empty
'...'	Literal editing
"..."	Literal editing
S	Processor determines sign output; same as SS on HP 1000
SP	Output optional plus signs
SS	Inhibit optional plus sign output

Numeric Format Descriptors

The numeric format descriptors are used to specify the input/output fields of integer, double integer, real, double precision, complex, and double complex data. 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 ignored 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 zero.

- On output, the data is right-justified in the field. If the data length is less than the field width, leading blanks are inserted in the field. If the data is longer than the field width for any descriptors the decimal point, if present, may be moved to make the result fit in the field. If the result still does not fit, the entire field is filled with asterisks, as specified in the output examples of the particular descriptors.

- At compile time, the sizes of *w*, *d*, and *n* fields within format specifications are checked only for values greater than 2047.

- 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 Descriptors: Iw and Iw.m

The I*w* and I*w.m* format descriptors define a field for an integer or double integer number. The corresponding input/output list item must be a numeric type.

On input, an I*w* or I*w.m* 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 item currently using the descriptor. A plus sign is optional for positive values. For compatibility with older programs, a decimal point is accepted in the field without being flagged as an error. However, new programs should not place a decimal point in the field. The *m* value is ignored on input.

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 a position for a minus sign for negative values. The optional *m* value specifies a minimum number of digits to be output. If *m* is not supplied, a default value of 1 is assumed. If *m* = 0, a 0 value is output as all blanks. When *m* is supplied, the corresponding list item should be of type INTEGER.

Descriptor	Internal Value	Output
I4	+452.25	Δ452
I2	+6234	**
I3	−11.92	−12
I5	−52	ΔΔ−52
I3	−124	***
I10	123456.5	ΔΔΔΔ123457
I6.3	3	ΔΔΔ003
I3.0	0	ΔΔΔ

Real and Double Precision Format Descriptors: Fw.d, Ew.d[Ee], Dw.d[Ee], and Gw.d[Ee]

The F*w.d*, E*w.d*[E*e*], D*w.d*, and G*w.d*[E*e*] format descriptors define fields for real, double precision, complex, and double complex numbers. (Note that two descriptors must be specified for complex and double complex values.) The input/output list item corresponding to a F*w.d*, E*w.d*[E*e*], D*w.d*[E*e*], or G*w.d*[E*e*] descriptor must be a numeric type.

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 last *d* digits 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:

Descriptor	Input Field	Value Stored
F6.5	4.51E4	45100
G4.2	51−3	.00051
E8.3	7.1ΔEΔ5	710000.
D9.4	ΔΔΔ45E+35	$.0045 \times 10^{35}$
BZ,F7.1	−54E24Δ	-5.4×10^{240} Error (overflow)
F2.10	34	34×10^{-10}

Note that the value of *d* is used as a scale factor; *d* can be greater than the number of digits in the field.

The appearance of the output field depends on whether the format descriptor specifies a fixed- or floating-point format.

Fixed-Point Format Descriptor: Fw.d

The F*w.d* format descriptor defines a fixed-point field on output for real, double precision, complex, and double 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.

Descriptor	Internal Value	Output	
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	−1000.0	(Adjusted to fit.)
F4.1	23	23.0	

Floating-Point Format Descriptors: Ew.d[Ee] and Dw.d[Ee]

The E*w*.*d*[E*e*] and D*w*.*d*[E*e*] format descriptors define a normalized floating-point field on output for real, double precision, complex, and double complex values. The value is rounded to *d* digits. The exponent part consists of *e* digits. If E*e* is omitted, then the exponent occupies two positions. The field width, *w*, should follow the general rule:

$w \geq d + 7$

or, if E*e* is used:

$w \geq d + e + 5$

to provide positions for a leading blank, the sign of the value, the decimal point, *d* digits, the letter D or E, the sign of the exponent, and the exponent. This is a recommendation only, because the input/output library adjusts the specifications to represent the output for even the following:

```
D10.9, E10.10, G10.11
```

The form D*w*.*d*E*e* is an extension to the ANSI 77 standard. The standard form is D*w*.*d*.

Descriptor	Internal Value	Output
D10.3	12.342	ΔΔ.123Δ+02
E10.3E3	−12.3454	−.123E+002
E12.4	+12.340	ΔΔΔ.1234E+02
D12.4	−.00456532	ΔΔ−.4565D−02
E10.8	99.99995	.10000E+03 †
D10.9	99.999123	.99999D+02 †
E11.5	+999.997	Δ.10000E+04
E10.3E4	$.624\text{x}10^{-30}$	.624E−0030

† The input/output library adjusts the format specification to avoid outputting all asterisks.

Fixed- or Floating-Point Format Descriptor: Gw.d[Ee]

The G*w*.*d*[E*e*] format descriptor defines a fixed- or floating-point field, as needed, on output for real, double precision, and complex values. The G*w*.*d*[E*e*] 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*-*n*) format descriptor is used, where n is the number of nonzero digits before the decimal point. That is, the decimal point is placed so that exactly d digits are produced. 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, then the exponent occupies two positions. The field width, w, should follow the general rule for floating-point descriptors:

$$w \geq d + 7$$

or, if E*e* is used:

$$w \geq d + e + 5$$

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.

Field Descriptor	Internal Value	Interpreted As	Output
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	E12.1	ΔΔΔΔΔΔ.1E+05
G8.2	+999	E8.2	Δ.10E+04
G7.2	−999	E7.2	−.1E+04†

† The input/output library adjusts this specification to show the output value, instead of filling the field with asterisks.

Character Format Descriptors: A[w] and R[w]

The A[*w*] and R[*w*] format descriptors define fields for character data. The size of the list variable (the 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.

When the A[*w*] and R[*w*] format descriptors are used 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. The possibilities are summarized in Table 4-5.

Table 4-5. 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*	Output as the value, preceded by blanks.
R[*w*]	*w*<*len*	Taken from right part of variable.
	w>*len*	Output as the value, preceded by blanks.

In the following examples, ƶ represents a binary 0.

Descriptor	**Input Characters**	**Variable Length**	**Value Stored**
A3	DEF	3	DEF
R3	DEF	4	ƶDEF
A5	ABCΔΔ	10	ABCΔΔΔΔΔΔΔ
R9	RIGHTMOST	4	MOST
A5	CHAIR	5	CHAIR
R8	CHAIRΔΔΔ	8	CHAIRΔΔΔ
A4	ABCD	2	CD

On output, if *w* is greater than or equal to the specified byte size (**n*), of the output variable, the data is right-justified within a *w* character field. If *w* is less than *n*, only the leftmost *w* characters appear in the output field. If *w* is less than *n* and R*w* is used, only the rightmost *w* characters appear in the output field.

Descriptor	Internal Characters	Variable Length	Output
A6	ABCDEF	6	ABCDEF
R4	ABCD	4	ABCD
A4	ABCDE	5	ABCD
A8	STATUS	6	ΔΔSTATUS
A4	NEXT	4	NEXT
R8	STATUS	6	ΔΔSTATUS
R4	STATUS	6	ATUS

Logical Format Descriptor: Lw

The L*w* format descriptor defines a field for logical data. The input/output list item corresponding to a L*w* descriptor must be of type logical.

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) is used to determine the value to be stored in the declared logical variable. If this first nonblank character is not a T or an F, an error is generated.

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.

Descriptor	Internal Value	Output
L5	.FALSE.	ΔΔΔΔF
L4	.TRUE.	ΔΔΔT
L1	.TRUE.	T
L2	.FALSE.	ΔF

The logical value true or false is determined by the leftmost bit in the internal data storage: 1 = true, 0 = false.

Octal Format Descriptors: Kw, @w, and Ow

As an extension to the ANSI 77 standard, the K*w*, @*w*, and O*w* descriptors define a field for octal data. These descriptors provide conversion between an external octal number and its internal representation. The list elements must be of type integer.

No more than six octal digits can be input or output. If any nonoctal digit is used, or the value exceeds 177777, an error occurs.

On output, if *w* is greater than or equal to 6, six octal digits are written right-justified in the output field. If *w* is less than 6, the *w* least significant octal digits are written.

Note that double-word integer values cannot be input or output in the octal format.

Input Examples

Descriptor	Input Field (Octal)	Value Stored (Octal)
@6	123456	123456
O7	−123456	error: no sign allowed
2K4	Δ396ΔΔΔ5	000036 and 000005

Output Examples

Descriptor	Internal Value (Decimal)	Output (Octal)
K6	99	000143
O2	99	43
@8	−1	ΔΔ177777
@6	32767	077777

Edit Descriptors

Edit descriptors specify editing between numeric, Hollerith, and logical fields on input and output records. There are 11 edit descriptors. Two descriptors, BN and BZ, apply only to input, and two descriptors, "...", and '...', apply only to output. The other seven descriptors, *n*H, /, T*n*, TL*n*, TR*n*, :, and *n*X, apply to both input and output.

Blank Interpretation Edit Descriptors: BN and BZ

The BN and BZ edit descriptors are used to interpret embedded and trailing blanks in numeric input fields. At the beginning of execution of an input statement, blank characters within numbers are ignored. (An exception to this rule occurs when the unit is connected with BLANK = 'ZERO' specified in the OPEN statement. See Chapter 5 for more details.) 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.

The BN and BZ edit descriptors affect only I, F, E, D, and G format descriptors during execution of an input statement; BN and BZ have no effect during execution of an output statement.

Descriptor	Input Characters	BN Editing In Effect	BZ Editing In Effect	
I4	1Δ2Δ	12	1020	
F6.2	Δ4Δ.Δ2	4.2	40.02	
E7.1	5Δ.ΔE1Δ	$5.x10^{1}$	$50.0x10^{10}$	
I2	ΔΔ	0	0	
E5.0	3E4ΔΔ	$3.x10^{4}$	$3.x10^{400}$	(overflow)

Literal Edit Descriptors: '...' and "..."

To write a character constant string, surround the string with single or double quotation marks ('...' or "..."). The width of the field is the number of characters, including blanks, contained between the quotation marks.

To print a single or double quotation mark, you must either surround the string with quotation marks of the opposite kind or type two quotation marks of the same kind. For example, to print a single quotation mark, either type two single quotation marks and surround the string with single quotation marks, or type one single quotation mark and surround the string with double quotation marks.

Single and double quotation mark delimiters cannot be used in input.

Descriptor	Field Width	Output
’BEGIN DATA INPUT’	16	BEGIN DATA INPUT
“DAVID’S TURN”	12	DAVID’S TURN
“THE ENDΔΔΔ”	10	THE ENDΔΔΔ
‘ΔΔSPACESΔΔ’	10	ΔΔSPACESΔΔ
“$*[/\^<;,#!”	11	$*[/\^<;,#!
’Ain”t’	5	Ain’t
””””	1	”

Position Edit Descriptor: nX

The *n*X edit descriptor is used to skip *n* positions of an input/output record; *n* must be a positive nonzero integer.

On input, the *n*X edit descriptor causes the next *n* positions of the input record to be skipped.

Descriptor	Input Record	Value Stored
F6.2,3X,I2	673Δ21END45	673.21, 45
1X,I2,A3	$6ΔEND	6, END

On output, the *n*X edit descriptor causes *n* positions of the output record to be filled with blanks. (If the positions were already defined, they are left unchanged. This can happen when T*c* or TL*c* is used.)

Descriptor	Internal Value	Output
F8.2,2X,I3	5.87, 436	ΔΔΔΔ5.87ΔΔ436
F4.2,3X,“TOTAL”	32.4	32.4ΔΔΔTOTAL

The *n*X descriptor is identical to TR*n*.

Tab Edit Descriptors: Tn, TLn, and TRn

The tab edit descriptors are used to position the cursor in the input or output record. The T*n* edit descriptor references absolute column numbers (*n*), while the descriptors TL*n* and TR*n* reference a relative number of column positions to the left (TL*n*) or right (TR*n*) of the current cursor position. Note that the TR*n* descriptor is identical to the *n*X descriptor.

The term *column* refers to a character position in the record. If the record contains special characters, the column position specified in the tab descriptor may not match the external representation (such as the CRT display).

Descriptor	Value	Output
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

Descriptor	Record	Stored
A4,T1,F4.0	1234	'1234',1234.0

Record Terminator Edit Descriptor: /

The / edit descriptor terminates the current record and begins processing 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.

If a series of two or more / edit descriptors are written in a format specification, as many records as there are slashes are skipped. If a format contains only *n* slashes, *n* + 1 records are skipped. The / edit descriptor does not need to be separated from other descriptors by commas.

Colon (Fence) Edit Descriptor: (:)

The colon (:) edit descriptor terminates format control (just as if the final right parenthesis in the format specification had been reached) if there are no more items in the input/output list. If more items remain in the list, the colon edit descriptor has no effect.

Format	Stored Value	Output
(10(' value=',I2))	1,2	value= 1 value= 2 value=
(10(:,' value=',I2))	1,2	value= 1 value= 2

Scale Factor: nP

The scale factor, *n*P (*n* is the scale value), is a descriptor that modifies the normalized output of the E*w.d*, D*w.d*, and G*w.d* (when 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 F*w.d*, E*w.d*, D*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 input values containing an exponent.

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 previous value, 0, remains in effect.
(E10.3,2PF12.4,I9)	Scale factor remains at 0 for E10.3, changes to 2 for F12.4, and has no effect on I9.

On input, the scale factor affects fixed-field (no exponent) input to the F*w.d*, E*w.d*, D*w.d*, and G*w.d* format descriptors. The external value is divided by 10 raised to the (*n*)th power, as illustrated below.

Scale Factor and Format Descriptor	Input Value	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*

* If the input includes an exponent, the scale factor has no effect.

On output, the scale factor affects F*w.d*, E*w.d*, D*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 right *n* places while reducing the exponent by *n* (the overall value remains the same) as illustrated below. The number of significant digits printed is equal to (*d*) for negative or zero scale factors and (*d*+1) for positive scale factors.

Scale Factor and Format Descriptor	Internal Value	Value Stored
E12.4	12.345678	ΔΔΔ.1235E+02
3PE12.4	12.345678	ΔΔ123.46E−01
−3PD12.4	12.345678	ΔΔΔ.0001D+05
1PG10.3	1234	Δ1.234E+03

For the F*w.d* format descriptor, the internal value is multiplied by 10 raised to the (*n*)th power, as illustrated below.

Scale Factor and Format Descriptor	Internal Value	Output
F11.3	1234.500	ΔΔΔ1234.500
−2PF11.3	1234.500678	ΔΔΔΔΔ12.345
2PF11.3	1234.500678	Δ123450.068

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 precede a F*w.d*, E*w.d*, D*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 F*w.d*, E*w.d*, D*w.d*, or G*w.d* 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 F, E, D, and G specifications until either the end of the FORMAT statement 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. The largest repeat factor allowed is 2047. If a scale factor is needed also, it is written to the left of the repeat specification.

Examples	**Notes**
`(3F10.5)`	is equivalent to `(F10.5,F10.5,F10.5)`
`(2I3,2(3X,A5))`	is equivalent to `(I3,I3,3X,A5,3X,A5)`
`(L2,2(F2.0,2PE4.1),I5)`	is equivalent to `(L2,F2.0,2PE4.1,F2.0,2PE4.1,I5)`
`(2P3G10.4)`	is equivalent to `(2PG10.4,G10.4,G10.4)`

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.

As an extension to the ANSI 77 standard, the repeat specification can also be used in conjunction with a record terminator edit descriptor (for example, `3/`) and with literal edit descriptors (for example, `5" COLUMN" or 5' COLUMN'`).

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. A maximum of five levels of nesting is allowed in FORTRAN 77 format specifications.

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 nested level 2.
`(A,(3X,(I2,(A3)),I3),A)`	One group at nested level 1, one at level 2, and one at level 3.

A formatted input/output statement references each element of 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, these three steps are performed:

1. The current record is terminated.
2. A new record starts.
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.

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:

Syntax

```
     {*,list}
READ {(unit,*,..optional keywords) list}
```

where:

unit is the unit number of the file (see Chapter 5 for further information).

list is a list of variables that specifies where the data is to be transferred. If *list* is omitted, the file is positioned at the next record without data transfer. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" in Chapter 3.

See "READ Statement" in Chapter 3 for detailed information on the syntax and meaning of the optional keywords.

The first READ statement syntax shown above is used for transferring information from the standard input device. The second READ statement is used for transferring data from a disk file or device. (Files, along with other READ statement options, are discussed in Chapter 5.)

Input data for list-directed input consists of values separated by one or more blanks, or by a 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 to be part of a value separator unless followed by a comma. Input data can also take the forms

*r*c*

or

*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 occurs in the input record.

Embedded blanks are not allowed in input values (a blank is always interpreted as a value separator). The forms of values in the input record are as follows:

Integers Same form as an integer constant (see Chapter 2).

Octal Consists of the character @ followed by a field of octal digits. (Octal is an extension to the ANSI 77 standard.)

Real and Double Precision Any valid form for real and double precision constants (see Chapter 2). In addition, the exponent can be indicated by a signed integer constant (the D or E can be omitted), and the decimal point can be omitted for those values with no fractional part.

Complex and Double Complex Two integer, double integer, real, or double precision constants, separated by commas and enclosed in parentheses. The first number is the real part of the complex or double complex number, and the second number is the imaginary part. Each of the numbers can be preceded or followed by blanks.

Logical Consists of a field of characters, the first nonblank character of which must be a T for true or an F for false (excluding an optional leading decimal point).

Character Same form as a character constant. 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. Because the FMGR file system cannot represent odd record lengths (bytes), a record can have one more character than expected.

As an extension to the ANSI 77 standard, a character value can appear without the single quotation marks. In this case, the constant must not contain embedded blanks, commas, or slashes. Constants of this form are not continued from one record to the next; the first blank, comma, slash, or end-of-record terminates the constant.

The data in the input record is converted to that of the list item, following the same assignment rules as given in Table 3-3 in Chapter 3.

For example, the statement

```
READ *,a,b,c,d,e
```

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	Type	Value Assigned
a	Character	TOTAL
b	Complex	(42.,1.)
c	Logical	true
d	Real	362.
e	Double Precision	563.63x10 ^{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 item is defined, it retains its value, or, if the item 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 placing a comma before the first input value on a line.

Examples

The statement

```
READ *,a,b,c
```

and the input record

```
Δ,5.12Δ,ΔΔ,
```

cause the following assignments to take place:

Variable	Type	Value Assigned
a	Real	Retains previous value
b	Real	5.12
c	Real	Retains previous value

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 as 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.

List-Directed Output

List-directed output is specified by the following output statements:

Syntax:

```
PRINT *,list

WRITE (unit,*,...optional keywords) list
```

where:

unit is the unit number of the file (see Chapter 5 for further information).

list is a list of variables or expressions that specifies the data to be transferred. If *list* contains a function reference, that function must not contain any READ or WRITE statements. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" in Chapter 3.

See "WRITE Statement" in Chapter 3 for details on the syntax and meaning of the optional keywords.

The PRINT statement is used for transferring information to the standard output unit. The WRITE statement is used for transferring information to external files or devices. (Files, along with other WRITE statement options, are discussed in Chapter 5.)

The forms of values in a list-directed output records are as follows:

Integer Output as an integer constant.

Real and Double Precision Output with or without an exponent, depending on the magnitude of the value.

Complex Output as two numeric values separated by commas and enclosed in parentheses.

Logical A T is output for the value true and an F for the value false.

Character A character value is not delimited by single or double quotation marks, and each single or double quotation mark within the value is represented by 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.

If the field is longer than the number of character positions left in the record, the current record is written and a new one started. The default limit on record size is 72 characters. This limit can be changed with the library routine FFRCL, described in Appendix B.

Examples

Internal Values (Types)

a = 11.15 (REAL)

b = .11145D-05 (DOUBLE PRECISION)

c = (10,3.0) (COMPLEX)

d = (1.582D-03,4.9851) (COMPLEX*16)

e = .TRUE. (LOGICAL)

f = .FALSE. (LOGICAL*4)

i = 11250 (INTEGER)

j = -32799 (INTEGER*4)

n = 'PROGRAM NAME' (CHARACTER*15)

p = 'TEST::RT' (CHARACTER*8)

Output Statement	**Output Record**
PRINT *,a,i	Δ11.15Δ11250 (Output to the standard output unit. The first character, a blank, is not shown; it is used as carriage control.)
WRITE(1,*)c	Δ(10.,3.)
WRITE(1,*)j,e	Δ-32799ΔT
PRINT *,b	Δ1.1145E-6
WRITE(1,*) d	Δ(1.582E-3,4.9851)
WRITE(1,*)n,p	PROGRAMΔNAMEΔΔΔTEST::RT

If the length of the values of the output items is greater than 72 characters, a new record is begun.

Slashes, as value separators, and null values are not output by list-directed formatting.

Unformatted Input/Output

Unformatted input/output allows you to transfer data in internal (binary) representation. Each unformatted input/output statement transfers exactly one record. Unformatted input/output to devices is done in binary mode.

On input or output, if the record size exceeds 120 bytes (60 words), the LGBUF routine must be called to supply a larger input/output buffer. If LGBUF is not used, an input/output run-time error is generated. (Note that the buffer size limit for paper tape is 118 bytes.)

Unformatted Input

Unformatted input is specified by the following input statement:

Syntax

`READ` (*unit*, . . .*optional keywords*) *list*

where:

unit is the unit number of the file (see Chapter 5 for further information).

list is a list of variables that specifies where the data is to be transferred. If *list* is omitted, the file is moved to the next record without data transfer. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" in Chapter 3.

None of the optional keywords can be FMT=.

See "READ Statement" in Chapter 3 for a detailed description of the syntax and meaning of the optional keywords.

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 or double precision values.

The type of each input value should agree with the type of the corresponding list item. However, a complex or double complex value in the input record can correspond to two real or double precision list items, or two real or double precision values can correspond to one complex list item.

The data is transferred exactly as it was written; thus, no precision is lost.

The input record is a stream of bytes, not words. If an odd-length character item is input, the next item begins at an odd byte offset (in the middle of a word).

The 60-word input/output buffer size applies to an unformatted READ statement. A cause of input/output error number 496 is an attempt by an unformatted READ to input values when not enough data is read (that is, the record is too small). You can avoid this problem by including the ERR= and IOSTAT= options, or both, identifying the cause of the error, then using the ITLOG function to find the actual length of the transmitted record. See Chapter 6 for more information on the ITLOG function.

Unformatted Output

Unformatted output is specified by the following statement:

Syntax

```
WRITE (unit,...optional keywords) list
```

where:

unit is the unit number of the file (see in Chapter 5 for further information).

list is a list of variables or expressions that specifies the data to be transferred. *list* can contain implied DO loops. For syntax and detailed information on implied DO loops, see "Implied DO Loops" under "DO Statement" 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.

None of the optional keywords can be FMT=.

See "WRITE statement" in Chapter 3 for a detailed description of the syntax and meaning of the optional keywords.

The output list must not specify more values than can fit into one record. If the specified values do not fill a direct access record, the remainder of the record is undefined. (Because sequential records are variable length, they have no remainder.) The data is transferred exactly as it is stored in memory; thus, no precision is lost. The output record is a stream of bytes, not words. If an odd-length character item is output, the next item begins at an odd byte offset (in the middle of a word).

The 60-word input/output buffer size applies to an unformatted WRITE statement. A cause of input/output error 496 is an attempt by an unformatted WRITE to output more values than one record can hold.

5

FORTRAN File Handling

The input/output statements described in Chapter 4 (READ, WRITE, and PRINT) reference a unit number of a file. The unit number refers to a FORTRAN logical unit (LU) number assigned to a disk file or a peripheral input/output or storage device. FORTRAN 77 allows access to disk and nondisk units through two methods, both of which assign a FORTRAN logical unit number to the unit. If a nondisk device or spool file is present in your session environment (SST), or in the nonsession environment, you have access to a system unit through its system logical unit number, and the unit is said to be preconnected. You can access all preconnected units without executing an OPEN statement.

The second method of accessing files in FORTRAN is by connecting a disk file to a FORTRAN logical unit number in an OPEN statement. The OPEN statement can also be used on devices and spool files to assign standard FORTRAN unit numbers or change certain specifications.

This chapter contains details on the types of files, methods of assigning a unit number, control specifications, FORTRAN DS methods, and other file status and manipulation procedures. For the syntax and a discussion of each file handling statement, see Chapter 3.

File Definition

A file in FORTRAN is defined as a collection of related information logically organized into records. A file can be stored on disk or can reference nondisk peripheral devices by LU or name (that is, a type 0 file). The information in files can consist of programs or data. For a detailed discussion of the types of files available, see the appropriate programmer's reference manual.

A record is defined as a sequence of data values or characters. A record does not necessarily refer to a physical entity (such as a punched card), but refers to a logical representation of data or characters.

The three types of records are:

- Formatted
- Unformatted
- End-of-file

A formatted record consists of data that is edited during both the input and output processes. The length of a formatted record is measured in characters. Formatted records should be read or written only by formatted and list-directed input/output statements, and can contain a maximum of 134 characters (67 words) unless the routine LGBUF is used (see Appendix B for information on LGBUF).

An unformatted record consists of data that can be read or written without incurring the overhead of editing. The length of an unformatted record is measured in bytes (the maximum is 120 bytes per record unless LGBUF is used). An unformatted record should be read or written only by unformatted input/output statements.

An end-of-file record is the last logical record of a sequential file. This record is written by the ENDFILE statement, and contains no data. As an extension to the ANSI 77 standard, some devices can contain multiple end-of-file records.

The terms *external file* and *internal file* are defined as:

external file is a file located on a storage medium external to the program (such as disk) or an external device.

internal file is an area of storage that is internal to the program, such as an array in main storage.

Moving data from one internal storage to another and converting data can be done more easily with internal files. See "Internal Files" later in this chapter for more information.

This chapter uses the following terms to specify positioning within a file:

current record is the record within which the pointer is currently positioned.

file pointer is the current position within a file.

initial point is the position just before the first record of the file.

next record is the next record to be read or written; if the file pointer is at the terminal point, there is no next record.

previous record is the record just read or written; if the file pointer is at the initial point, there is no previous record.

terminal point is the position just after the last record of the file.

File Access

External files are categorized by the method of access: either sequential or direct. (Access means to read from or write to a file.) Some files allow both access methods, while others are restricted to one access method. For example, a file with a constant and known record length, storing either ASCII or binary data, can allow both sequential and direct access, while a file with variable length records can only be accessed sequentially. (See the appropriate programmer's reference manual for the specific file types and access methods allowed.)

Sequential access is the accessing of records in the order in which they were written. A sequential file may contain both formatted and unformatted records. A sequential file is terminated by an end-of-file record.

Direct access refers to the access of the records in any order, by record number. Reading and writing records is done by direct access input/output statements (that is, READ and WRITE statements containing a `REC=` specification). Each record of the file is identified by a record number, which is a positive integer. Once established, a record number of a specific record cannot be changed or deleted, although the record can be rewritten.

Records can be read or written in any order. For example, record 3 can be written before writing record 1. The records of a direct access file cannot be read or written using list-directed formatting. A direct access file does not contain an end-of-file record as an integral part of the file with a specific record number; therefore, when accessing a file with a direct access READ or WRITE statement, the END= specification is not allowed.

$FILES Directive

The $FILES compiler directive must be specified in a FORTRAN main program to set aside a storage area describing any connection between a FORTRAN logical unit and a system logical unit or file. Programs that use only the preconnected units (that is, that do not contain any OPEN statements) can be executed without the $FILES directive.

The $FILES directive can also be specified in a subprogram. It has an effect only if the program does not use disk files and the $FILES directive in the subprogram specifies zero for the number of disk connections. This can cause a reduction in the size of the loaded program (see "Reducing the Size of a Loaded Program" in Chapter 7). If the program uses files, a $FILES directive in a subprogram has no effect.

The general form of the $FILES directive is:

```
        {m,n[,s[,b]]}
$FILES  {m,n,s,FREESPACE}
        {m,n,DS}
```

where:

m is the maximum number of nondisk units that can be connected at one time (0 to 128, default = 0).

n is the maximum number of disk units that can be connected at one time (0 to 128, default = 0).

s is the default number of 128-word blocks in each Data Control Block (DCB) buffer for each file (1 to 128, default = 1). You can override this default for any given file through the BUFSIZ parameter in the OPEN statement.

b is the total number of 128-word buffer blocks that can be allocated (default = *n***s*).

FREESPACE indicates that the area at the end of the user partition is available for use as 128-word buffer blocks. The NFIOB function allows you to determine the number of input/output buffers available for allocation. The SZ system command tells you how much free space your program has. The SZ command in LINK lets you increase the free space in your program. See the examples in "Specifiers in the OPEN Statement" later in this chapter.

DS allows connections to remote nodes when using the FMGR file system. When the CI file system is used, remote connections are always allowed and DS must not be specified.

Example

```
$FILES 2,3
     PROGRAM filex
     :
```

Notes

The $FILES directive begins in column 1 and should appear before the first FORTRAN statement in the program unit. This directive reserves storage in the main program of two nondisk units and three disk units, using the default Data Control Block (DCB) buffer size of one block.

The *m* and *n* parameters are required positional parameters that indicate the maximum number of connections to be made simultaneously within the main program and all subprogram units. The default when no directive appears in the main program unit is:

```
$FILES 0,0
```

The size of the table area in the main program can be approximated by:

$$3*(m + n) + n*(32 + 128*s)$$

or, if FREESPACE is used, by:

$$3*(m + n) + 16*n$$

or, if DS is used, by:

$$3*(m + n) + 20*n$$

For further information about the Data Control Block (DCB) and system file structures, see the appropriate programmer's reference manual.

See Chapter 7 for further information about the $FILES directive.

File Existence and Connection

A unit is said to exist for an executable program if the unit is preconnected for use, or if the unit is connected by the program through an OPEN statement. At any one time a specific set of units exists for a program. All input/output statements can refer to any unit that exists, while the INQUIRE, OPEN, and CLOSE statements can also refer to units that do not yet exist for the program.

Units that are preconnected for use by a program can be accessed in READ or WRITE statements without prior execution of an OPEN statement. These FORTRAN logical units coincide with the system logical units available to the particular user (in the user's session environment, or SST). System logical unit numbers are normally limited to 0 through 63. Unit numbers 64 through 255 can be accessed by changing Z$CSWD; see the FORTRAN 77 installation guide. The user can add these preconnected devices by using system commands dependent on the assigned user capability level (such as the SL command). For more detailed information on the system-dependent commands and tables, see the appropriate system reference manual.

A unit cannot be connected to more than one file at the same time; conversely, a file cannot be connected to more than one unit at a time. If a unit is disconnected in a program by a CLOSE statement, the unit number is available for reconnection to the same file or for connection to a different file in the program. Similarly, a particular file that is disconnected by a CLOSE statement can be reconnected to the same unit number or to a different unit number.

Note that the only way to refer to a disconnected file in an OPEN or INQUIRE statement is by name; therefore, if a scratch file is disconnected, it cannot be reconnected nor can data be reclaimed (since scratch files are purged on CLOSE or program termination).

The following input/output and file positioning statements must reference a unit that is connected:

READ inputs data from a connected unit.

WRITE outputs data to a connected unit.

PRINT outputs data to the default system unit, which is preconnected.

BACKSPACE moves the file pointer of the connected file to the position immediately before the previous record.

ENDFILE writes an end-of-file record as the next record of the file.

REWIND moves the file pointer of the connected file to the initial point of the file.

The following file control statements can reference a file that is either connected or not connected:

OPEN connects an existing file to a unit, creates a file and connects it to a unit, or changes certain specifiers of a connection between a file and a unit.

CLOSE disconnects a unit from a file.

INQUIRE requests information about the properties of a particular named file or of the connection to a particular unit (inquire either by file name or by unit number).

The INQUIRE and CLOSE statements can refer to files that do not exist. All other input/output statements must refer to files that exist.

File Control Specifiers

File control specifiers are used with various file input/output statements or file positioning statements. Some of the specifiers can be used with any file manipulation statement, while others have meaning only in particular statements.

This section describes the file control specifiers allowed in each of the file input/output statements, and describes any restrictions on their position or occurrence.

There are no positional requirements for the specifiers if all the keywords appear in the input/output statement. If the keyword `UNIT=` is omitted, the unit number specifier must appear first in the control list. If the optional keyword `FMT=` is omitted, the format specifier must be the second item in the list, following the unit specifier without the optional keyword `UNIT=`.

Detailed syntax requirements of the file control statements are described in Chapter 3.

READ and WRITE Statements

The following file control specifiers have meaning in the READ and WRITE statements:

`UNIT`	=	*unit*
or `UNIT`	=	*unit:sec*[*:ter*]
`FMT`	=	*fmt*
`REC`	=	*rn*
`IOSTAT`	=	*ios*
`ERR`	=	*label*
`END`	=	*endlabel*
`ZBUF`	=	*ibuf*
`ZLEN`	=	*ilen*

Examples	**Notes**

`READ(1,33) a,b,c`
`READ(UNIT=1,FMT=33) a,b,c`
	These two lines have the same effect; they both request input from logical unit 1, which is controlled by the format statement labeled 33.

`WRITE(61,11,REC=irec,IOSTAT=ios,ERR=99)`	This example specifies an output request to a direct access unit 61, writing record number `irec` as specified in the format statement labeled 11. If an error occurs, control transfers to the statement labeled 99 and the error code is stored in the variable `ios`.

The specification list must include exactly one unit specifier and at most one each of the other specifiers. If a `REC=` specifier appears, the statement is a direct access request. On a direct access request, the `END=` specifier must not appear, since the end-of-file record is not considered a part of a direct access file. The `END=` specifier is never allowed in a WRITE request. Also, the `FMT=` specifier must not indicate list-directed formatting (*) with a direct access file.

The `UNIT=`*unit:sec*[*:ter*] optional format permits secondary and tertiary addressing to be passed to the I/O driver. ZBUF and ZLEN also permit a second buffer to be passed to the I/O driver.

When ZBUF and ZLEN are used, the z control bit is set in the input/output request. See the appropriate I/O driver manual for details.

The HP-IB driver cannot distinguish between a zero secondary address and a missing secondary address. To ensure that the driver understands that you really do want to send a secondary address of 0, use 400B instead; this sets bit 8 in the address field. Because secondary addressing uses only the low 5 bits, the driver will then properly use secondary address 0. If you prefer, you may set bit 8 on any secondary address.

OPEN Statement

The following file control specifiers have meaning in the OPEN statement:

UNIT	= *unit*
IOSTAT	= *ios*
ERR	= *label*
FILE	= *name*
STATUS	= *sta*
USE	= *use*
ACCESS	= *acc*
FORM	= *fm*
RECL	= *rcl*
BLANK	= *blnk*
MAXREC	= *mrec*
NODE	= *node*
BUFSIZ	= *bufs*

The OPEN statement is used to connect a unit number to a file or to change certain specifiers of a connection between a file and a unit. When a file is opened, the file pointer is positioned at the beginning of the file. A redundant OPEN does not affect the current position of the file.

For a detailed description of the syntax and meaning of each of the control specifiers, see Chapter 3.

Examples

- Connect FORTRAN logical unit number 41 to a file called DAT. If an error is encountered in the OPEN, save the error code in the variable ios and transfer to the statement labeled 99.

  ```
  OPEN(41,IOSTAT=ios,ERR=99,FILE='DAT')
  ```

  (If DAT exists, it is connected to unit number 41. If DAT does not exist, it is created as a type 4 sequential access file.)

- Create a file called FIL1 in directory GEORGE, on node number 4, in the DS network. Connect the new file to FORTRAN unit number 88. Handle any OPEN errors the same as in the previous example.

  ```
  OPEN(88,FILE='/GEORGE/MYDATA.DAT>4',STATUS='NEW',IOSTAT=ios,ERR=99)
  ```

The specification list must contain exactly one unit specifier and at most one each of the other specifiers.

The FILE=*name* specification must be present when a named disk file is to be created or opened. The file name may contain a directory name or cartridge reference number, network node, security code, file type, or file size. (See the appropriate system reference manual for more detailed information on file names.) A file name containing the above values is sometimes called a *namr.*

The STATUS=*sta* specifier determines whether or not the file must already exist. If OLD is specified, the file must already exist. If NEW is specified, an error occurs if the file exists. The NEW specification directs that the file is to be created. If SCRATCH is specified, a file name must not be specified; a file with a unique name is created and USE is forced to EXCLUSIVE. A system-wide maximum of 99 scratch files can be open simultaneously (one user can have a maximum of 128 files open at one time, up to 99 of which can be scratch files). The UNKNOWN specifier can be supplied if you do not know whether a file or device exists; if no STATUS=*sta* specifier is given, the status defaults to UNKNOWN.

All the specifiers except the unit specifier are optional. If the file is opened with ACCESS=DIRECT, you must include the RECL specifier to declare the record length of the file.

A file can be connected to a unit number by an OPEN statement in any program unit of an executable program. Once connected to a unit, a file can be referenced in any program unit.

The OPEN statement can be used to connect existing files to a unit number, create and connect a named or scratch file, or change the control specifiers on a file that is already connected to a unit by referencing the same file name or omitting it and specifying different characteristics to the file. This is effectively the same as executing a CLOSE statement on the file that was previously connected to the unit number referenced in the OPEN statement, except that the file position is not changed.

Conversely, once a file has been connected to a unit number, that file cannot be connected to a different number until the file is closed.

The specifiers that have default values if omitted from the control list are:

```
STATUS = 'UNKNOWN'       FORM = 'FORMATTED' if ACCESS = 'SEQUENTIAL'
   USE = 'EXCLUSIVE'     FORM = 'UNFORMATTED' if ACCESS = 'DIRECT'
ACCESS = 'SEQUENTIAL'
 BLANK = 'NULL'
```

Additional Examples

- Open a scratch file for direct access, with a user-defined record length of 80 characters, assuming defaults for the remaining parameters.

```
OPEN(77,ACCESS='DIRECT',RECL=80,STATUS='SCRATCH')
```

- Connect a file named OUT1 to logical unit number 1. The file OUT1 exists on cartridge reference number 25 as a sequential file for formatted input/output. Specify that all blanks should be treated as zeros.

```
OPEN(1,FILE='OUT1::25',STATUS='OLD',BLANK='ZERO')
```

If logical unit number 1 is preconnected to the user's terminal, all references to unit 1 refer to the file OUT1. If a CLOSE statement is executed on unit 1, the terminal is again preconnected and accessible to the program as unit 1, without requiring an OPEN statement.

- Connect a direct access file named DATABASE in directory JOHN, sized to 4 blocks, with a record size of 80 words.

```
OPEN(888,FILE='/JOHN/DATABASE::::4',ACCESS='DIRECT',RECL=80)
```

- Open two files for sequential access on the local node, specifying one with a 2-block buffer and the other with the number of buffers remaining in the user program area of the current partition. The $FILES directive is:

```
$FILES 2,3,1,FREESPACE
```

The statements to accomplish this are:

```
OPEN(110,FILE='IN1',BUFSIZ=2)
OPEN(111,FILE='OUT1',BUFSIZ=NFIOB())
```

NFIOB is a no-argument integer function that returns the number of 128-word buffer blocks available in your program area. To increase the number of buffers available, use the SZ system command. SZ increases the size of the unused program area in the partition.

- Connect session LU 6 on node 20 to FORTRAN LU 6 (FMGR file system only).

```
OPEN(6,FILE='6:20')
```

- Connect a direct access scratch file with a maximum size of 10000 records to FORTRAN LU 880.

```
OPEN(880,STATUS='SCRATCH',ACCESS='DIRECT',MAXREC=10000,RECL = 10)
```

Since the default extent size for files is 24 blocks, using MAXREC can speed up execution. If MAXREC is not used, accessing records outside the first 24 blocks can cause a directory search.

CLOSE Statement

The following file control specifiers have meaning in the CLOSE statement:

`UNIT`	=	*unit*
`IOSTAT`	=	*ios*
`ERR`	=	*label*
`STATUS`	=	*sta*

The CLOSE statement is used to terminate the connection of a unit to a file. For a detailed description of the syntax and meaning of each of the control specifiers, see Chapter 3.

Examples	**Notes**
`CLOSE(55,IOSTAT=ios,ERR=99,STATUS='DELETE')`	Disconnects the file that was connected to unit 55 and specifies that the file should be deleted. If an error occurs, control transfers to the statement labeled 99 and the error code is stored in the variable `ios`.

If a CLOSE statement referencing a preconnected device is executed, there is no effect on subsequent input/output statements referencing the unit number (that is, the unit number and device remain connected).

Specifying `STATUS='KEEP'` causes the file to continue to exist. Specifying `STATUS='DELETE'` causes the file to be purged from the disk.

A file whose status is `SCRATCH` is always deleted by the system when the file is closed or at normal program termination, even if a CLOSE statement is executed specifying `STATUS='KEEP'`. For named files, if the `STATUS=` specifier is omitted, the default specification is `KEEP`. If a file is opened for nonexclusive use and is open to another program at the time of a CLOSE with `STATUS='DELETE'`, an error is reported.

Scratch files are not deleted if the user aborts the program (with the OF command) before normal termination. Scratch files that are kept are named .FT*nn*, where *nn* is an integer between 01 and 99.

A CLOSE statement can be executed referencing a unit that does not exist, and no action is taken.

If a file is not closed by a CLOSE statement in the execution of a program, the file is closed automatically upon program termination.

INQUIRE Statement

The following file control specifiers have meaning in the INQUIRE statement:

```
       UNIT = unit
       FILE = name
     IOSTAT = ios
        ERR = label
      EXIST = ex
     OPENED = od
     NUMBER = num
      NAMED = nmd
       NAME = fn
        USE = use
     ACCESS = acc
 SEQUENTIAL = seq
     DIRECT = dir
       FORM = fm
  FORMATTED = fmt
UNFORMATTED = unf
       RECL = rcl
    NEXTREC = nr
      BLANK = blnk
     MAXREC = mrec
       NODE = node
```

The INQUIRE statement requests properties of a file or device by either specifying the unit number, or in the case of a named file, by specifying the file name in the control list. The INQUIRE statement returns information on a file that is not connected to a unit, as well as on a connected file or device.

The INQUIRE-by-file version of the INQUIRE statement references exactly one file name specifier and any of the other specifiers except the unit specifier. The INQUIRE-by-unit version references exactly one unit specifier and one each of the other optional specifiers as desired, excluding the file name specifier.

Refer to Chapter 3 for a description of each of the control specifiers.

Table 5-1 lists all of the possible combinations for each specification in the INQUIRE statement.

A file is considered to exist (EXIST=.TRUE.) only if it can be opened to the inquiring program in shared mode (USE=NONEXCLUSIVE). The file may actually be opened momentarily to determine its properties; this could interfere with a separate program that simultaneously attempts to open the file for exclusive access.

A file is considered open (OPENED= .TRUE.) only if it is open to the inquiring program (through a FORTRAN OPEN statement).

The system routine LUTRU is used for local LU numbers only to verify existence. When a system LU is returned as a name, it is returned in ASCII, in a form legal for input to OPEN.

For sequential files, MAXREC is set to the number of blocks in the file, as if RECL was 256 (bytes).

Table 5-1. INQUIRE Statement Specifications

File	Unopened Unit	Unopened File	Nonexistent File	Opened File
EXIST	By LUTRU	True	False	True
OPENED	False	False	False	True
NAMED	True	True	True	False if a scratch file. Otherwise, true.
NUMBER	FTN unit	Undefined	Undefined	FTN unit
NAME	FTN unit	FILE name	FILE name	FILE name (if named) or system unit.
NODE	−1	−1	−1	*
ACCESS	Sequential	By type	Sequential	*
SEQUENTIAL	Yes	By type	Yes	* As
DIRECT	No	By type	No	*
FORM	Formatted	By type	Formatted	* given in
FORMATTED	Yes	By type	Yes	*
UNFOMATTED	No	By type	No	* OPEN
USE	Undefined	Undefined	Undefined	*
BLANK	Null	Null	Null	*
RECL	Undefined	In bytes (if direct).	Undefined	In bytes (if direct).
NEXTREC	Undefined	Undefined	Undefined	Record number (if file).
MAXREC	Undefined	Size/recl	Undefined	Size/recl (if file).

Example

```
 INQUIRE (FILE='EXFL',IOSTAT=ios,ERR=99
+ EXIST= ex,OPENED=iop,NUMBER=num
+ USE=use,ACCESS=acc)
```

Notes

Requests information on specified properties of the file names EXFL. If EXFL exists and is connected to a unit in the program, the variables `ex` and `iop` return the value true, the unit number is stored in `num`, and the character variables `use` and `acc` are defined. If EXFL does not exist, `ex` and `iop` return the value false, and the other specifiers are not defined.

In general, upon execution of an INQUIRE-by-file statement, if the file name is illegal or if the file does not exist, the specifiers *nmd*, *fn*, *seq*, *dir*, *fmt* and *unf* are undefined. If the file exists and is connected to a unit, *ex* and *od* return true, *num* become defined, and the variables *acc*, *fm*, *rcl*, *nr*, and *blnk* become defined if they are included in the INQUIRE statement.

Upon execution of an INQUIRE-by-unit statement, if the unit exists and is connected to a file, all specifiers become defined. If the unit is preconnected, all specifiers are defined except *use*, *rel*, *hr*, and *mrec*.

The specifiers `EXIST=`*ex* and `OPENED=`*od* always become defined with a true or false value if no error condition is encountered.

File Positioning Statements

The BACKSPACE, REWIND, and ENDFILE statements are used to control the position of the file pointer within a file. The following specifiers have meaning in these statements:

```
  UNIT = unit
IOSTAT = ios
   ERR = label
```

Exactly one external unit specifier must be included in the control list of the file positioning statements. The unit specified should be connected for sequential access.

The BACKSPACE statement causes the file pointer to be positioned before the preceding record. Backspacing over records written using list-directed formatting produces unpredictable results, because it is difficult to predict the number of records written.

The REWIND statement causes the file pointer to be positioned at the initial point of the file. The BACKSPACE and REWIND statements are allowed for all files, sequential and direct. To preserve program portability, these statements should not be used on files connected for direct access.

The ENDFILE statement writes an end-of-file record as the next record of the file. Note that the ENDFILE statement is not allowed on files connected for direct access.

Examples	**Notes**
`BACKSPACE 66`	Moves the file pointer of unit 66 to the previous record.
`REWIND (58,IOSTAT=ios,ERR=99)`	Moves the file pointer to the initial point in the file connected to logical unit 58. If an error occurs, control transfers to statement 99 and the error code is stored in the variable `ios`.
`ENDFILE 58`	Writes an end-of-file record as the next record of the file connected to unit number 58.

Note: If the second and third examples above appeared in sequence in a program unit, the effect would be to delete the information in the file.

Internal Files

Internal files provide a means of memory-to-memory data transfer. An internal file can be a character variable, a character array element, a character substring, or a character array. Each variable, substring, or array element is considered to be one record.

An internal file is accessed by a sequential formatted input/output statement. The name of the internal file appearing as the value of the UNIT parameter identifies the file. For example, the WRITE statement:

```
   WRITE(UNIT=ADDRESS,'(I10)') street_address
```

writes the value of the variable `street_address` into the first 10 positions of the internal file ADDRESS. (ADDRESS must be of type CHARACTER.) If ADDRESS is longer than 10 characters, the rest of the record is filled with blanks.

Example

```
   CHARACTER buffer*20
   READ(10,'(A)') buffer                 ! Enter input into buffer.
   READ(buffer,'(I10)',ERR=99)value      ! Check if buffer is an integer.
     SUM=sum+value                       ! Buffer was integer and stored
         :                               ! in value.
99 IF (buffer .EQ. 'end') THEN...        ! Buffer was not integer.
```

Internal files are not the same as disk files and do not have any internal record structure except the one-to-one correspondence between records and array elements.

The operation “advance to next record,” such as is caused by a slash in a format statement, causes processing of the internal file to advance to the next array element. If there is no next element, an EOF error occurs. Note that an internal file that is a variable, substring, or array element has exactly one record. When an internal file is an array element, other elements of the array are ignored.

Internal files should be used in place of the DECODE and ENCODE statements of FORTRAN 4X, because DECODE and ENCODE are not part of the ANSI 77 standard.

Another example of writing to a character variable is shown in the following program. This program assigns the character string (`10x,5(I3,x)`) to the variable `ifmt,` and then uses `ifmt` as a format in a WRITE statement.

Example

```
   PROGRAM inl
   CHARACTER *14 ifmt
   INTEGER iarray (5)
   DATA iarray/1,2,3,4,5/
   n=10
   m=5
   WRITE (ifmt,10) n,m
10 FORMAT ('('I2,'x,'I1,'(I3,x))')
   WRITE (6,ifmt) iarray
   END
```

Standard Input/Output Units

The user's terminal, normally unit 1, is the standard input unit. Unit 6 is the standard output unit. These units are initially preconnected. Preconnected files need not be opened prior to use in any input/output statement.

These standard unit numbers can be used to refer to a disk file in a user program if desired, and thus are not reserved unit numbers. You must use an OPEN statement to reassign these standard unit numbers to other files.

To change the standard input/output units from their default values, a call to the routine FSYSU can be made in the following form:

`CALL FSYSU(`*input_lu*`,`*output_lu*`)`

Example

```
CALL FSYSU(1,1)
```

See "PRINT Statement" and "READ from the Standard Input Unit Statement" in Chapter 3 for information on referencing standard unit numbers.

General File Examples

The following examples demonstrate several options of the file manipulation statements.

Example 1

The following program computes the mean of all the data items in the disk file DATA located on directory /JW/STAT. The file contains an unknown number of records, and each record contains one real number.

```
FTN77,L
$FILES 0,1                                      !Area for 1 disk file
       PROGRAM jwfil
       sum = 0.0                                !Initialize
       n = 0
       OPEN (333, IOSTAT = ios, ERR = 99, FILE = '/JW/STAT/DATA'
    + ACCESS='SEQUENTIAL',STATUS='OLD')

       DO WHILE (.TRUE.)
         READ (333, 22, END = 88, IOSTAT = ios, ERR = 99) anum
    22   FORMAT(F10.5)
         sum = sum + anum                        !Add data entries
         n = n + 1                               !Count entries
         END DO                                  !Loop

C  Out of loop

      88 WRITE(1,33) sum/n           !Output to preconnected terminal
      33 FORMAT ('The average is ', F12.6)
         CLOSE (333)
         STOP

C  If there is an output error in the OPEN or READ,
C  output to a preconnected terminal.

      99 WRITE(1,*) 'Error encountered=',ios
         END
```

Example 2

The following example inserts a single-number data entry in the proper position in a sorted sequential file. A direct access scratch file is used to store the temporary results prior to rewriting the original data file.

```
$FILES 0,2
       PROGRAM jwex

       nrec=0
C
C  Connect 555 to data file and 666 to scratch file C
C
       OPEN(555,FILE='JWDT1',STATUS='UNKNOWN',IOSTAT=ios,ERR=99)

OPEN(666,STATUS='SCRATCH',ACCESS='DIRECT',IOSTAT=ios1,ERR=99,RECL=80)
C
       READ *,anum                           !Enter number to insert.
       DO WHILE (.TRUE.)
         READ(555,*,END=999,IOSTAT=ios,ERR=99)fnum !Begin reading.
         nrec=nrec+1
C
         IF(anum.LE.fnum) THEN                 !Found the place.
           WRITE(666,*)anum                    !Enter the number.
           DO WHILE (.TRUE.)
            WRITE(666,*)fnum                   !Enter the file item.
            READ(555,*,END=1111,IOSTAT=ios,ERR=99)fnum
            nrec=nrec+1                        !Copy remainder of
C                                              !file to scratch file.
           END DO
         ELSE                                  !Not the place.
           WRITE(666,*)fnum                    !Copy item to scratch
file.
         END IF
       END DO                                  !Read next number.
C
C  Execute 999 if empty file or if item goes at end of file.
C
   999 WRITE(666,*)anum
C
C  Now copy data from scratch file to data file.
C
  1111 REWIND 555                              !Data file sequential.
       DO i=1,nrec
         READ(666,44,REC=I)fnum                !Scratch file direct.
    44   FORMAT(F16.6)                         !Read record I.
         WRITE(555,*)fnum
       END DO
       CLOSE(555)
       CLOSE(666)
       STOP 'All done.'
C
C  Handle error that occurred: ios from open or read of data file;
C  ios1 from open of scratch file.
C
    99 WRITE(1,*) "IOS=",ios," IOSI=",iosl
       END
```

6

Procedures and Block Data Subprograms

Procedures are self-contained computational and data units that must be activated by the main program or another procedure. FORTRAN procedures provide a way to organize a program into small, manageable pieces, each of which performs a well-defined task (such as solving a mathematical problem, performing a sort, or outputting standard headings) and provide initial values for variables and array elements in labeled common blocks.

Block data subprograms are used to initialize variables in labeled common blocks. They cannot contain executable statements.

Procedures can be grouped into subroutines and functions:

Subroutines
Subroutine subprograms
Functions
Function subprograms Statement functions Intrinsic functions

Subroutine and function subprograms can be written in languages other than FORTRAN, and can come from the system library. For more information, refer to “Interfacing FORTRAN with Non-FORTRAN Subprograms” in Chapter 7 and “Input/Output Library Interface Functions” in Appendix B.

Subroutines and functions are referenced differently and return values differently.

Subroutine Subprograms

The subroutine subprogram is a program unit that has a SUBROUTINE statement (refer to Chapter 3 for syntax) as its first statement. Subroutine subprograms are user-written procedures that perform a computation or a subtask for another program unit. Values can be passed to the subroutine and returned to the calling program unit by arguments or common blocks (see "Using the Command Statement" below).

Examples of SUBROUTINE statements are:

```
SUBROUTINE next(arg1,arg2)

SUBROUTINE last(a,*,*,b,i,k,*)

SUBROUTINE noarg
```

In the above examples, the formal arguments `arg1`, `arg2`, `a`, `b`, `i`, and `k` pass values to the specified subroutine subprograms. The second example shows an alternate return form, using asterisks (described in "Alternate Returns from a Subroutine" below).

A subroutine subprogram can contain any statement except a BLOCK DATA, FUNCTION, or PROGRAM statement.

The last line of a subroutine subprogram must be an END statement. One or more RETURN statements can be included to return control to the calling program unit. If no RETURN statement is included in the subroutine, the subroutine END statement returns control to the calling program unit.

Referencing a Subroutine

A subroutine is executed when a CALL statement (see Chapter 3 for syntax) is encountered in a program unit. Examples of CALL statements are:

```
CALL next (x,y,z)

CALL last (a,*10,*20,b,i,k,*30)

CALL noarg
```

When the subroutine is executed, the actual arguments (`x`, `y`, `z`, `a`, `b`, `i`, and `k` above) in the CALL statement are associated with their equivalent formal arguments. The subroutine is then executed using the actual argument values. When a RETURN statement is executed in the subroutine, control normally returns to the statement following the CALL statement in the calling program unit. The second example above shows the form of alternate return specifiers in the CALL statement (discussed in "Alternate Returns from a Subroutine" below).

Alternate Returns from a Subroutine

Normally, control returns from a subroutine to the calling program unit at the statement following the CALL statement. The alternate return statement allows return to the calling program unit at any labeled executable statement.

The RETURN statement specifies an alternate return with an integer expression (which can be an integer constant) that identifies a statement label number in the CALL statement. The SUBROUTINE statement must contain one or more asterisks corresponding to alternate return labels in the CALL statement. (See “CALL Statement” and “RETURN Statement” in Chapter 3 for the syntax of calls with alternate return statements.) Here is an example of a CALL and its associated SUBROUTINE and alternate return statements:

```
CALL sub (a,*10,*20*,*30)
  ⋮
SUBROUTINE sub (a,*,*,*)
  ⋮
RETURN n
```

Control returns to statement 10, 20, or 30 depending on whether n evaluates to 1, 2, or 3.

If the RETURN statement contains an expression, the value of that expression should not exceed the number of asterisks in the SUBROUTINE statement; also, the number of asterisks in the SUBROUTINE should equal the number of alternate return labels specified in the CALL statement. Upon execution of a RETURN statement with an expression whose value is either less than 1 or greater than the number of alternate return labels in the CALL statement, control returns to the statement following the CALL statement.

Note If the CALL statement specifies, for example, three return statements, but the SUBROUTINE statement contains only two asterisks, the RETURN statement is effective for the values 1, 2, and 3 (even though 3 is greater than the number of asterisks in the subroutine statement), with all other values causing a “normal” return to the statement following the CALL statement. (A compiler warning is generated if RETURN 3 appears in the source file, but this does not prevent loading and executing of the program.)

Example

```
PROGRAM main
 ⋮
CALL sort (a,*10,b,*20,*30)
 ⋮
END
SUBROUTINE sort (a,b,*,*)
 ⋮
RETURN 3
END
```

Notes

If the compiler warning is ignored, loading and executing this program causes a return to statement 30.

If the RETURN statement contains a variable or variable expression (but not a constant), FORTRAN 77 permits all alternate returns specified in the CALL statement to be represented in the SUBROUTINE statement by a single asterisk. (For ease of understanding and portability, however, it is recommended that the SUBROUTINE statement always contain as many asterisks as the number of alternate returns specified in the CALL statement.)

Here is an example that uses alternate returns. Its subroutine searches a file (PARTS) to validate a part number. Each record in PARTS contains two integers; the first is the part number and the second is a code. A negative code indicates an obsolete part number. All existing part numbers are in this file. The records in the file are in ascending part number order. For simplicity, the part number search is sequential. This subroutine uses the normal return if the part number is found and is not obsolete. If the part number is obsolete, the first alternate return is taken. If the part number is not found, the second alternate return is taken.

```
$FILES 0,1
      PROGRAM prog
        ⋮
C  Get a part number
        ⋮
      CALL validate(part_number,*100,*999)

C  Normal return.  Process for valid part number follows.
        ⋮
C  First alternate return.  Process for obsolete part number follows.

  100   ⋮ ...

C  Second alternate return.  Process for invalid part number follows.

  999   ⋮ ...

      END

      SUBROUTINE validate(part_number,*,*)
      INTEGER part_number, rec_part, rec_code
      LOGICAL obsolete_flag,part_found_flag

C  Initialize variables

      obsolete_flag = .FALSE.
      part_found_flag = .FALSE.

C  Search for part number and set flags accordingly

      OPEN(111,FILE='PARTS')
      DO WHILE (.NOT. part_found_flag)
        READ(111,*, END=99) rec_part, rec_code
        IF (rec_part .EQ.  part_number) THEN
          part_found_flag = .TRUE.
          IF (rec_code .LT. 0) obsolete_flag = .TRUE.
        ENDIF
      END DO
   99 CLOSE(111)

C  Return to calling program depending on flags

      IF (obsolete_flag) RETURN 1
      IF (.NOT.  part_found_flag) RETURN 2
      RETURN
      END
```

Functions

A function can be intrinsic (see "Intrinsic Functions" below) or defined in a user-written function subprogram. When a function reference in an expression is executed, the specified function is evaluated and a value is returned. As with a subroutine, a function can return values through its arguments or common block. However, this practice should be avoided, because it is difficult to follow changes to the values of the arguments.

When a function is executed, the function name is associated with a value the same way a variable is. When the function exits, the value returned is the last value assigned to the function name.

Function Subprograms

A function subprogram is a user-written FORTRAN function included in a FORTRAN 77 program. A function subprogram is a program unit that has a FUNCTION statement (see Chapter 3 for syntax) as its first statement. Examples of FUNCTION statements are:

```
FUNCTION time()

INTEGER*4 FUNCTION add(k,j)

LOGICAL key_search(char_string,key)
```

Values are passed to function subprograms by arguments (`k` and `j`, `char_string` and `key` in the above examples) or common blocks (see the "Using the COMMON Statement" below). Note that an argument list is not required, but parentheses are required to differentiate the function name from a simple variable.

A function subprogram can contain any statement except another FUNCTION statement or a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

Since the function returns a value, it must have a type. The type can be associated with the function in one of three ways:

- If the type is given as the first part of the FUNCTION statement, that type is assigned to the function. If the type is specified in the FUNCTION statement, the function name must not appear in a type statement. A name must not be explicitly typed more than once in a program unit.
- If the type is not given in the FUNCTION statement, the function name can be included in a type statement within the function subprogram. (A type statement is the only nonexecutable statement in which a function name can appear. See "Type Statement" in Chapter 3.)
- If the function name is not included in a type statement and the type is not given in the FUNCTION statement itself, the type is assigned implicitly according to the first letter of the function's name.

The type associated with the function name in each referencing program unit must agree with the type of the function according to the above methods.

In a character function reference, the length of the character function must be an integer constant expression of the same length as the referenced function. Note that there is always agreement of length if the function has a length of (*).

To associate a value with the function subprogram name, use the name within the function subprogram in one or more of the following ways:

- On the left side of an assignment statement
- As an element of an input list in a READ statement
- As an actual argument of a function or subroutine subprogram reference, where the subprogram defines the value

Examples | **Notes**

```
INTEGER FUNCTION fact(n)
fact=1
DO i = 2,n
   fact = fact*i
END DO
RETURN
END
```

The function name is associated with a value by appearing on the left side of an assignment statement.

```
FUNCTION tot(num,sum)
REAL num
IF (num .GE.  0) THEN
   tot = sum + num
ELSE
   READ (10,*) tot ENDIF
   RETURN
   END
```

The function name is associated with a value in one of two ways: by appearing on the left side of an assignment statement or by appearing in the input list of a READ statement.

```
FUNCTION next(back,fwrd)
IF (back .GT. 1.5) THEN
   CALL gtfwrd(next)
ELSE
   CALL gtback(next)
ENDIF
RETURN
END
```

The function name is associated with a value in one of two subroutine subprograms. Within the subroutines, `next` must be assigned a value.

Here is an example of a character function. It returns the uppercase equivalents of any lowercase characters in a string.

```
CHARACTER*(*) FUNCTION upshift(string)
CHARACTER*(*) string
DO i = 1,LEN(string)
   IF (string(i:i) .GE. 'a' .AND. string(i:i) .LE. 'z') THEN
      upshift(i:i) = CHAR (ICHAR (string(i:i) ) - 40B)
   ELSE
      upshift(i:i) = string(i:i)
   ENDIF
END DO
RETURN
END
```

A function returns the value last assigned to the function name at the time a RETURN statement is executed within the subprogram.

The last line of a function subprogram must be an END statement. One or more RETURN statements can be included to return control to the calling program unit. If no RETURN statement is included in the subprogram, the END statement returns control to the calling program unit.

Alternate returns are not allowed in function subprograms. A function subprogram always returns to the expression from which it was invoked.

An example of a calling program unit and a function subprogram follows. This program asks for input of two numbers, m and n, and computes the combinations of m items taken n at a time. That is, it computes:

$$\frac{m!}{n!\ (m-n)!}$$

In this example the function subprogram fact is invoked in the expression fact(m)/(fact(n) * fact(m-n)).

Here is the source program listing followed by two typical runs:

```
$CDS ON
       PROGRAM main
       INTEGER*4 fact,result
       WRITE (1,*) 'm and n?_'
       READ (1,*) m,n
       result = fact(m)/(fact(n)*fact(m-n))
       WRITE (1,'(I5," things taken",I5," at a time =",I8)')m,n,result
       END

       INTEGER*4 FUNCTION fact(num)
       IF (num .EQ. 1) THEN
         fact = 1
       ELSE
         fact = num + fact(num-1)
       ENDIF
       RETURN
       END
```

```
m and n?7,4

    7 things taken 4 at a time = 35

m and n?10,2

    10 things taken 2 at a time = 45
```

This example uses recursion, which is available only with CDS ON.

Values over 12 cause `fact` to exceed the range of double integers. You can extend the range by using real or double precision variables.

Statement Functions

A statement function is a user-defined, single-statement computation that applies only to the program unit that defines it. Its form is similar to that of an arithmetic, logical, or character assignment statement. Only one value is derived from a statement function. Examples:

```
root(a,b,c) = (-b + SQRT(b*b - 4.*a*c))/(2.*a)

disp(c,r,h) = c*3.1416*r*r*h

indexq(a,j)=IFIX(a) + j - ic
```

The statement function is referenced by using its symbolic name with an actual argument list in an arithmetic, logical, or character expression. A statement function can be referenced only in the program unit that contains it.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow any specification statements used in the program unit.

A variable or array with the same name as a statement function must not appear in the same program unit.

As an extension to the ANSI 77 standard, the symbolic name of a statement function can be an actual argument. Also, statement functions can appear in an EXTERNAL statement.

All arguments in the formal argument list are simple variables that assume the values of actual arguments in the same program unit when the function is invoked. These variables are completely distinct from variable, array, function, subroutine, or common block names used in the program unit or in other statement functions (that is, they are local to the statement function). Variables used in the statement function and not included in the argument list assume the current value of the variable in the program unit (for example, the third example above, in which `ic` is not an argument, just an ordinary variable, defined outside the statement function, but used in it).

For syntax and more information about statement functions, see "Statement Function Statement" in Chapter 3.

Intrinsic Functions

An intrinsic function is a function provided by FORTRAN 77 that is available to any program. Intrinsic functions perform such operations as converting a value from one type to another. Intrinsic functions also perform basic mathematical functions such as finding sines, cosines, and square roots of numbers. The intrinsic functions available to FORTRAN 77 are listed in Tables B-1 through B-7 of Appendix B. The tables give 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.

Generic Names

Generic names simplify referencing of intrinsic functions by allowing the same name to be used with more than one type of argument.

The highest level of generic function name allows the use of one name with distinct types of arguments. For example, in the function ABS(*var*), *var* can be a single or double integer, or a two- or four-word real, complex, or double complex constant, variable, or expression.

A second, lower, level of generic function name allows the use of one function name for any precision of a particular type of argument. For example, in the expression IABS(*ivar*), *ivar* is either a single or double integer constant, variable, or expression.

The type of the generic function result is determined by the type of its arguments. An IMPLICIT statement or type declaration does not change the type of an intrinsic function. If a particular function, such as MOD, requires more than one argument, then all arguments in that function must be of the same general type. For example, single integer arguments can be mixed with double integer arguments, but integer arguments cannot be mixed with real.

If a formal argument has the same name as a generic or specific function, appears as a formal argument, that name identifies the argument, not the intrinsic function, in that program unit or statement function.

Example	**Notes**

`SUBROUTINE x(log,f)`
`:`
`f = log(f)`
`:`
`END`
	In this context, `log` is not an intrinsic function.

Referencing a Function

A function is executable when it is referenced in an expression.

Syntax

name ([*parm1*,*parm2*, . . .])

where:

name is the name of the function.

parm is an actual argument.

Examples	**Notes**
`a = z + root(a,b,c)`	`root` is a user-defined function (defined in a function subprogram) that uses the values of `a`, `b`, and `c` to compute a value for `root`.
`s = SIN(6.5)`	`SIN` is an intrinsic function that computes the sine of 6.5.
`b = time( )`	`time` is a user-defined function that has no arguments.

A function reference returns a specific value of the type associated with the function and is equivalent to using a variable reference of the same type. In the above examples, a real number is returned as the value of `SIN(6.5)`. When a function reference is encountered during evaluation of an expression, control passes to the referenced function. The function is executed using the actual arguments listed in the function reference. The function name is assigned a value that is passed back to the referencing expression. The referencing expression continues its evaluation, using the passed value where the function reference appeared.

The length of a character function in a character function reference must be the same as the length of the character function in the referenced function. A function length of (*) matches all references.

A function that does input/output must not be referenced in the input/output list of a READ, WRITE, or PRINT statement.

Procedure Communication

Values are passed between a calling program and a procedure in an argument list. In addition, values can be passed through common blocks to and from subprograms.

Using Arguments

The arguments passed by the calling program are called actual arguments. The procedure, which is structured with formal arguments, uses the actual arguments passed to it to replace the formal arguments and perform the computation. For example, when the call is made to the function subprogram from the following calling program unit:

```
a=6.5
b=8.3
r=rfunc(a,b)*3.14159
```

a and b are passed to the subprogram. The subprogram could be the following:

```
FUNCTION rfunc(c,d)
rfunc = (c*d) + (d**3)
RETURN
END
```

Variables used as formal arguments (such as c and d in the above example) are said to be *passed by reference*. This means that they refer to the storage locations of the actual arguments (a and b in the above example) and therefore assume the current values of the actual arguments. Changing the values of the formal arguments passed by reference changes the actual arguments in the calling program unit.

Actual arguments in a subroutine call or function reference should agree in number, order, and type with the corresponding formal arguments. An actual argument must be a variable (simple or subscripted), array name, substring, procedure name, constant, or expression. The expression can be a character expression. However, it cannot involve concatenation of an operand whose length specification is (*), unless that operand is the symbolic name of a constant. The actual arguments for statement functions must be variables, constants, or expressions.

When a procedure name is used as an actual argument, it does not pass a value as other actual arguments do. Instead, it passes a pointer to the subprogram to the referenced subroutine or function. A subprogram name used as an actual argument must appear in an EXTERNAL statement. An intrinsic function name used as an actual argument must appear in an INTRINSIC statement.

When an actual argument is an expression, the formal argument in the subprogram unit that corresponds to the actual expression should not be altered in the subprogram. The concept of call by value, as defined in Pascal and other languages, does not exist in FORTRAN 77.

When an EMA variable is passed, the corresponding formal argument must be declared to be in EMA (as described in "EMA Statement" in Chapter 3). If the formal argument is not in EMA, the actual argument must not be an EMA variable. If you need to pass an EMA variable to a subprogram that does not expect EMA, use the variable in an expression. You can do this by enclosing the EMA variable in parentheses, for example. This produces an addressing level change that causes the two-word EMA address to be replaced by a one-word address pointing to a temporary copy of the value of the EMA argument. You should not alter the value of the formal argument corresponding to the parenthesized EMA argument within the subprogram.

Formal arguments of a statement function can consist of simple variable names only. Formal arguments of a subprogram, on the other hand, can consist of names that represent variables, arrays, or procedures. The argument of a subroutine subprogram can also be an alternate return specifier (described in "Alternate Returns from a Subroutine" above).

Although formal arguments must agree with actual arguments by type, they may have a different form. An array may have different dimensions, and a character variable may have a different length. No conversion or rearrangement is performed; the formal argument merely specifies a different way to access memory. To use this feature, you should have a thorough understanding of the way arrays are accessed in memory.

Examples	**Notes**
`func(a,b,c) = a*b/c`	Formal arguments within a statement function can only be simple variables.
`FUNCTION nect (z,i,j)`	`z` is a simple variable of type real, `i` is a double
`DOUBLE PRECISION*8 i`	precision variable, and `j` is a 10-element integer array.
`DIMENSION j(10)`	
`SUBROUTINE add(a,f,get)`	a and `f` are real variables and `get` is a function name.
`a = get(f)`	

Examples of Argument Correspondence

```
CALL sub1(a,x,i,b(1),fcn)
   ⋮
SUBROUTINE sub1(array,r,in1,tmp,f)
```

Notes

a is an array name. The formal argument `array` must be dimensioned in the subprogram. If `x` is a real variable; the formal parameter in the second position of the subroutine argument list (`r`) must also be a real variable. `b(1)` is an element of array `b` and can correspond to a single variable name (not dimensioned) or an array (dimensioned) in the formal argument list (`tmp`). `fcn` is a function name; `f`, therefore, must be used in the context of a function in the subprogram.

Example of Array Passing

```
      PROGRAM main
      DIMENSION x (10,10)
         ⋮
      CALL colz (x(1,5))
         ⋮
      END
      SUBROUTINE colz (colx)
      DIMENSION colx (10)
         ⋮
      DO 10 i=1,10
   10 colx (i)=0.0
      END
```

Notes

The main program unit dimensions an array `x` having 10 rows and 10 columns. One element of `x` appears in the argument list of the reference to subroutine `colz`; this is the element in the fifth column of the first row of `x`. `colz` dimensions the formal argument `colx` to have 10 elements, thus corresponding to the entire fifth column of the actual array `x`. `colx` then sets each element of the fifth column of `x` to 0, and returns.

Here is an example of a character argument:

```
FUNCTION size(string)
CHARACTER*10 string
```

All variable names are local to the program unit that defined them. Similarly, formal arguments are local to the subprogram unit or statement function containing them. Thus, they can be the same as names appearing elsewhere in another program unit.

No element of a formal argument list can occur in a COMMON statement (except as a common block name), EQUIVALENCE statement, or DATA statement. When an array name is used as a formal argument, the formal argument array name must be dimensioned in a DIMENSION or type statement within the body of the subprogram.

If the actual argument is a constant, a symbolic name of a constant, a function reference, an expression involving operators, or an expression enclosed in parentheses, the associated formal argument must not be redefined within the subprogram.

Using the COMMON Statement

A common block can be used to pass values between a calling program unit and a subprogram. (See "COMMON Statement" in Chapter 3 for more information on using common blocks.) The example below shows how common blocks can be used to pass values to and from subprograms.

Example

```
PROGRAM comex
COMMON a,b,side
READ *,a,b
CALL tri
PRINT *, side
END

SUBROUTINE tri
COMMON x,y,side
side = SQRT(x**2 + y**2)
RETURN
END
```

Notes

The variable a in the main program shares storage space with x in the subroutine. When a value for a is determined by the READ statement, x automatically shares this value. Similarly, b and y also share storage space, as does the variable side in the main program and the subprogram. The subroutine uses the values input for a and b to compute the length of the hypotenuse of a right triangle.

Arrays in Subprograms

Since only the name of an array appears in the formal argument list of a subprogram, an array declarator must appear in a DIMENSION or type statement for that array name. The number and size of dimensions of an actual argument array should match those in the calling subprogram. The number and size can differ, but when the array is accessed it will have the properties of the new declaration.

You should understand subscripting when using different declarations in the calling routine and the subroutine. The size of the formal argument array must not exceed the size of the actual argument array. Since array bounds are not checked at run-time, no warning is issued if the formal array size exceeds the actual array size. Altering these unreserved locations could yield unpredictable results.

Normally, array bounds are specified by integer constants and the bounds are fixed by the values of these constants. You can, however, use adjustable array declarators in subprograms. With adjustable array declarators, one or more of the array bounds are specified by an expression involving integer variables instead of integer constants. An example of an adjustable array declarator is shown below.

Example

```
    PROGRAM ardim
    DIMENSION iarr(10,10)
    i = 10
    j = 10
    CALL sb(iarr,i,j)
    PRINT *,iarr
    END
    SUBROUTINE sb(ivar,k,m)
    DIMENSION ivar(k,m)
    DO 10 nr=1,k
    DO 10 nc=1,m
    ivar(nr,nc) = nr*nc
 10 CONTINUE
    RETURN
    END
```

Notes

The example declares an array iarr in the main program. iarr has two dimensions of 10 elements each. A subroutine (sb) is called to fill iarr with values. The variables i and j are set equal to the array bounds, and these variables are used as the actual arguments to be passed to the subroutine. The subroutine formal arguments k and m assume the values passed to them through i and j. These variables are used in a DIMENSION statement to establish the bounds for array ivar.

The last upper bound of a formal argument array is not used; FORTRAN 77 allows this bound to be an asterisk. A declarator of this type is called an assumed-size array declarator. If the last subscript exceeds the last bound of the actual argument, the results are unpredictable. The following example demonstrates the use of assumed-size array declarators.

Example

```
PROGRAM main
   ⋮
DIMENSION a(10,10)
CALL sub1(a)
   ⋮
END
SUBROUTINE sub1(z)
DIMENSION z(10,*)
   ⋮
INTEGER num(5,10,0:6)
j = 5
i = func1(num,j)
   ⋮
END
FUNCTION func1(arry,k)
INTEGER arry(k,10,0:*)
   ⋮
END
```

Notes

The last subscript of the array z can take any value from 1 to 10. The last upper bound of the array num can take any value from 0 to 6.

A variable used to dimension a formal argument in a bounds expression in a subprogram must appear in a common block or be a formal argument.

EMA arrays can be variably dimensioned. It is very important that the bounds (variable dimensions) accurately describe the range of the subscripts. If the bounds indicate that the subscripts will fit in 16-bit (single) integers, more efficient code will be generated.

For this purpose, a *single integer bound* is a single integer variable, or expression, or a constant which fits in a single integer. The value "123J" is a single integer bound. A *double integer bound* is a double integer variable, or expression, or a constant which does not fit in a single integer. An asterisk bound is a single integer under the I compiler option or a double integer under the J option.

If an EMA array is dimensioned with single integer bounds, the subscripts must be in the range [-32768 to +32767] (because the .IMAP instruction is generated). If any bounds are double integers, then double integer subscripts can be used (because the .JMAP instruction is generated). Large EMA arrays may *not* be declared using an asterisk under the I option because the subscripts are converted to single integers for use with .IMAP. The same restriction applies to arrays dimensioned with a last upper bound equal to 1, a convention used in older programs in lieu of an asterisk.

Adjustable and assumed-size array declarators cannot be used in COMMON statements.

Character Arguments

If a formal argument is of type character, the associated actual argument must be of type character and the length of the formal argument must be less than or equal to the length of the actual argument. If the length of a formal argument of type character is less than the length of an associated actual argument, then the characters associated with the formal argument are the leftmost number of characters of the actual argument up to the length of the formal argument. For example, if an actual character argument is a variable assigned the value `abcdefgh` and the length of the formal argument is 4, then the characters `abcd` are associated with the formal argument. If the formal argument is changed, only the first part of the actual argument is changed. If the last part of the actual argument was undefined, it remains undefined.

If a formal argument of type character is an array name, the restriction on length is for the entire array and not for each array element. The length of an individual array element in the formal argument array can be different from the length of an array element in an associated actual argument array, array element, or array element substring, but the formal argument array must not extend beyond the end of the associated actual argument array. Association of array elements will be changed in a manner similar to changing array dimensions, as shown in the following example:

```
CHARACTER*3 x(5)
CHARACTER*1 y(15)
EQUIVALENCE (x,y)
     :
CALL SUB (x)
     :
SUBROUTINE sub (z)
CHARACTER*1 z (15)
```

The formal argument `z` is referenced exactly like the variable `y` in the calling program.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, its length is the sum of the lengths of the operands.

The length of a formal argument can be declared by an asterisk, as in this example:

```
SUBROUTINE sub(char_dummy)
CHARACTER*(*) char_dummy
```

When the length is declared by an asterisk, the formal argument assumes the length of the associated actual argument for each reference of the subroutine or function. If the associated actual argument is an array name, the length assumed by the formal argument is the length of an array element in the associated actual argument array.

SAVE Statement

A SAVE statement causes the definition status of an entity to be retained after execution of a RETURN or END statement in a subprogram. It is used to save the values of entities from one call of a subprogram to the next. Within a function or subroutine subprogram, an entity specified by a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, such an entity in a common block can become undefined or redefined by another unit.

Syntax:

```
SAVE [a1 [,a2 [,...an] ] ]
```

where:

a is a named common block (preceded and followed by a slash), a variable name, or an array name. Each item can appear only once.

Formal argument names, procedure names, and names of entities in a common block cannot appear in a SAVE statement.

A SAVE statement without a list is treated as though it contains the names of all allowable items in that program unit.

The SAVE statement is unnecessary in a main program, except when it is required for consistency with subprogram common blocks, as explained below.

A common block name surrounded by slashes in a SAVE statement specifies all the entities in the block.

If a particular common block name is specified in a SAVE statement in one subprogram of an executable program, it must be specified in a SAVE statement in each suprogram where the common block appears, including the main program.

If a named common block is specified in the main program unit, the current values of the common block storage sequence are made available to each subprogram that specifies that named common block. In this case, a SAVE statement in a subprogram has no effect.

Execution of a RETURN or END statement in a subprogram causes all entities within the subprogram to become undefined, except for:

- Those specified by SAVE statements
- Those in a blank common block
- Those in a named common block that appears in the subprogram and also in at least one other program unit that directly or indirectly references that subprogram

A SAVE statement is not always necessary on the HP 1000, but should be used whenever values must be saved, for readability and portability. On the RTE-A Operating System, SAVE statements are often needed in CDS programs. On RTE-6, SAVE statements are often needed in MLS programs.

The linker may require special commands when linking a program that contains a SAVE statement. Refer to the appropriate linker reference manual.

ENTRY Statement

An ENTRY statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which it appears. It can appear anywhere within a function subprogram after the FUNCTION statement or within a subroutine subprogram after the SUBROUTINE statement, with this exception: an ENTRY statement cannot appear within a block IF statement or a DO loop.

A subprogram can have one or more ENTRY statements. An ENTRY statement is a nonexecutable statement. For the syntax of the ENTRY statement, refer to "ENTRY Statement" in Chapter 3.

Referencing an External Procedure by Entry Name

An entry name in an ENTRY statement in a function subprogram can be referenced as a function. An entry name in an ENTRY statement in a subroutine subprogram can be referenced as a subroutine.

When an entry name is used to reference a procedure, execution of the procedure begins with the first executable statement that follows the ENTRY statement with that entry name.

An entry name can be referenced by any program unit of an executable program, except the program unit that contains that entry name in an ENTRY statement.

The order, number, type, and names of the formal arguments in an ENTRY statement can differ from those of the formal arguments in the FUNCTION or SUBROUTINE statement and of other ENTRY statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the formal argument list in the corresponding FUNCTION, SUBROUTINE, or ENTRY statement. (Using an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.)

Entry Association

Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable 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 also to become defined and all associated variables of different type to become undefined. Such variables need not be of the same type (unless the type is character), but a variable that references the function must be defined at the time 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.

ENTRY Statement Restrictions

Within a subprogram, a name cannot appear both as an entry name in an ENTRY statement and as a formal argument in a FUNCTION, SUBROUTINE, or ENTRY statement. It cannot appear in an EXTERNAL statement.

In a function subprogram, a variable name that is the same as an entry name cannot appear in any statement that precedes the appearance of the entry name in an ENTRY statement (except in a type statement).

If an entry name in a function subprogram is of type character, each entry name and the name of the function subprogram must also be of type character. If the name of the function subprogram or any entry in the subprogram is declared to be of length asterisk (*), all such entities must also be declared to be of length (*). In all other cases, all such entities must have a length specification of equal integer value.

If a formal argument appears in an executable statement, the statement can be executed only during a call to the function that defines the formal argument. That is, if a formal argument is present in one argument list but not another, it may be used only when the subroutine or function is entered through the appropriate entry. Note that the association of formal arguments with actual arguments is not retained between references to a function or subroutine.

Block Data Subprograms

Block data subprograms are used to initialize the variables declared in common blocks.

The BLOCK DATA statement, described in Chapter 3, 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. Each different named common block within a block data subprogram produces a separate subprogram module, which has the common block name.

In general, only specification statements and data initialization statements are allowed in the body of a block data subprogram. Allowable statements include COMMON, DIMENSION, type statements (INTEGER*4, REAL*8, etc.), DATA, SAVE, PARAMETER, and IMPLICIT statements.

Block data subprograms are optional in FORTRAN 77, but are required in FORTRAN 4X. Labeled common can be set up the same way as in FORTRAN 4X by using the NOALLOCATE option of the $ALIAS compiler directive. See Chapter 7 for a description of the NOALLOCATE option.

Example

```
BLOCK DATA null
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
```

Notes

null 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.

As an extension to the ANSI 77 standard, variables in labeled common blocks can also be initialized in other program units.

7

Using FORTRAN 77

This chapter contains information about FORTRAN 77 operations in an RTE operating system, including the capabilities and invocation procedures of the compiler, error messages to the user that can arise during compiler operations, a sample listing of a FORTRAN program, a discussion of the compiler directives, information on reducing the size of a loaded program, and information on interfacing FORTRAN with other languages.

FORTRAN 77 is a problem-oriented programming language that is translated by a compiler into relocatable object code. Source programs are accepted from either a disk file or a device. Error messages, list output, and relocatable object code are stored in files or output to devices. The object code produced by the compiler can be loaded by an RTE linker and then executed.

FORTRAN Control Statement

The FORTRAN control statement is an optional directive specifying compiler options. If a control statement is used, it must be the first line in the source file.

Syntax

FTN[*xx*],*p1*,*p2*,*p3*,*p4*,*p5*,*p6*,*p7*,*p8*,*p9*,...,*pn*

where:

xx is 66 or 77. Make this specification when you wish to use the nondefault mode. (See the note below.)

p1–pn are compiler options in any order, chosen from the options in Table 7-1.

Note ANSI 66 mode or ANSI 77 mode is chosen as the default when the compiler is loaded into the system. The FORTRAN control statement can override this default. If the control statement begins with FTN66, the compiler operates in ANSI 66 mode. If it begins with FTN77, the compiler operates in ANSI 77 mode. If the control statement begins with FTN or is omitted, the compiler operates in the default mode set when the compiler itself is linked. A complete discussion of ANSI 66 and ANSI 77 modes is found in Chapter 8.

If the program does not begin with a control statement, the compiler assumes FTN,L.

Table 7-1. FORTRAN 77 Compiler Options

Option	Meaning
L	Selects list output. A listing of the source language program is output to the list file.
Q	Includes the relocatable address of each statement on a listing. Each line of the listing becomes six characters longer. If the Q option is specified, the L option is implied. The Q option is useful for debugging.
T	Selects a symbol table listing. A symbol table for each program unit is output to the list device. The M option also produces a symbol table listing.
M	Selects a mixed listing. A listing of both the source and object program is produced. Each source line is included with the object code generated in the compilation process. The M option implies the T option.
C	Selects a cross-reference table. A cross-reference table of symbols and labels used in the source program is produced. See Appendix F.
P	Enable warning about possible overlap in character assignments. See Error 106 in Appendix A.
S	Symbolic Debug. Causes information about symbols, data types, and line numbers to be included in the relocatable output file. This information may be passed to Debug/1000 for use in debugging the code. See the *Symbolic Debug/1000 User's Manual*, part number 92860-90001. Note that the S option results in a larger relocatable file. However, the extra records can be removed later using the appropriate option in the MERGE utility.
B	This option is a legal option but is ignored when included in a FORTRAN 77 control statement. Specifying any option in the command line, such as B, cancels all options stated in the FORTRAN control statement, except the I, J, X, Y, and E options.
I	Integers are stored in one word (default). The J suffix can be useful for passing double integer constants to routines that expect double integers.
J	Integers are stored in two words. The default setting can be changed to two words when the compiler is linked (see the installation guide for further information). Care must be taken to pass the correct size of integers between subprograms. In particular, programs compiled with the J option pass constants as two-word integers, even to system routines that expect one-word integers. The I suffix can be useful for integer constants passed to routines that expect one-word constants, such as system routines.
Y	Double precision is four words (default).
X	Double precision is three words. The default setting can be changed to three words when the compiler is linked (see the installation guide for further information). See Chapter 8 for more information on three-word real variables.

Table 7-1. FORTRAN 77 Compiler Options (continued)

Option	Meaning
E	Selects EMA transparency mode, which causes all subprogram arguments to be passed using 32-bit addresses. Therefore, subroutines do not have to distinguish between EMA and non-EMA arguments. Non-EMA arguments are made into EMA arguments by constructing an EMA address with the first word equal to -32768 and the second word containing the non-EMA address. A subprogram can be called at one time with EMA arguments and at other times with non-EMA arguments, as long as the main program and all subprograms are compiled under the E option.
1 - 9	Uses a different error handler. Errors are normally handled by calling .EXIT or .NFEX. This option causes the routine ERR*n* to be called, where *n* is a single-digit option number from 1 through 9. The ERR*n* routine is supplied by the user. The ERR*n* routine must use the direct, or .ENTN, calling sequence. For example, if the routine is ERR3, place the directive "$ALIAS ERR3,DIRECT" just before the definition of ERR3. Note that option 9 is reserved for use by HP system software.

Compiler Invocation

Syntax

[RU,]FTN7X, *source_input* [*list_output* [*binary_output* [*line_count* [*options* [*directive*]]]]]

where:

source_input is the name of a disk file or the logical unit number of the device containing the FORTRAN source code.

Under the CI file system, if the file does not exist and the file name does not have an extension, an extension of ".FTN" is assumed. A source file name must not begin with an ampersand and end with ".FTN".

If an interactive device is specified, the compiler prints a right bracket (]) on the device as a prompt. It then accepts input one line at a time and continues to issue prompts until a control-D is entered.

list_output is one of the following:

- – (minus sign)
- file name
- logical unit number
- null (omitted)

If a minus sign is specified, the list file name is derived from the source file name. If the source file is a CI file with an extension of ".FTN", the list file name is formed by changing ".FTN" to ".LST". If the source file is an FMGR file and its name begins with an ampersand (&), the list file name is formed by changing the ampersand to an apostrophe ('). If the source file is of neither form, an error results.

The list file is created if it does not exist. If it already exists, it must be named with a “`.LST`” extension or a leading apostrophe.

If an LU (logical unit) number is specified, the listed output is directed to that logical device.

If the *list_output* parameter is omitted, the user's terminal is assumed.

binary_output is one of the following:

- – (minus sign)
- file name (*namr*)
- logical unit number H
- null (omitted)

If a minus sign is specified, the binary file name is derived from the source file name. If the source file is a CI file with an extension of “`.FTN`”, the binary file name is formed by changing “`FTN`” to “`.REL`”. If the source file is an FMGR file and its name begins with an ampersand (&), the binary file name is formed by changing the ampersand to a percent sign (%). If the source file is of neither form, an error results.

The binary file is created if it does not exist. If it already exists, it must be named with a “`.REL`” extension or a leading percent sign and have a file type of 5 (signifying relocatable object code).

If an LU (logical unit) number is specified, the binary output is directed to that logical device.

If this parameter is omitted, no binary relocatable code is generated.

line_count is a decimal number that defines the number of lines per page for the list device or file.

Specification of this parameter is optional. If it is omitted, 59 lines per page are assumed. If a number less than 10 is specified, the compiler runs material together over page boundaries, treating the page size as if it is infinite. (The default of 59 can be changed by changing the value of Z$LPP; see the installation guide.)

options is up to six characters that select control function options. No commas are allowed within the options. All options allowed in the FORTRAN 77 control statement are also legal options in the run command, except I, J, X, Y, and E. These options override the options declared in the FORTRAN 77 control statement (except I, J, X, Y, and E).

The B option has no effect in the run command, beyond overriding (and thus negating) any options in the FORTRAN 77 control statement (except I, J, X, Y, and E, which cannot be overridden).

directive is a compiler directive beginning with $. Most directives must be quoted (for example, with back quotes) so that embedded spaces and commas are part of the command argument. The $INCLUDE directive is not allowed. The directive is listed as line zero of the program. It is processed after any FTN control statement.

If any parameters are specified, but some are skipped, commas must be used as placeholders. Otherwise, blanks or commas may be used to separate parameters in the CI file system; commas are required in the FMGR file system.

If the L, Q, T, M, and C options are all off, the list output has no headings and the listing consists of the summary lines only, as shown:

Module ....
FTN7X ....
.
. (Two blank lines between each module.)

Module ....
FTN7X ....

(See "Sample Listing" on page 7-7.)

Examples

```
:RU,FTN7X PROGA - -
```

Runs FORTRAN 77 to compile the source file PROGA.FTN. Listed output is directed to list file PROGA.LST, and binary relocatable code is directed to binary file PROGA.REL. The number of lines per list file page defaults to 59.

```
:RU,FTN7X MYFILE.FTN TEMP.LST
```

Runs FORTRAN 77 to compile the source file MYFILE.FTN. Listed output is directed to the file TEMP.LST. No binary relocatable code is generated. The number of lines per list file page defaults to 59.

```
:RU,FTN7X &ABCD
```

Runs FORTRAN 77 to compile the source file &ABCD. Listed output defaults to the user's terminal. No binary relocatable code is generated. The number of lines per list file page defaults to 59.

```
:RU,FTN7X &AAAA - - 80
```

Runs FORTRAN 77 to compile the source file &AAAA. Listed output is directed to list file 'AAAA. Binary relocatable code is directed to the binary file %AAAA. The number of lines per list file page is 80 (appropriate for eight lines per inch on 11-inch paper).

```
:RU,FTN7X TRANSACT - -,,MS
```

Runs FORTRAN 77 to compile the source file TRANSACT.FTN. Listed output is directed to the list file TRANSACT.LST. Binary relocatable code is directed to the binary file TRANSACT.REL. The number of lines per list file page defaults to 59. A mixed listing and a symbol table will be produced, and symbolic debug information will be included in the binary file.

```
:RU,FTN7X TRANSACT.FTN /LIST/- /REL/-,,MS
```

This command line has the same results as the previous example, except that the listing is put in directory /LIST and the binary file in directory /REL.

```
:RU,FTN7X,&JW1,-,-,,B
```

This command line schedules FORTRAN 77 to compile the source file &JW1. The commas are required for commands used under FMGR. List output is directed to list file 'JW1. Binary relocatable code is directed to the binary file %JW1. The number of lines per list file page defaults to 59. The command line cancels all the options specified in the FORTRAN 77 control statement by specifying the B option, which is otherwise ignored. Any option in this position overrides all options in the FORTRAN 77 control statement except I, J, X, Y, and E. Use B to override control statement options without specifying any other command line option.

```
:RU,FTN7X, &JW1,-,-,,B,'$SET(System='RTE-A')'
```

Refer to the previous example. In addition, execute a $SET directive following the (optional) FTN control line, but before any other directives or statements.

Compiler Messages

At the end of the compilation (when the compiler detects the end of source condition), the following message is printed:

```
END ftn7x: nn disasters nn errors nn warnings
```

where:

nn is the number of occurrences of each problem type. "`No`" is printed if there are no occurrences of that type.

All error messages are output to the list file or device, unless there is an error in the list file specification itself. This generates one of two error messages:

- If the user incorrectly specified the list file, the following message appears on the user's terminal:

  ```
  /FTN7X: Access failed on list
  ```

- If the user incorrectly specified both the source and list files, the following message appears on the user's terminal:

  ```
  /FTN7X: Access failed on list and source
  ```

If the listing is sent to LU 0 (the null device) and errors are detected, the errors and explanations are output to the terminal.

Compiler Status Values

The FORTRAN compiler returns five integer status values using "PRTN". These values are returned in the CI variables "RETURN1" through "RETURN5", or via RMPAR if the compiler is run programmatically.

The five values returned are:

RETURN1:	Total number of errors (sum of RETURN2+RETURN3+RETURN4 below).
RETURN2:	Number of disasters.
RETURN3:	Number of errors.
RETURN4:	Number of warnings.
RETURN5:	Compiler revision in numeric form.

Sample Listing

Here is a program followed by the listing produced by FORTRAN 77. (The program contains errors.)

```
PROGRAM loop2
INTEGER odd_sum
odd_sum = 0
i = 1
DO WHILE (i .LT.  1000)
    odd-sum = odd_sum + 1
    i = i + 2
END DO
WRITE(1,'(I5)) odd_sum
END
```

```
Page 1                Opts: 77/LYI         Sat, Jun 27, 1992  8.43  PM
                                           &LOOP2.FTN::SCRATCH
     1     PROGRAM loop2
     2     INTEGER odd_sum
     3     odd_sum = 0
     4     i = 1
     5     DO WHILE (i .LT.  1000)
     6       odd-sum = odd_sum + 1

          od?
 "LOOP2"   error  10 detected at line 6 column 12

     7     i = i + 2
     8     END DO

 "LOOP2"   error  30 detected at line 8

     9     WRITE(1,'(I5)) odd_sum
```

```
          WRITE(1,'(I5))?

 "LOOP2"   error  28 detected at line 9 column 20

   10      END

Module LOOP2        3 errors     Data    4     Blank Common:   None
FTN7X 5000/861229  No warnings   Code:  32     Stack size:       10

Page 2  LOOP2        Opts: 77/LYI         Sat, Jun 27, 1992,   8:43 PM
                                          LOOP2.FTN::SCRATCH

  Error Directory

  Number  Explanation

    10    Unrecognized statement.
    28    Unexpected character or unexpected end of statement.
    30    Incorrect nesting.  May be due to other errors.
```

Notice that one error can generate other error messages. For example, if the compiler encounters an unrecognized statement within a DO loop, it loses the context of the loop and concludes that the END DO statement has no prior DO statement.

In the listing a two-line heading appears at the top of each page. This heading gives the page number, the options in effect, the date and time at which the listing was created, and the source file name.

The compiler options can be used to send additional information to the list file (see Table 7-1).

Each line of the source file is listed. The numbers in the leftmost column are the source file line numbers.

Each line containing errors is followed by an error line. Lines 6, 8, and 9 in the previous example contain errors. For a detailed explanation of error lines, see Appendix A.

The last two or three lines after the source code are a summary containing applicable information from the following list:

`Module`	The name of the module (main program, subroutine, function, or block data subprogram).
nn `errors`	The number of errors in the module.
nn `warnings`	The number of warnings in the module.
`FTN7X revision/date`	The version of the compiler.
`Program`	The number of words of code and data in a non-CDS module.
`Blank Common`	The number of words of blank common.
`Save`	The number of words of SAVE (excluding common).
`Local Ema`	The number of words of EMA (excluding common).
`Code`	The number of words of code in a CDS module.

`Data` The number of words of data in a CDS module.

`Stack Space` The number of words of stack space used by a CDS module.

`Program, Save, Blank Common, Data, Code,` and `Local Ema` list the amounts of relocation space required. FORTRAN 77 does not use the base page (although the linker and run-time library do). These values do not include labeled common.

At the end of the listing, the number of each error encountered during compilation is listed with a description of the error it represents.

Linking a Program

Once a program has compiled without errors, it can be linked on the system by the linker. For more information, refer to the appropriate system reference manual.

Running a Program

Once a program has been successfully linked, it can be executed by specifying the command:

pgm[*parameters*]

where:

pgm is the program name as specified in the PROGRAM statement. If no PROGRAM statement is used, *pgm* is `FTN`.

parameters are command line parameters that can be accessed by the library routines RCPAR and RHPAR and by using formal parameters in the PROGRAM statement.

Compiler Directives

The compiler directives that can be included in the FORTRAN 77 source file to control certain compiler functions include $ALIAS, $CDS, $EMA, $FILES, $INCLUDE, $LIST, $MSEG, $OPTPARMS, $PAGE, $TITLE, and $TRACE. These statements are explained in detail below. All compiler directives begin in column 1 of the source file.

$ALIAS Directive

The $ALIAS directive can be used to change certain default features of a subprogram, entry, or labeled common block. In the following discussion, *subprogram* refers to a subroutine or function that is to be called. *Entry* refers to the name of the current subroutine or function or a name used in an ENTRY statement.

Syntax

```
       {subpgm_name [='external_name'] [,WXTRN] [,DIRECT] [,NOABORT]
                    [,ERROREXIT] [,EMA] [,NOEMA]}
$ALIAS {entry_name [='external_name'] [,DIRECT]}
       {/common_blk_name/ [='external_name'] [,NOALLOCATE]}
       {/common_blk_name/ = value}
```

A common block name must be enclosed in slash marks. The external name must be delimited by single quotation marks. The options follow the external name (if specified). A comma must be used before the first option and between all subsequent options.

Examples

```
$ALIAS iopsy ='.OPSY', DIRECT

$ALIAS sub1, NOEMA,NOABORT

$ALIAS /time/ ='$TIME',NOALLOCATE

$ALIAS /mem/ = 0
```

The $ALIAS directive must precede the first use of the aliased name.

The common block, subprogram, or alternate entry name is used in the source code for that program. The external name (if specified) is the name that will be used in the code generated by the compiler. Any other program unit that references a common block, subprogram, or alternate entry name that has been defined with an external name must include a similar $ALIAS directive. This use of external names is primarily applicable to system routines and entry points with names that do not follow the FORTRAN symbolic name conventions.

Note that using quoted names in the $ALIAS directive is an exception to the rule that external names are output in the relocatable file with all lowercase letters converted to uppercase. Because quoted $ALIAS external names are used in the relocatable file verbatim, without converting to uppercase, most such names should be supplied in uppercase. Otherwise, they may not match instances of the names in other relocatable files.

$ALIAS should not be used for library routine names that are generated by the compiler. For example, the names EXEC or SIN should not be used. The compiler already knows about the unusual calling sequences of such routines.

The options specify nonstandard features of the common block, alternate entry, or subprogram. Each option is described in detail below.

WXTRN Option

The WXTRM option can be used with subprograms only. It indicates that the external name should be *weak external.* Weak externals are never found in library searches, and undefined weak externals do not cause linker errors. For more information about weak externals see the appropriate system reference manual.

DIRECT Option

The DIRECT option is used with entries and subprograms. It indicates that the subprogram or entry does not use the `.ENTR` calling sequence in the code generated by the compiler. `.ENTR` is not used in the code for an entry, and references to a subprogram do not contain an initial DEF to the return address. `.ENTR` is discussed in detail in the appropriate system library reference manual.

NOABORT Option

The NOABORT option can be used with subprograms only. For each alternate return specified in the subprogram reference, the compiler issues a jump to that return point immediately after the code generated by the compiler for the reference.

The following example shows how this option can be used with the routine RNRQ. (RNRQ is described in detail in the appropriate programmer's reference manual.) In this example, the call to RNRQ specifies the no-abort bit. Instead of specifying the error return in a GOTO statement following the call, an alternate return can be used when the NOABORT option is specified for the subprogram.

Example

```
$ALIAS rnrq, NOABORT
      no_abort_bit = 40000B
         ⋮
      CALL rnrq(icon+no_abort_bit,irn,istat,*999)
      normal return
         ⋮
  999 error return
```

Note Using GOTO after such calls is not supported in FORTRAN 77.

ERROREXIT Option

The ERROREXIT option can be used with subprograms only. It is similar to the NOABORT option. With ERROREXIT, the compiler generates code to handle an alternate return by calling the standard FORTRAN run-time error handler. The error handler displays an error message, closes any open FORTRAN files, and, in CDS programs, displays a traceback list showing the names of all active subprograms. The subprogram must have only one alternate return, and the CALL statement for the subprogram must *not* reference it explicitly (that is, it must not contain a statement number parameter). Using ERROREXIT is preferable to letting RTE abort the program, because FORTRAN files are closed and (if CDS) a traceback is displayed.

Example

```
     $ALIAS rnrq, ERROREXIT
           no_abort_bit = 40000B
                 :
           CALL rnrq(icon+no_abort_bit,irn,istat)
           normal return
           (error return handled automatically)
```

NOEMA Option

The NOEMA option can be used with subprograms only. NOEMA is used with EMA transparency (refer to “FORTRAN Control Statement” in Chapter 7). EMA transparency mode causes all arguments to be passed as 32-bit addresses. This mode can be overridden by specifying NOEMA for the desired subprogram. Arguments to this subprogram are then passed with their natural address sizes; that is, non-EMA variables are passed with 16-bit addresses and EMA variables are passed with 32-bit addresses.

The following example shows how the NOEMA option can be used with the subroutine ABREG (see the appropriate system library reference manual). The E option in the control statement indicates EMA transparency mode. ABREG expects a 16-bit address, so NOEMA must be indicated for ABREG.

Intrinsic functions of FORTRAN 77 (refer to Appendix B) do not need the NOEMA option specified in order to pass 16-bit addresses. The intrinsic functions of FORTRAN 77 include EXEC and REIO. For all other subroutines and functions that expect one-word addresses, the NOEMA option must be specified.

The NOEMA option must be specified at the beginning of any program unit that calls subroutines that expect 16-bit addresses.

Example

```
  FTN77,L,E
  $ALIAS ABREG, NOEMA
     :
     CALL ABREG(ia,ib)
```

Because formal parameters are treated as EMA variables under the E option, any formal parameter that is passed through to another subroutine is passed with a 32-bit address, even if that subroutine was declared NOEMA. In other words, an address may be expanded because of the E option or ALIAS...EMA, but it may not be reduced. (However, be aware of the effect of using extra parentheses, as described in the section “EMA Directive.”)

EMA Option

The EMA option can be used with subprograms only. EMA is used without EMA transparency (see “FORTRAN Control Statement” earlier in this chapter). Without EMA transparency mode, all arguments are passed as 16-bit addresses. This can be overridden by specifying EMA for the desired subprogram. Arguments to this subprogram are then passed as 32-bit addresses.

The following example shows how the EMA option can be used with the subroutine EMREG. EMREG expects a 32-bit address, so EMA must be indicated for EMREG.

The EMA option must be specified at the beginning of main programs and of subprograms or main programs calling subroutines that expect 32-bit addresses.

Example

```
FTN77, L
$ALIAS EMREG, EMA
   ⋮
   CALL EMREG (ia, ib)
```

NOALLOCATE Option

The NOALLOCATE option can be used with labeled common blocks only. When NOALLOCATE is specified for a labeled common block in a block data subprogram, the compiler creates a module (NAM,END) containing an entry point (ENT) that matches the common block name. When NOALLOCATE is specified in a main program, function, or subroutine, the compiler generates an external (EXT) reference matching the common block name. When NOALLOCATE is not specified for a labeled common block, the compiler creates an allocate record (ALLOC) for each labeled common block reference.

The NOALLOCATE option must be used for common blocks that are in the operating system. This includes SSGA and system labeled common.

NOALLOCATE must be specified for each non-EMA, non-SAVE labeled common block in a program that is to be used by SGMTR to create an MLS-LOC linker command file. Each labeled common block that is specified with the NOALLOCATE option must be defined in a block data subprogram. This requirement and others for using SGMTR are discussed in detail in the *RTE-6/VM Loader Reference Manual*, part number 92084-90008.

Absolute Common Blocks

When a value is specified after the "=" in the $ALIAS directive, the common block starts at that absolute address. This can be used to access free space, as in the following example:

```
$ALIAS /mem/ = 0
   PROGRAM free
   IMPLICIT INTEGER (a-z)
   COMMON /mem/ mem(0:0)
    ⋮
   CALL limem(dum,fwa,len)
   CALL sub(mem(fwa),len)
     ⋮
```

The first argument to sub begins at the first word of free space.

Note that variables in absolute common blocks cannot be initialized in DATA statements.

$CDS Directive

CDS (Code and Data Separation) is available exclusively under the RTE-A Operating System with the VC+ System Extension Package. The $CDS directive controls whether or not the program is compiled for CDS systems. Programs compiled for systems without CDS should not use the $CDS directive.

Syntax

```
      {ON}
$CDS {OFF}
```

The $CDS directive must be placed between modules or before the first module. (A module is a main program or a subprogram.) The option is left in the new state until the next $CDS directive. The initial (default) state is determined by the Z$CDS symbol used when the compiler was loaded onto the system. (The source file &FRPLS contains the entry points that set compiler default states at load time.)

When CDS is OFF, program instructions (code) and local variables (data) are both put in the program area (relocation space). When CDS is ON, instructions are put in the code area, and local variables are put in the data area.

Programs compiled with CDS ON can have up to 7.9M bytes of code. Overlaying (segmenting, in RTE-6/VM terms) is not needed (and cannot be done). CDS programs can have up to 62k bytes of data, which is expandable to 2M bytes with EMA and 128M bytes with VMA.

There are some restrictions on mixing CDS and non-CDS subprograms (for example, non-CDS modules cannot call CDS modules). Also, some programs that rely on local variables being statistically allocated (holding their value from one subprogram call to the next) may not work in CDS mode unless SAVE statements are added. See Appendix H, “CDS Usage,” for further explanation and some restrictions.

$CLIMIT Directive

In a CDS program, small constant strings are kept in data space and large constant strings are kept in code space (and copied to data space when needed).

The directive, $CLIMIT *n,* sets the minimum size for string constants in code space. When *n* is 32767, all string constants are in data space. Otherwise *n* may be in [1,100]. Very small values of *n* are extremely inefficient.

The default may be set by changing the Z$CLM value in &FRPLS, assembling &FRPLS, and relinking the compiler with the new %FRPLS.

$EMA Directive

EMA (Extended Memory Area) and VMA (Virtual Memory Area) provide a means for storing and manipulating large amounts of data. EMA and VMA are fully discussed in the appropriate programmer's reference manual.

The $EMA directive specifies that certain common blocks are to reside in VMA or EMA.

Syntax

```
$EMA /name1/,/name2/,...,/namen/
```

where:

name1...namen are names of common blocks to be put into VMA or EMA.

Blank common can be put into VMA or EMA by specifying "//".

VMA and EMA are memory access methods that allow quick referencing and manipulation of large amounts of data. The data can reside in physical memory (EMA) or virtual memory (VMA). VMA and EMA are declared the same way in the source program; the distinction between them is made during linking.

More than one $EMA directive can be used, but all must appear before the first FORTRAN statement in the program unit. All variables specified in the common block go into EMA.

Each program unit that declares EMA common must be preceded by an $EMA directive. Multiple $EMA directives are permitted in each program unit.

A VMA or EMA variable is referenced within a program unit like any other variable, except when being passed to other subroutines or functions. When calling subprograms that do not expect VMA/EMA arguments, the user should pass VMA/EMA variables by enclosing the variable in an extra layer of parentheses, for example, `F((x))`. This causes the compiler to generate code to put the argument into a local temporary variable and to pass the temporary variable as an argument to the subprogram. This works only if the subprogram does not modify the argument, because the function receives a pointer to the temporary variable, not to the original argument. The corresponding formal argument must not be modified.

EMA variables passed to intrinsic functions need not be enclosed in parentheses. A local copy of such variables is made automatically.

For subprograms expecting VMA or EMA variables, arguments can be passed normally. The subprogram must declare the corresponding formal arguments in an EMA statement. These arguments can be modified just like non-EMA arguments.

When the main program and all referenced subprograms are compiled with the E option (see Table 7-1), all arguments use long addresses and need not be declared in EMA statements.

An additional restriction on VMA or EMA variables is that they cannot be used as keyword values for results in input/output statements. For example, in `IOSTAT = j`, the variable `j` must not be in VMA or EMA. The restricted keywords are:

- IOSTAT
- All keywords in INQUIRE except FILE and UNIT

Also, when integer arrays are used instead of character variables for keyword values, those arrays must not be in VMA or EMA.

A VMA or EMA variable can be equivalenced the same as any other variable. The same restrictions apply. See "EQUIVALENCE Statement" in Chapter 3 for more information.

Variables in EMA may be initialized in DATA statements in the same way as non-EMA variables. However, this is only supported on RTE-A. If any EMA variables appear in DATA statements in a program, the RTE-A linker will change the program from an EMA program to a VMA program without warning. See the *RTE-A Programmer's Reference Manual*, part number 92077-90007, for more information on EMA and VMA programming. Initialized VMA is not supported on the RTE-6/VM Operating System.

The old form of the $EMA directive (for FORTRAN 4X and FORTRAN 4), whose syntax follows, is still accepted for compatibility, but should not be used in new programs:

```
   $EMA (common_block_name, mseg_size)
```

A comprehensive example including the VMA/EMA directive and the EMA statement follows:

```
$EMA /xyz/
      PROGRAM test
      COMMON /xyz/a(100,200),c(3000,80)
      EQUIVALENCE (a(99,100),b)
      DIMENSION e(200,330)
      EMA e
       ⋮
      b = SIN(a(j,k))

C     Change address level and pass to ufun.

      d = ufun((a(j,k)))
       ⋮
C     Pass subscripts for VMA/EMA arrays to subroutine add1.
C     Subroutine add1 has VMA/EMA arrays defined in named common.

      CALL add1 (j,k)
       ⋮

C     Pass VMA/EMA array e by reference with its dimensions
C     to subroutine add2.

      CALL add2(e,200,300,sum)
       ⋮
      END

      FUNCTION ufun(x)

C     Square the number.

      ufun = x * x
      RETURN
      END
```

```
$EMA/ /xyz/

     SUBROUTINE add1(m,n)

C    m and n are subscript arguments.

     COMMON /xyz/a(100,200),c(3000,80)

C    Increment an element in the VMA/EMA array a.

     a(m,n) = a(m,n) + 1
      ⋮
     RETURN
     END

     SUBROUTINE add2(eprime,me,ne,sum)

C    eprime is a VMA/EMA array passed by reference and sum is
C    non-VMA/EMA.  Note that subroutine add2 does not require a
C    $EMA directive or any VMA/EMA named common blocks.

     EMA eprime(me,ne)
      ⋮
     j = 1
     DO  i = 1,ne
      eprime(j,i) = eprime(j,i) + 2
  END DO
      ⋮
     RETURN
     END
```

Notes on the Preceding Example

Arrays a and c are in VMA/EMA common because they are in the block common named xyz, which is declared in the $EMA directive; e is in VMA/EMA because it is declared in an EMA statement; b is in VMA/EMA because it is equivalenced to a; and eprime is a formal argument declared by the EMA statement to be in VMA/EMA.

The call to SIN can use standard argument notation because SIN is an intrinsic function. The call to ufun must use the addressing level change technique because ufun's argument is not declared in an EMA statement. This change is indicated by enclosing its argument in an extra layer of parentheses as shown. An element in array a is incremented in the subroutine add1, which has declared the VMA/EMA common block. The array e is passed by reference to the subroutine add2, which has declared the formal argument, eprime, to be in VMA/EMA.

$FILES Directive

The $FILES directive must be specified in a FORTRAN main program to set aside a storage area describing any connection between a FORTRAN logical unit and a disk file or LU. Programs that use only the preconnected units (that is, that do not contain any OPEN statements), can be executed without the $FILES directive.

The $FILES directive can also be specified in a subprogram. It has an effect only if the program does not use disk files and the $FILES directive in the subprogram specifies 0 for the number of disk connections. This can cause a reduction in the size of the loaded program (see "Reducing the Size of a Loaded Program" below). If the program uses files, then a $FILES directive in a subprogram has no effect. (If a subprogram is compiled with a $FILES directive that does not match the $FILES directive in the main program, and the program terminates while executing the subprogram, files may be left open.)

Information about the $FILES directive also appears in Chapter 5.

Syntax

```
        {m,n[,s[,b]]}
$FILES  {m,n,s,FREESPACE}
        {m,n,DS}
```

where:

m is the maximum number of nondisk units that can be connected at one time (0 to 128, default = 0).

n is the maximum number of disk units that can be connected at one time (0 to 128, default = 0).

s is the default number of 128-word blocks in each Data Control Block (DCB) buffer for each file (1 to 128, default = 1). You can override this default for any given file through the BUFSIZ parameter in the OPEN statement. See Chapter 5.

b is the total number of 128-word buffer blocks that can be allocated (default = *n*s*).

`FREESPACE` indicates that the area at the end of the user partition is available for use as 128-word buffer blocks. The NFIOB function allows you to determine the number of input/output buffers available for allocation. The SZ system command tells you how much free space your program has. The SZ command in LINK lets you increase the free space in your program. See the examples in "Specifiers in the OPEN Statement" later in this chapter.

`DS` allows connections to remote nodes when using the FMGR file system. When the CI file system is used, remote connections are always allowed and DS must not be specified.

Example

```
$FILES 2,3
     PROGRAM filex
     :
```

Notes

The $FILES directive begins in column 1 and should appear before the first FORTRAN statement in the program unit. This directive reserves storage in the main program of two nondisk units and three disk units, using the default Data Control Block (DCB) buffer size of one block.

The *m* and *n* parameters are required positional parameters that indicate the maximum number of connections to be made simultaneously within the main program and all subprogram units. The default when no directive appears in the main program unit is:

```
$FILES 0,0
```

The size of the table area in the main program can be approximated by:

$$3*(m + n) + n*(32 + 128*s)$$

or, if `FREESPACE` is used, by:

$$3*(m + n) + 16*n$$

or, if `DS` is used, by:

$$3*(m + n) + 20*n$$

$IF Directive

The $IF directive conditionally compiles blocks of source code.

Syntax

```
$IF (condition)
```

where:

condition is a logical expression for conditional compilation.

The $IF directive can appear anywhere in,the source code. The condition is a constant expression evaluating to type logical. All identifiers in the expression must be defined in previous $SET directives.

Directives used with $IF are $ELSE, $ELSEIF, $ENDIF, and $SET. The $IF directive has optional $ELSEIF blocks, 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, $ELSEIF, or $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 multiple $ELSEIFs corresponding to each $IF. There can be, at most, one $ELSE corresponding to each $IF.

The identifiers in $SET and $IF compiler directives are not 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.

$IFDEF and $IFNDEF Directives

The $IFDEF and $IFNDEF directives are similar to $IF, except that the condition tested is the existence (or nonexistence) of a $SET variable.

Syntax

```
$IFDEF (name)
```

or

```
$IFNDEF (name)
```

Subsequent lines are compiled if *name* has (for $IFDEF) or has not (for $IFNDEF) been given a value in a preceding $SET directive. The value of *name* itself is ignored.

$ELSE Directive

The $ELSE directive is used with the $IF directive.

Syntax

```
$ELSE
```

The $ELSE directive semantically parallels the ELSE statement. The source following the $ELSE line is compiled only if the condition in the matching $IF directive and in all previous matching $ELSEIF directives is *false*.

$ELSEIF Directive

The $ELSEIF directive is used with the $IF directive.

Syntax

`$ELSEIF` *(condition)*

or

`$ELSE IF` *(condition)*

The $ELSEIF directive semantically parallels the ELSEIF statement. There may be multiple $ELSEIF directives between a $IF directive and the matching $ELSE or $ENDIF.

If the condition in the matching $IF directive and all previous matching $ELSEIF directives is *false*, and this condition is *true*, the text after this $ELSEIF directive is compiled. Otherwise the text between this directive and the next matching $ELSEIF, $ELSE, or $ENDIF is treated as a comment.

$ENDIF Directive

The $ENDIF directive terminates the $IF directive.

Syntax

```
$ENDIF
```

Each $IF directive requires a $ENDIF, and vice versa. It may appear after the last FORTRAN END statement.

$INCLUDE Directive

The $INCLUDE directive causes the compiler to include and process subsequent source statements from a specified file or LU. When EOF is read from this file or LU, the compiler continues processing with the line following the $INCLUDE directive.

Syntax

```
                     {[,LIST]}
$INCLUDE file_name {[,NOLIST]}
```

where:

file_name is either a disk file name or an LU number. *file_name* may be quoted or unquoted.

`LIST` causes the included lines to be listed. `LIST` is the default.

`NOLIST` causes the included lines not to be listed.

$INCLUDE directives cannot be continued.

An include file can contain an $INCLUDE directive; that is, $INCLUDE can be nested nonrecursively.

Examples

```
$INCLUDE specs

$INCLUDE '../includes/graphics:inc',NOLIST
```

The LIST status after an $INCLUDE is completed is restored to the status just before the $INCLUDE was encountered, even if LIST or NOLIST is specified or the $LIST directive appears in the included file.

Line numbering within the listing of an included file begins with 1. The line numbers are suffixed with a "+" (for example, 153+). The original line numbering of the source resumes after the included file listing.

If an interactive LU is specified, information is included interactively from that LU, in which case the prompt listed at the LU is "+".

$INCLUDE can also be used as a statement. The INCLUDE statement begins in or after column 7. (See "INCLUDE Statement" in Chapter 3.)

$LIST Directive

The $LIST OFF directive suppresses listing of individual source lines and their corresponding mixed listings (if any). $LIST ON turns the listing back on. If the L, Q, and M compiler options are off, $LIST has no effect.

Syntax

```
      {ON}
$LIST {OFF}
```

Error messages are always listed, even when $LIST OFF is specified.

When the M option is specified, the mixed listing generated after the END statement is always listed, regardless of the options of the $LIST directive.

The $LIST directive has no effect on the accumulation of cross-references for the C option.

$MSEG Directive

The $MSEG directive specifies the optional size in pages of the RTE MSEG. The concept of an RTE MSEG is discussed in the appropriate programmer's reference manual. FORTRAN 77 programs usually use the default size. (That is, they do not use the $MSEG directive.) Certain products (for example, the Vector Instruction Set) require specification of the MSEG size.

Syntax

```
$MSEG n
```

where:

n is the number of pages of RTE MSEG requested.

$MSEG 0 requests as many pages of MSEG as possible.

Example

```
$EMA /larg/
$MSEG 2
        PROGRAM main
        COMMON/larg/lgary(250000),x,y
```

In the preceding example, the $EMA directive declares that a common block name `larg` resides in EMA.

The $MSEG directive declares the size of MSEG to be two pages.

Note The operating system always allocates one more page of MSEG than requested.

The size of the MSEG has no effect on execution time.

$OPTPARMS Directive

The $OPTPARMS directive protects CDS programs when too few parameters are passed in a subprogram call.

Syntax

```
              {ON}
$OPTPARMS {OFF}
```

Because EMA and character parameters are passed through descriptors, they are always referenced in the entry (prologue) code of the subprogram. If any EMA or character parameters are omitted, these references can cause a memory protect error. $OPTPARMS ON causes the prologue code to check for missing parameters to avoid aborting the program.

A subprogram must not access missing parameters; the PCOUNT function can be used to detect missing parameters. See Appendix B for more information on PCOUNT.

Programs compiled with $OPTPARMS on will run slightly slower.

$PAGE Directive

The $PAGE directive causes the compiler to send a page eject to the list file or device.

Syntax

```
$PAGE
```

The $PAGE directive must begin in column 1, and can appear anywhere in the source file. This directive has no effect if there are no list options in effect.

$SET Directive

The $SET directive assigns values to identifiers used in $IF, $ELSEIF, and $SET directives and for {*name*} substitution.

Syntax

$SET (*id1* = *value1*[, *id2* = *value2*]...)

where:

id is one or more identifiers given constant values.

value is a constant expression; all identifiers in the expression must be defined in previous $SET directives.

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 the program, the one has no effect on the other.

The $SET directive may appear only between modules. Among those directives placed between any two modules, before the first module, or after the last module, all $SET directives must appear before any $ALIAS or $EMA directives.

Examples

```
$SET (TOGGLE=.TRUE.,DEBUG=.FALSE.)
$SET (SYSTEM1=.TRUE.)
```

Identifiers defined in $SET directives may be referenced in FORTRAN statements by enclosing them in braces. The construct {*id*} is treated as if the constant value of {*id*} were used.

Example

```
$SET (version= '1.0')
   .
   .
   write(1,*) 'version', {version}
```

$SET identifiers can also be set using a $SET directive in the compiler command line. See "Compiler Invocation".

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

$SET (system='RTE-A' )
   .
   .
$IF  (system  .eq. 'RTE-4B')
   .
$ELSE IF (system  .eq. 'RTE-6')
   .
$ELSE IF (system  .eq. 'RTE-A')
   .
$ELSE
   .
$ENDIF
```

$TITLE Directive

The $TITLE directive causes the compiler to send a page eject to the list file or device and changes the main title to the string of characters specified in the directive.

Syntax

```
   $TITLE title
```

where:

title is up to 46 characters long and begins with the first nonblank character after $TITLE.

The $TITLE directive must begin in column 1 and can appear anywhere in the source file.

The main title appears on the first line following the page/option/date heading line. If no $TITLE directive appears as line 1 of the source file (if it has no control statement) or line 2 of the source file (immediately following its control statement, if there is one), the first main title is normally blank. If the first $TITLE directive appears in one of these two places, the first page of the listing does have the correct title.

If $LIST OFF is in effect, the title is changed from whatever it was to the specification in the $TITLE directive, but does not appear until the first page after $LIST ON is specified.

Example

The following is a source file that includes $TITLE directives:

```
$TITLE Example Program Number 1
      PROGRAM exg1
      x=0.
      DO 10 i=1,20
         CALL xray(i,x)
   10 CONTINUE
      END
$TITLE Different Title for Subroutine
      SUBROUTINE xray(i,x)
      x=x+i
      WRITE(1,'(F5.2)') x
      RETURN
      END
```

If this command is entered:

```
   FTN7X EXG1 1 -
```

The following listing results:

```
Page 1  FTN.            Opts: 77/LYI      Wed Sep 23, 1981 2:45 pm
 Example Program Number 1                 EXG1.FTN::SCRATCH

     2       PROGRAM exg1
     3       x=0.
     4       DO 10 i=1,20
               CALL xray(i,x)
     6    10 CONTINUE
     7      END

Module EXG1         No errors    Data:     2     Blank Common: none
FTN7X 5000/861229   No warnings  Code:    36     Stack size:     12

Page 2  FTN.            Opts: 77/LYI      Wed Sep 23, 1981   2:45pm
  Different Title for Subroutine              &EXG1 EXG1.FTN::SCRATCH

     9      SUBROUTINE xray(i,x)
    10      x=x+i
    11      WRITE(1,'(F5.2)')x
    12      RETURN
    13      END

Module XRAY         No errors     Data:  none     Blank Common:  none
FTN7X 2121/810908   No warnings   Code:    46     Stack size:      13
```

$TRACE Directive

The $TRACE directive causes a program to display subprogram entries and exits. Only subprograms compiled with TRACE ON are included. Tracing is done only in CDS programs.

Syntax

```
        {ON}
  $TRACE {OFF}
```

Tracing is always off when the compilation starts. Once tracing is turned on, it remains on until the next $TRACE OFF directive. $TRACE directives must be placed between modules. In subprograms with ENTRY statements, tracing is either in effect for all entry points to a module, or none of them. Tracing is ignored for non-CDS modules, but the option is remembered; if tracing is on but ignored (because of CDS OFF), a CDS ON directive restores tracing.

When $TRACE is used in the main program, a program cannot access its RMPAR parameters; those values are changed by the initial TRACE print. However, RCPAR, RHPAR, and program statement formal parameters are unaffected.

Example

Here is an example of trace output:

```
Enter: MAIN_PROGRAM
 Enter: INITIALIZE
  Enter: OPEN_FILES
   Enter: CHECK_FILE_NAME
    Enter: UPPERCASE
    Exit: UPPERCASE
   Exit: CHECK_FILE_NAME
  Exit: OPEN_FILES
 Exit: INITIALIZE
 Enter: PROCESS_COMMANDS
  Enter: DO_FIRST_COMMAND
  Exit: DO_FIRST_COMMAND
  Enter: DO_OTHER
  Exit: DO_OTHER
 Exit: PROCESS_COMMANDS
 Enter: CLEAN_UP
 Exit: CLEAN_UP
Exit: MAIN_PROGRAM
```

An Example with Multiple Directives

The following example uses several of the compiler directives described in this chapter to declare a VMA/EMA common area and sets up table area for two disk files. The program inverts a large two-dimensional array stored in the file LGDAT and stores it in a new file, NDT. The program does not show the user-written subroutine that performs the matrix inversion.

Example

```
$TITLE Matrix Inversion                              !These directives can
$EMA /set1/                                          !be in any order.
$FILES 0,2
        PROGRAM exg2
        COMMON /set1/matrix(250,250),inverse(250,250) !VMA/EMA common decl.
        CHARACTER*64 filename
        DATA fname/'../INPUT/'LGDAT '/
        OPEN(88,IOSTAT=ios,err=99,FILE=filename,STATUS='OLD')
C
C    First file opened.
C
        READ(88,11,END=111)((matrix(i,j),j=1,250),i=1,250)
     11 FORMAT(10F8.0)
        CALL matrix_inversion(matrix,inverse)   !User subroutine to invert.
        OPEN(89,IOSTAT=ios,err=99,FILE='../OUTPUT/NDT',STATUS='NEW')
C
C    Second file opened.
C
        WRITE(89,22)((matrix(i,j),j=1,250),i=1,250)
     22 FORMAT(10F8.2)
        CLOSE (88)
        CLOSE (89)
        STOP 'All done'
$PAGE

C    Print errors; quit.

      99 WRITE(1,*)'Error on open =',ios
         STOP
     111 STOP 'Bad data file - EOF encountered'
         END
```

Interfacing FORTRAN with Non-FORTRAN Subprograms

FORTRAN 77 programs can call subprograms written in Pascal, BASIC, or Macro/1000, and programs in these languages can call FORTRAN 77 subprograms. When passing arguments between subprograms of different languages, be aware that the argument values may be represented differently in different languages.

Calling FORTRAN Subprograms from Non-FORTRAN Programs

When a non-FORTRAN program calls FORTRAN subprograms, each FORTRAN subprogram should do *one* of the following:

- Specify the $FILES 0,0 compiler directive.
- Use the IOSTAT keyword, or END and ERR keywords, or all, in each input/output statement. Also do not use the STOP statement, CALL EXIT, character operations, or mathematical intrinsics.
- Specify a compiler option (0 through 9) to change the name of the error handler, and provide a custom error handler with the new name. See "FORTRAN Control Statement" earlier in this chapter for information on the compiler options.

FORTRAN and Pascal

You should be aware of the following differences between FORTRAN 77 and Pascal when interfacing between them:

- Logical and Boolean

  In FORTRAN the logical value of true is any negative value and the value of false is any non-negative value. In Pascal, the Boolean type defines the value of true as 1 and the value of false as 0.

- Arrays

  In FORTRAN, arrays are stored in column-major order, while in Pascal, arrays are stored in row-major order.

- Files

  Pascal files can be accessed from FORTRAN by passing the Pascal file variable to the FORTRAN subprogram. The FORTRAN subprogram then can access the file using FMP calls, specifying the file variable as the DCB. Pascal cannot access FORTRAN files. Files accessed through FMP calls can be used by either language.

- EMA

  To pass EMA arguments between Pascal and FORTRAN 77 programs, the FORTRAN program should use the EMA statement or the EMA option of the ALIAS directive or be compiled with the E option; the Pascal program should use the $HEAPPARMS ON$ directive. When a FORTRAN program passes EMA arguments, all arguments should have long addresses, either by being in EMA, by using the ALIAS directive, or by being compiled with the E option.

- Pascal value parameters

  FORTRAN treats Pascal value parameters as though they were passed as VAR parameters, that is, as though they were passed by reference. Pascal treats FORTRAN parameters as VAR parameters or value parameters, according to the Pascal declarations.

Reducing the Size of a Loaded Program

When a FORTRAN program uses file input/output, additional initialization and error handling routines are loaded with the program. This increases the size of the loaded program. STOP, PAUSE, CALL EXIT, character operations, and some intrinsic functions also cause additional initialization and error handling routines to be loaded.

If a FORTRAN program does no file input/output, the size of the loaded program can be minimized by doing the following:

- In the main program, specify $FILES 0,0 or no $FILES directive.
- In each subprogram, do *one* of the following:
  - Specify the $FILES 0,0 directive.
  - Use the IOSTAT keyword, or the END and ERR keywords, or all, in each input/output statement. Also, do not use the STOP statement, CALL EXIT, character data, or mathematical intrinsics.
- Specify a compiler option (0 through 9) to change the name of the error handler, and provide a custom error handler with the new name. See "FORTRAN Control Statement" earlier in this chapter for information on the compiler options.

8

ANSI 66 Compatibility Extensions

In addition to the ANSI 77 standard features and extensions discussed previously in this manual, the FORTRAN 77 compiler provides some ANSI 66 compatibility extensions. These extensions are for backward compatibility only. Their use in new program development is discouraged, as the current ANSI 77 standard is more widely accepted. The descriptions in this chapter assume the reader is familiar with the ANSI 77 standard features of FORTRAN 77.

The ANSI 66 compatibility extensions can be described in two parts. Certain features are included in FORTRAN 77 but either operate differently or are implemented differently under the ANSI 66 standard. These conflicts are discussed in "66 Mode Compared with 77 Mode" below. Other features of the ANSI 66 standard and extensions are part of the FORTRAN 4X compiler, but are not defined as part of the ANSI 77 standard. These features are described in "Compatibility Features" below.

66 Mode Compared with 77 Mode

Certain features of the ANSI 66 standard and the extensions used in FORTRAN 4X are not compatible with FORTRAN 77. These features are summarized in Table 8-1. The FORTRAN 77 compiler provides a means of resolving these conflicting features in a 66 manner (using 66 mode) or according to the ANSI 77 standard (using 77 mode).

To specify 77 mode, type `FTN77` at the beginning of the control statement in the source file. To specify 66 mode, type `FTN66` or `FTN4`. Then, when the source file is compiled, all conflicts encountered in that source file are resolved according to the specified mode.

If no control statement is given, or if the control statement does not specify a mode (for example, `FTN`) the program compiles in the mode specified when the compiler was linked. When linking the compiler, the System Manager can change the RPL value of Z$F67 to 7 for a default of 77 mode or to 6 for a default of 66 mode. (Refer to the "FTN7X file for more information.)

This section briefly describes the methods of 66 mode for resolving conflicting features, and briefly compares the methods of 77 mode. The methods of 77 mode are fully discussed in previous chapters of this manual.

The following table summarizes the conflicts between 66 mode and 77 mode. The number in the left column indicates the subsequent section number in which each topic is discussed.

Table 8-1. 77 Mode and 66 Mode Conflicts

1	DO loop index and evaluation.
2	Computed GOTO value out-of-bounds condition.
3	Intrinsic functions declared in EXTERNAL and type statements.
4	Complex variables, slash (/), and end-of-line in list-directed input.
5	Mixed-type operations on complex and double precision operands.
6	Storage of Hollerith constants.
7	Unformatted input/output and paper tape length words.

1. The syntax of the DO loop is the same in 66 mode as in 77 mode, but the loop exit condition is tested at a different point in the loop.

   Syntax

   DO [*label* [,]] *index* = *init*,*limit* [,*step*]

   When a DO statement is executed in 66 mode, the following sequence occurs:

   a. The control variable (*index*) is assigned the value of *init*.

   b. The range of the loop is executed.

   c. *index* is incremented by the value of *step*.

   d. *index* is compared with *limit*, and the following occurs:

   If *index* is less than or equal to *limit* and *step* is positive, or if *index* is greater than or equal to *limit*, and *step* is negative, the sequence is repeated starting at step (b).

   If *index* exceeds *limit* and *step* is positive, or if *index* is less than *limit* and *step* is negative, the DO loop is satisfied and control transfers to the statement following the termination statement.

   The major difference in DO loop execution is that in 66 mode, the index is tested at the end of the loop, whereas in 77 mode, the index is tested at the beginning of the DO loop. Therefore, in 66 mode, the DO loop is always executed at least once.

   In 66 mode, the index must be of type integer or double integer.

   The index, limit, step values, or all can be modified in the DO loop if they are variables, which can affect the number of times the loop is executed. In 77 mode, the loop count is unaffected.

2. The syntax of the computed GOTO is the same in 66 mode as in 77 mode.

   Syntax

   GOTO (*label1*,*label2*, ... ,*labeln*) [,] *index_exp*

   The difference for the computed GOTO in 66 mode and 77 mode lies in how the compiler handles an index expression that does not have a corresponding label. In 66 mode, if *index_exp* is less than 1, control passes to the statement whose label is first in the label list (*label1*). If the expression evaluates to a value greater than the number of labels in the label list, control passes to the statement whose label is last in the list of labels. With 77 mode, control passes to the statement following the computed GOTO if the index expression (*index_exp*) does not have a corresponding label.

3. In 66 mode, if an intrinsic function name is to be passed as an actual argument, it must be specified in an EXTERNAL statement. An intrinsic function becomes nonintrinsic (that is, a user-defined symbolic name) only if used as a variable or subroutine name, or if explicitly typed differently from its implicit type (with a type statement).

   In 77 mode, intrinsic functions to be passed as actual arguments must be specified in the INTRINSIC statement. An intrinsic function name in an EXTERNAL statement always loses its intrinsic properties and becomes a user-defined symbolic name. Type statements have no effect on intrinsic names.

4. In the input record of a list-directed READ, complex values can have a different form. The slash (/) and end-of-record have opposite meanings in 66 mode and 77 mode.

   In 66 mode, the complex value in an input record must not have parentheses surrounding the real and imaginary numbers. These numbers are separated by only a comma when that value is read with a list-directed READ statement. The parentheses are required in 77 mode.

   In 66 mode, when a list-directed READ statement is executed and a slash (/) is encountered in the input record, all entries following the slash on the current record are ignored, and the read continues on the next record until the input list items are satisfied. If an end-of-record is encountered, the READ terminates and any items remaining in the READ statement are unchanged.

   In 77 mode, if an end-of-line (end-of-record) is encountered, the read is 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.

5. Arithmetic expressions involving operands of two different types, complex and double precision, produce a result of type complex in 66 mode. In 77 mode a similar expression produces a result of type double complex.

6. Hollerith constants are discussed in "Compatibility Features" below. Note that the constant is stored as a different type depending on the number of characters and on the mode.

Number of Characters	**66 Mode**	**77 Mode**
1 or 2	Integer	Integer (I compiler option) or double integer (J compiler option)
3 or 4	Real	Double integer
5 or 6	Extended precision	Double precision
7 or 8	Complex	Double precision

7. Using unformatted input/output in 66 mode, device types 00 through 17 octal use paper tape length words. In 77 mode, device types 1 and 2 use paper tape length words.

Compatibility Features

The compatibility features are included in FORTRAN 77 for backward compatibility only. These features are defined as part of the ANSI 66 standard or as extensions that were included in FORTRAN 4X but were not included as part of the ANSI 77 standard. The compatibility features are listed in Table 8-2, with complete descriptions following. These features are always available as part of FORTRAN 77 in addition to the features described in previous chapters. No special action is required to enable the compatibility features.

Although FORTRAN 77 accepts these features, they should not be used in new programs. For each feature, there is a corresponding or alternative feature in the ANSI 77 standard which should be used instead. For example, the CHARACTER data type should be used instead of Hollerith.

Each of the features in Table 8-2 is discussed in detail after the table. The number in the left column of Table 8-2 refers to the number of the paragraph where the feature is discussed.

Table 8-2. Compatibility Features

1	Extended precision type.
2	Hollerith constants.
3	*n*H editing in formatted input.
4	Character constant initializing noncharacter item in DATA statement.
5	An integer array name where character expression required in an input/output statement.
6	The A and R format descriptors for input/output of noncharacter items.
7	DECODE and ENCODE statements.
8	Improper array dimensioning in EQUIVALENCE statement.
9	Record number connected to unit number (*unit'recnum*).
10	Statement function in EXTERNAL statements as arguments.
11	Two-way arithmetic IF statements.
12	Parentheses around simple input/output lists.
13	D in column 1 denoting debug line.
14	$ as statement separator.
15	A subscript may have a non-INTEGER value.
16	Ampersands (&) instead of asterisks (*) in alternate returns.

1. The *extended precision* type defines a set of real numbers. Extended precision values have the following range:

   $-1.70141183460 \times 10^{+38}$ to $-1.469367938528 \times 10^{-39}$
   0.0
   $1.469367938528 \times 10^{-39}$ to $1.70141183460 \times 10^{+38}$

   Objects of type extended precision are represented in three 16-bit words and have an accuracy of approximately 11.6 to 11.9 digits (that is, one part in $10^{11.6}$ to $10^{11.9}$).

   A variable can be explicitly typed as extended precision by specifying it in a REAL*6 or DOUBLE PRECISION*6 type statement, or in a DOUBLE PRECISION statement with the X option specified in the control statement.

   Extended precision constants have the same syntax as double precision constants. Constants with this syntax are stored as extended precision if the X option in the control statement is specified; otherwise, they are double precision constants.

The data format in memory of extended precision storage is shown in Appendix D.

2. Hollerith constants are a special format for character strings that are stored as numeric values. A Hollerith constant consists of an integer specifying the number of characters (including blanks), followed by the letter H and the character string (without delimiting single quotation marks).

   A Hollerith constant is a numeric value. It can be used in an arithmetic expression; it cannot be used in a character expression.

   Hollerith constants are stored as the following types:

Number of Characters	66 Mode	77 Mode
1 or 2	Integer	Integer (I compiler option) or double integer (J compiler option)
3 or 4	Real	Double integer
5 or 6	Extended precision	Double precision
7 or 8	Complex	Double precision

   A Hollerith constant with greater than eight characters is legal only as an actual argument in a CALL statement or function reference, or to initialize multiple array elements in a DATA statement, in which case the Hollerith constant must be the exact size of the array it is used to initialize.

   Examples

```
2H$$                6H&PROGA
8HA STRING          3H12A
12HReport Title     7HQU'OTED
```

Note Since Hollerith constants offer no advantages over character constants and are less flexible, they should be avoided.

The data format in memory of Hollerith constants storage is shown in Appendix D.

3. The *n*H Hollerith edit descriptor is mainly used to write to the output record the *n* characters (including blanks) that follow the H. *n*H editing, if used on input, replaces the corresponding characters in the format descriptor with the characters in the input field. If the format is then used on output, the new value is used. Use of *n*H editing on input is discouraged, since program portability may be inhibited. This feature does not work in CDS programs.

4. Character constants can be used as Hollerith constants to initialize noncharacter items in DATA statements. These constants must be delimited by single quotation marks ('). The size of the character constant can be less than the size of the noncharacter item.

Example

```
DIMENSION namr(10)
DATA namr/'FILE1:SU:-31'/
```

As with Hollerith constants, a character constant may initialize more than one element of an array. If the length of the constant is not an exact multiple of the array element size, the remaining characters in the last initialized element are padded with blanks. The next constant initializes the following array element. The rule for concatenating adjacent character constants does not apply in this case.

5. An integer variable can be used as the format specifier in an input/output statement. If the variable represents an array, the array is assumed to contain an ASCII format. As defined in the ANSI 77 standard, if the variable is a simple variable, it must be defined by an ASSIGN statement.

   An integer array name is also allowed with the following specifiers in input/output statements:

   ```
   FILE=       NAME=          UNFORMATTED=
   USE=        SEQUENTIAL=    FMT=
   STATUS=     DIRECT=        FORM=
   ACCESS=     FORMATTED=     BLANK=
   ```

   Example

   ```
   INTEGER fname(5)
   DATA fname/10H&FILE1::MW/
   OPEN(UNIT=66,FILE=fname)
   ```

   In this example the integer array `fname` is used where a character expression is required by the ANSI 77 standard. Simple variables or array elements may *not* be used this way.

6. Input and output of ASCII data to and from noncharacter variables can be done with the A and R format descriptors.

7. The DECODE and ENCODE statements allow you to do formatted internal-to-internal data transfer (that is, from and to internal records). This feature is provided in the ANSI 77 standard through internal files. DECODE and ENCODE should not be used in new programs; use internal files instead.

   Syntax

   ```
   DECODE (c,f,buffer[,IOSTAT=ios][,ERR=label]) list
   ENCODE (c,f,buffer[,IOSTAT=ios][,ERR=label]) list
   ```

   where:

   c is the maximum number of characters in the internal record.

   f is a format statement label.

   buffer is the location of the internal record and can be any type except character.

ios is an integer variable or integer array element name for error code return (refer to Appendix A for IOSTAT error codes).

label is the statement label of an executable statement in the same program unit as the DECODE or ENCODE statement. If an error occurs during the execution of the DECODE or ENCODE statement, control is transferred to the specified statement rather than the program's being aborted.

list is an input/output list of variables.

DECODE converts the first *c* characters in the internal record buffer from ASCII data into the input/output list items, using the format specification *f*; that is, DECODE reads from the internal record to the list. List-directed DECODEing is allowed (that is, *f* = *). The value of *c* determines the record size.

ENCODE converts the elements of the list to ASCII data according to the format specification *f* and stores the result in the internal record buffer (an array or a simple variable); that is, ENCODE writes from the list to the record. The value of *c* must be greater than or equal to the record size specified in the FORMAT statement. List-directed formatting can be used in the ENCODE statement.

Example

```
    n = 10
    m = 5
    ENCODE (20,33,ifmt) n,m
33  FORMAT ('('I2,'X,'I2,'(I2,X))')
```

Shows the use of ENCODE to form a FORMAT specification containing variables. The resultant FORMAT stored in `ifmt` is:

```
(10X, 5(I2,X))
```

and might be used by:

```
WRITE (1,ifmt) iarray
```

If the format/list combination requires that more than one record be read or written, the buffer is considered to have as many records as needed, in contiguous storage locations. If *c* is odd, every other record begins in the middle of a word.

8. A multidimensional array can be referenced in an EQUIVALENCE statement with fewer subscripts than that which the array was declared. The missing subscripts are set to the corresponding lower bounds of the array. If only one subscript is used and the first lower bound is 1, the subscript specifies a linear position in storage. For example, the statements:

```
DIMENSION a(2,2),i(4)
EQUIVALENCE (a(2,1),i(2))
```

produce the following storage space allocation:

Array a	Storage Space Word Number	Array i
a(1,1)	1	
	2	i(1)
a(2,1)	3	i(2)
	4	i(3)
a(1,2)	5	i(4)
	6	
a(2,2)	7	
	8	

The statements:

```
DIMENSION a(2,2), i(4)
EQUIVALENCE (a(3),i(4))
```

produce the same storage space allocation as the previous example. This EQUIVALENCE statement references a multidimensional array with a single subscript.

9. The `UNIT=` specifier in direct access input/output statements can be qualified by a record number:

   `[UNIT=]`*unit’recnum*

   where *recnum* is an integer expression. If this form is used, the specifier `'REC=recnum'`, which is the ANSI 77 standard method of specifying the record number, is not allowed in that statement.

   Example

   ```
   WRITE(UNIT=61'irecnum,11,IOSTAT=ios,ERR=99) record
   ```

10. A statement function name can be used in an EXTERNAL statement, allowing that function to be passed as an actual argument in a subprogram call. The ANSI 77 standard does not allow statement functions to be used as arguments.

11. A two-way arithmetic IF statement transfers control to one of two statements rather than three. The syntax is the same as the arithmetic IF statement defined by the ANSI 77 standard except that the third label is not specified.

    Syntax

    `IF (`*exp*`)` *label1*`,`*label2*

    where:

    exp is an arithmetic expression of any type except complex.

    label is the statement label of an executable statement.

    The statement to which control transfers is determined as follows:

If *exp* < 0, control transfers to *label1*.

If *exp* ≥ 0, control transfers to *label2*.

Therefore, the statement

```
IF (a+b) 100,200
```

is the same as the statement

```
IF (a+b) 100,200,200
```

The expression in a two-way arithmetic IF statement can be a logical expression, with the first branch executed if the value is true and the second branch executed if the value is false.

12. Each READ, WRITE, or PRINT statement has a list of items to be read into or written from. These items can be variables, expressions, or an implied DO loop surrounded by parentheses. As a compatibility feature, any combination of one or more of these items in an input/output statement can optionally be surrounded by parentheses.

Note Use this feature carefully with list-directed output. If the input/output list enclosed in parentheses has the form of a complex constant, it is used as a complex constant.

Example

```
READ(1,*) (var1,var2,var3),((matrix(i,j),i=1,5),j=1,10)
```

In this example, `var1`, `var2`, and `var3` are surrounded by parentheses. The parentheses are not required but are allowed as a compatibility feature. This READ statement also involves an implied DO loop for the array matrix. This implied DO loop is also surrounded by parentheses, which in this case are required.

13. The letter D in column 1 of a line designates that line as a debug line. Compilation of debug lines is optional. Unless specifically directed to compile debug lines, the compiler treats debug lines the same as comment lines designated with C.

    To cause compilation of debug lines, specify the D option in the control statement or in the compiler command line (see Chapter 7 for the forms of the command line and control statement).

14. More than one statement can appear on a line if the dollar sign ($) is used as a statement delimiter.

    Example

    ```
    i = 0 $ r = 0.0589 $ s = q*(4.82+t)
    ```

15. If a subscript has a non-INTEGER value, it is converted to INTEGER (using truncation) before it is used.

16. An ampersand (&) may be used in place of an asterisk (*) to denote an alternate return in a SUBROUTINE statement or to indicate a statement number passed as an actual parameter for alternate returns.

Error Messages

There are two kinds of FORTRAN error messages. Those produced by the compiler are called *compilation errors, compile-time errors*, or just *compile errors.* Those produced while a FORTRAN program is running are called *run-time errors*.

Types of Compilation Errors

There are three types of FORTRAN 77 compilation errors:

Warning The compiler continues to process the statement, but the object code may be erroneous. The program should be recompiled.

Error The compiler ignores the remainder of the erroneous source statement, including any continuation lines. The object code is incomplete, and the program must be recompiled.

Disaster The compiler ignores the remainder of the FORTRAN 77 source file. The error must be corrected before compilation can proceed.

Note If the compiler detects an error in a program, the object code contains a reference to the undefined external name <Compile Errors>. This prevents loading of the object code, unless forced by the user. It is strongly recommended that a program with compilation errors not be executed.

The reference to the undefined external name is not produced for warnings.

Format of Compilation Errors

When an error is detected in a source statement, the source statement is printed. A question mark (?) is printed after the erroneous column. Then a message is printed in the format:

```
                    ⎧ warning  ⎫
** program name **  ⎨  error   ⎬  nn detected at line xx column cc
                    ⎩ disaster ⎭
```

where:

program name is the name of the program unit being compiled.

nn is the error number.

xx is the line number of the source file where the error occurred.

cc is the column number of the source line where the error was detected.

If the column number is unknown, it is not printed. If the source line is not available (for example, an error occurs on the second pass of the compiler with the L option), the source line of the error is not printed.

After the END statement, if the IMPLICIT NONE statement is used in the source file and some names are not explicitly typed, those names are listed after a warning message.

If undefined source program statement labels still exist, an error message for each undefined label is printed in the form:

```
undefined statement number: #nnnnn
```

where:

nnnnn is the statement number that does not appear in columns 1 through 5 of any of the initial lines of the program just compiled, or appears but is illegal.

A brief explanation of each error detected is output to the list file. The form of this error description is:

```
Error Directory

Number Explanation

nn  description
mm description
 :
etc.
```

where:

`Error Directory and`
`Number Explanation` are headings.

nn and *mm* are the error numbers detected.

description is a brief explanation of the error number.

When an error occurs in accessing files, a *disaster* is generated. On some operating systems, a message is printed giving the nature of the error and the file name. On other operating systems, only the disaster number is given; the file on which the error occurred can be found from the disaster number (for example, disaster 97 refers to an error in an access to the relocatable output file).

Compilation Error Summary

When compilation is completed, a summary message is displayed at the initiating terminal. This message reports the number of disaster, error, and warning conditions encountered during compilation of all the program units. The FORTRAN 77 compiler returns this information by the parameter return subroutine PRTN (see the appropriate system reference manual for a description of this subroutine). The message appears in the form:

```
END ftn7x: nn disasters, nn errors, nn warnings
```

where:

nn is the total number of disaster, error, or warning conditions detected during compilation of all programs (*nn* = No, if none were encountered).

The parameters returned by PRTN are:

parameter 1 The sum of parameters 2 through 4.

parameter 2 The number of disasters detected.

parameter 3 The number of errors detected.

parameter 4 The number of warnings detected.

parameter 5 The revision level of the compiler, as printed in the summary lines in the listing (see Chapter 7), for example, 5000.

FORTRAN 77 Compilation Error Messages

Table A-1 lists the FORTRAN 77 compilation error messages and briefly describes the causes of the errors.

Table A-1. FORTRAN 77 Compilation Error Messages

1	**ERROR MESSAGE**	`Error in ftn directive.`
	CAUSE	The first line of the source begins with "FTN" but does not conform to the syntax for the FTN directive or has an illegal option.
2	**ERROR MESSAGE**	`Illegal option in runstring.`
	CAUSE	The fifth parameter in the runstring contains a character that is not a legal option. The options I, J, X, Y, and E can appear in the FTN directive, but not in the runstring.
3	**ERROR MESSAGE**	`Compiler space overflow; compiler needs more EMA memory.`
	CAUSE	Refer to the installation manual for information on how to size the compiler.
4	**ERROR MESSAGE**	`Invalid common label.`
	CAUSE	The common label given has an illegal character or is not followed by a matching slash (/).
5	**ERROR MESSAGE**	`Implicit is redundant or retypes named constant.`
	CAUSE	A starting letter or range of letters is used in more than one IMPLICIT group, or a name in a PARAMETER statement has its implicit type changed by this statement.

6 **ERROR MESSAGE** `Transfer of control into a loop or IF-THEN-ELSE block.`

CAUSE A statement in a loop or IF-THEN-ELSE block is referenced from outside the loop or block. The program may not run as expected. When control is transferred into a DO loop, the loop may not be initialized correctly; in an IF-THEN-ELSE, the block may have been optimized out.

7 **ERROR MESSAGE** `ENTRY or RETURN in main program or BLOCK DATA.`

CAUSE These statements are legal only in subroutines and functions. If a SUBROUTINE or FUNCTION statement has an error, the compiler may consider this module to be a main program.

8 **ERROR MESSAGE** `Illegal complex number.`

CAUSE The number looks like a complex constant, but does not conform to the rules for forming such constants.

9 **ERROR MESSAGE** `Mismatched parenthesis.`

CAUSE There are either more right parentheses than left parentheses at some point, or not as many right parentheses as left parentheses at the end of the expression.

10 **ERROR MESSAGE** `Unrecognized statement.`

CAUSE The statement does not look like an assignment statement or a statement function, and does not start with a recognized keyword.

11 **ERROR MESSAGE** `Dimension/substring 2nd part < 1st part.`

CAUSE The upper bound or last character position is less than the lower bound or first character position.

12 **ERROR MESSAGE** `Return # too large or too many alternate returns.`

CAUSE The (constant) return number in a RETURN statement exceeds the number declared in the SUBROUTINE statement, or a system intrinsic (for example, EXEC) call has more than one alternate return specified.

13 **ERROR MESSAGE** Constant > 2047 in format, or illegal string.

CAUSE Numeric value in a FORMAT is too large, or the closing quote of a string is missing, or an ALIAS string is too long.

14 **ERROR MESSAGE** Constant or constant expression overflow or underflow.

CAUSE A constant (expression) exceeds the number range of the machine, using the data types of the constants.

15 **ERROR MESSAGE** Keyword unrecognized, repeated, or illegal.

CAUSE An I/O keyword is misspelled, has already been used in this I/O statement, or is illegal in this I/O statement.

16 **ERROR MESSAGE** Illegal octal or hex constant.

CAUSE The constant has an illegal character or is too big.

17 **ERROR MESSAGE** Missing constant or operand.

CAUSE An operand was expected but a delimiter or an unrecognized character was found. This can be caused in a number of ways by improper syntax or badly formed names or constants.

18 **ERROR MESSAGE** Illegal combination of keywords.

CAUSE Some of the keywords used in this I/O statement cannot be used together.

19 **ERROR MESSAGE** Integer constant expected.

CAUSE Only an integer constant, or possibly an integer constant expression, can be used in this context.

20 **ERROR MESSAGE** `Illegal character count in hollerith constant.`

CAUSE A negative or zero integer constant is followed by “H”. Hollerith constants must have character counts greater than zero, and the “H” is not legal here if Hollerith was not intended.

21 **ERROR MESSAGE** `Value out of range.`

CAUSE The constant value given is too large, too small, or otherwise unacceptable.

22 **ERROR MESSAGE** `Illegal use of name.`

CAUSE This name has been used before in a different context that conflicts with its current use.

23 **ERROR MESSAGE** `Step size = 0.`

CAUSE The step size in a DO loop cannot be zero.

24 **ERROR MESSAGE** `Variable or array name expected.`

CAUSE A variable or array name is expected here, but a delimiter, constant, named constant, or subprogram name is found.

25 **ERROR MESSAGE** `Variable name or constant expected.`

CAUSE A name is expected here, but a delimiter, constant, named constant, or subprogram name is found.

26 **ERROR MESSAGE** `Integer (logical) item expected.`

CAUSE The context calls for an item of a certain type (integer or logical) but the item given is of a different type.

27 **ERROR MESSAGE** `Duplicate statement number.`

CAUSE This statement number has already been used.

28 **ERROR MESSAGE** `Unexpected character or unexpected end of statement.`

CAUSE The compiler did not recognize the character, or was expecting more characters to complete the statement.

29 **ERROR MESSAGE** `Blank line has statement number.`

CAUSE Blank lines should not have statement numbers; blank lines are comments.

30 **ERROR MESSAGE** `Incorrect nesting.  May be due to other errors.`

CAUSE If there are other errors, this error may be ignored until the other errors are corrected. Can be caused by END DO, ELSE, ELSE IF, and ENDIF statements after errors. It may also be caused by ending a block (for example, a DO loop) before all inner blocks have been ended.

31 **ERROR MESSAGE** `Compiler space overflow; compiler needs more (EMA) memory.`

CAUSE Refer to the Configuration/Installation manual for information on how to size the compiler.

32 **ERROR MESSAGE** `Undefined, illegal or incorrectly used statement number.`

CAUSE This statement number has an illegal character in it, was never defined, or was used in the wrong context (for example, a FORMAT statement number cannot be used in a GOTO).

33 **ERROR MESSAGE** `Redundant/conflicting/missing EXTERNAL/INTRINSIC declarations.`

CAUSE A subprogram name was passed as an actual parameter but not declared EXTERNAL or INTRINSIC, or was declared more than once in EXTERNAL or INTRINSIC statements.

34 **ERROR MESSAGE** `Statement out of order.`

CAUSE This statement appears too late in the source. This can be due to errors in earlier statements.

---

35 **ERROR MESSAGE** `No path to this statement.`

CAUSE The statement before this one always transfers control (for example, a GOTO) but this statement is not labeled, so it can never be reached. If this happens in old code that uses GOTOs after EXEC calls with the no-abort bit, the calls *must* be changed to use alternate returns. The use of GOTO with the no-abort bit is *not* supported.

---

36 **ERROR MESSAGE** `Variable appears twice in common.`

CAUSE A variable or array name appears more than once in COMMON statements.

---

37 **ERROR MESSAGE** `Formal parameter in COMMON or DATA statement.`

CAUSE Parameters are illegal in this context.

---

38 **ERROR MESSAGE** `Wrong number of subscripts.`

CAUSE This array reference has a different number of subscripts than were declared when the array was dimensioned.

---

39 **ERROR MESSAGE** `Illegal variable in dimension bound expression.`

CAUSE Only formal arguments and variables in common may be used as dimension bounds or in bound expressions.

---

40 **ERROR MESSAGE** `Inconsistent equivalence group.`

CAUSE It is impossible to perform the equivalences specified, because more than one location has been given for an item. Examples: Items in different common blocks; or two different elements of an array specified, but not aligned properly.

---

41 **ERROR MESSAGE** `Negative extension of common via equivalence.`

CAUSE An array element was equivalenced to an item in common in such a way that the start of the array would be before the start of the common block.

---

42 **ERROR MESSAGE** `Left parenthesis expected.`

CAUSE Probably a function reference without a parameter list. Parameter lists are required, even if empty; for example, PCOUNT( ).

---

43 **ERROR MESSAGE** `Variable used in a context that requires a formal parameter.`

CAUSE Some operations, such as declaring an array to be variably dimensioned, are only legal for formal parameters.

---

44 **ERROR MESSAGE** `Constant missing, ill-formed, or of wrong type.`

CAUSE The compiler could not recognize a constant of an acceptable type in this context.

---

45 **ERROR MESSAGE** `Illegal combination of data types.`

CAUSE The current operation cannot be performed on the items given; the data types are not compatible.

---

46 **ERROR MESSAGE** `Name of a function not used or name of a subroutine is used.`

CAUSE A function name must be used for assigning a result value; a subroutine name must not be used as a variable.

---

47 **ERROR MESSAGE** `Variable dimension bound not in same entry list as the array.`

CAUSE The specified variable is a formal parameter but is not given in an entry list that contains the array; so the bound is undefined at that entry, and the array address calculations cannot be done.

---

48 **ERROR MESSAGE** `Illegal use of EMA variable.`

CAUSE EMA variables are not allowed in some contexts, such as some I/O keyword values.

---

49 **ERROR MESSAGE** `Function has illegal mix of entry point types.`

CAUSE Character and noncharacter entry points cannot be mixed, and character entry points must all have the same length.

---

50 **ERROR MESSAGE** `Illegal last statement of DO loop.`

CAUSE This statement cannot end a DO loop. In particular, any statement that always causes a transfer of control is illegal here (for example, GOTO or arithmetic IF).

---

51 **ERROR MESSAGE** `Control variable of DO statement already in use.`

CAUSE An outer loop uses the same index variable as this inner loop.

---

52 **ERROR MESSAGE** `This statement may not be used as the true part of a logical IF.`

CAUSE DO, logical IF, block-IF, ENTRY and END statements cannot be used as the statement following a logical IF.

---

53 **ERROR MESSAGE** `Illegal use of character*(*) or "*" last upper bound.`

CAUSE Character*(*) can only be used for a formal argument or named constant; “*” can only be used as the last upper bound of a formal argument.

---

54 **ERROR MESSAGE** `Array name dimensioned twice.`

CAUSE Array names followed by their dimensions can only appear once within the specification statements.

---

55 **ERROR MESSAGE** `Illegal use of non-character data.`

CAUSE Character data is required in this context.

56 **ERROR MESSAGE** `Illegal combination of data types.`

CAUSE These data types cannot be used together with any operator.

57 **ERROR MESSAGE** `Illegal data type.`

CAUSE This data type is illegal in this context (for example, with a certain operator, such as logical data with "+").

58 **ERROR MESSAGE** `Function used as subroutine or has alternate returns.`

CAUSE This subprogram is used as a function, but it has alternate returns or is also used as a subroutine. Mixed use as a function and subroutine is allowed for ALIASed items, but should not be used in new programs.

59 **ERROR MESSAGE** `Wrong # of arguments.`

CAUSE This intrinsic, statement function, or recursive call has an illegal or inconsistent number of actual arguments.

60 **ERROR MESSAGE** `Illegal argument type.`

CAUSE This intrinsic, statement function, or recursive call has an argument with an illegal or inconsistent type.

61 **ERROR MESSAGE** `Arithmetic IF with illegal data type.`

CAUSE The expression in an arithmetic IF must be numeric and not of type COMPLEX. Make sure the parentheses are balanced and the right side has the correct syntax.

62 **ERROR MESSAGE** `Logical IF or WHILE with non-logical data.`

CAUSE The expression in a logical IF or DO WHILE must be of type LOGICAL. Make sure the parentheses are balanced and the right side has the correct syntax.

63 **ERROR MESSAGE** `PCOUNT used with DIRECT entry point.`

CAUSE The number of actual parameters passed to a DIRECT entry point cannot be determined.

64 **ERROR MESSAGE** `"THEN" expected.`

CAUSE The only legal statement after ELSE IF (*expression*) is THEN.

65 **ERROR MESSAGE** `Non-character item on odd byte address.`

CAUSE A COMMON or EQUIVALENCE statement puts a word-addressed variable on an odd byte address. In a COMMON statement, a byte will be left unused.

66 **ERROR MESSAGE** `Program should (not) have executable statements.`

CAUSE Regular subprograms and main programs must have executable statements, and BLOCK DATA subprograms must not have them.

67 **ERROR MESSAGE** `Character item cannot be in EMA.`

CAUSE By explicit declaration, COMMON, or EQUIVALENCE, an attempt was made to put character data in EMA, which is illegal.

68 **ERROR MESSAGE** `ENTRY used in Block IF or DO loop.`

CAUSE All ENTRY statements must be at the "outer" level of a subprogram, and not within any DO or IF blocks.

69	**ERROR MESSAGE**	Substring out of bounds or non-character item has substring.
	CAUSE	In a character item, the starting character is less than one or the ending character is greater than the declared length.
70	**ERROR MESSAGE**	Constant subscript out of bounds.
	CAUSE	The subscripts are outside of the declared dimensions.
71	**ERROR MESSAGE**	Too many/few constants, or illegal repeat count.
	CAUSE	The number of constants does not match the number of variables and array elements, or the repeat count is negative or zero.
72	**ERROR MESSAGE**	Item must (not) be in common.
	CAUSE	In a BLOCK DATA subprogram, only items in labeled common can be used in DATA statements.
73	**ERROR MESSAGE**	Constant & variable have different types.
	CAUSE	This constant cannot be converted to the type of the corresponding variable. Make sure the number of variables and constants is correct up to this point.
74	**ERROR MESSAGE**	Undeclared array, or stmt funct after first executable stmt.
	CAUSE	This statement looks like an assignment to an array element or like a statement function, but the name is not declared as an array and it is too late in the program to define a statement function.
75	**ERROR MESSAGE**	Illegal use of current subprogram name or ENTRY name.
	CAUSE	Entry point names are illegal in this context (for example, a recursive call in non-CDS mode).

76 **ERROR MESSAGE** Duplicate formal parameter.

CAUSE A formal parameter appears more than once in this parameter list.

77 **ERROR MESSAGE** Statement number ignored.

CAUSE Statement numbers are not functional on specifications, DATA, or ENTRY statements. The number is ignored, as if the statement number field had been blank.

78 **ERROR MESSAGE** Overlays (type-5 programs) not allowed in CDS mode.

CAUSE The module has been converted to a zero-argument subroutine.

79 **ERROR MESSAGE** More than 255 parameters to a subprogram or library routine.

CAUSE The (CDS) PCAL instruction can only handle calls with 255 or fewer parameters. The offending call can be caused by certain long expressions, such as concatenations.

80 **ERROR MESSAGE** This statement not legal in BLOCK DATA.

CAUSE Only certain specifications and data statements are legal in BLOCK DATA; all executable statements and formats are illegal.

81 **ERROR MESSAGE** PROGRAM formal parameter is not type character.

CAUSE All formal parameters in a PROGRAM statement must be of type CHARACTER.

82 **ERROR MESSAGE** Compiler space overflow; compiler needs more memory or EMA.

CAUSE There was not enough memory available for the compiler to compile this module. Increase the "size" of the compiler. See your System Manager, who should refer to the Configuration/Installation manual for information on how to size the compiler.

83 **ERROR MESSAGE** `Attempt to retype a name.`

CAUSE A name appears in more than one type statement or twice in a single type statement.

---

84 **ERROR MESSAGE** `Code, data, common, save, stack or EMA space overflow.`

CAUSE The module is too big. If there are large arrays, try making them smaller or moving them to EMA. If the module is very long, try breaking it into two smaller modules.

---

85 **ERROR MESSAGE** `Program name conflicts with common, external or intrinsic name.`

CAUSE The program name should be unique. Most "internal" names used to access the library contain special characters, but a few do not, such as SIN. Choose another program name.

---

86 **ERROR MESSAGE** (not currently used)

CAUSE

---

87 **ERROR MESSAGE** `The given names were not typed.`

CAUSE IMPLICIT NONE was specified but some variables were not mentioned in type statements. Could be due to typographical errors.

---

88 **ERROR MESSAGE** `Cannot access list file.`

CAUSE An OPEN, CREATE, or WRITE of the list file failed. If the list file does not end with .LST (new files) or begin with a single quote (old files), it cannot be overwritten. If the list file is given the default name using the "–" convention, the source file name must end with .FTN (new files) or begin with an ampersand (old files).

---

89 **ERROR MESSAGE** (not currently used)

CAUSE

90 **ERROR MESSAGE**

```
Illegal continuation line.
```

CAUSE The first line of the program, or the first line after a directive, cannot be continued.

91 **ERROR MESSAGE**

```
The following external name is used in more than
one way.
```

CAUSE A subprogram name was used as an entry point or a common block name, or was ALIASed to match another subprogram name.

92 **ERROR MESSAGE**

```
External name conflicts with a library routine.
```

CAUSE An entry point, common block, or subprogram name matches the "internal" name used for a library routine. Most "internal" names contain special characters, but a few do not, such as SIN. Choose another name.

93 **ERROR MESSAGE** (not currently used)

CAUSE

94 **ERROR MESSAGE**

```
Item cannot be declared in EMA statement or is
declared twice.
```

CAUSE Only formal parameters and local variables can be declared to be in EMA, and must appear only once in EMA statements.

95 **ERROR MESSAGE**

```
Cannot open include file, or name greater than 63
chars.
```

CAUSE The OPEN of this include file failed. Check the name.

96 **ERROR MESSAGE**

```
Break detected.
```

CAUSE An operator BREAK was entered. The compilation was terminated. The relocatable file is not usable, and the list file may be incomplete.

97 **ERROR MESSAGE** `Cannot access relocatable output file.`

CAUSE An OPEN, CREATE or WRITE of the relocatable file failed. If the relocatable file does not end with .REL (new files) or begin with a percent sign (old files), it cannot be overwritten. If the relocatable file is given the default name using the "−" convention, the source file name must end with .FTN (new files) or begin with an ampersand (old files).

---

98 **ERROR MESSAGE** `Cannot access source file, or eof before end.`

CAUSE The end of the source file was encountered in the middle of a module (because of a missing or unrecognized END), or an OPEN or READ of the source file failed, or a READ of an included file failed.

---

99 **ERROR MESSAGE** `Can't access scratch file(s).`

CAUSE A CREATE or WRITE of internal scratch files failed, probably due to lack of space. Make sure there is space available on the scratch cartridge, working directory, or top cartridge, depending on your operating system. If in doubt, consult the System Manager.

---

100 **ERROR MESSAGE** (not currectly used).

CAUSE

---

101 **ERROR MESSAGE** `$IF/$ELSE/$ENDIF nested incorrectly, or nested too deeply.`

CAUSE These directives were nested to more than 16 levels, or incorrectly nested.

---

102 **ERROR MESSAGE** `Command-line directive must start with $; must not be INCLUDE.`

CAUSE The sixth command-line argument on the FTN7X command line did not start with $ or was $INCLUDE.

---

103 **ERROR MESSAGE** `Ill-formed {name} reference to $SET variable.`

CAUSE A left brace "{" was not followed by a name and a right brace "}".

104 **ERROR MESSAGE** `No such $SET variable.`

CAUSE This name has not been defined in a $SET directive yet.

105 **ERROR MESSAGE** `$SET used between $ALIAS or $EMA and start of subprogram.`

CAUSE $SET may not be used between $ALIAS or $EMA and the start of the following subprogram.

106 **ERROR MESSAGE** `Character item used on both sides of assignment; must not overlap.`

CAUSE A character variable, array, or substring was used on the left side of an assignment and in the expression on the right side. It is possible that the two usages overlap, which could produce an incorrect result or run-time error. The compiler can only determine the possibility of overlap. See the P option in Chapter 7.

107 **ERROR MESSAGE** `Variables in absolute common blocks may not appear in DATA statements.`

CAUSE When $ALIAS is used to place a common block at an absolute address, variables in that common block may not be initialized in DATA statements.

Library Subroutine Error Messages

During the execution of FORTRAN 77 programs, errors can be generated from several sources, including input/output formatter errors, remote file access errors, and errors from references to Relocatable Library Subroutines. Error messages from the Relocatable Library Subroutines are listed in the first part of this appendix. The input/output run-time errors are listed in the second part of this appendix. Refer to the appropriate system reference manual for applicable system error messages and information about FMP, DS, and EXEC.

Error messages caused by CDS subprograms and non-CDS subprograms are slightly different:

CDS: */pname* `Runtime error` *xxxx* `in code segment` *zz* `at address` *yyyy*

non-CDS: */pname* `*RUNTIME ERROR*` *xxxx* `@` *yyyy* where:

pname is the name of the program where the error was encountered.

xxxx is the error code; it can take one of three forms:

nnnn four-digit numeric error code
EOF end-of-file error
nnaa intrinsic routine error
nn = routine number
aa = `OF` integer or floating-point overflow.
`UN` operation is undefined for this argument, for example `log(-1.0)`.
`OR` operation is defined but is computationally unfeasible for this argument, for example `sin(1e22)`.

yyyy is the address within the program at which the error occurred. The module and line number can be determined using a load map and a compilation listing with the Q option.

zz is the CDS code segment in which the error occurred.

When a CDS subprogram causes an error, a traceback is done after the above message is printed. The traceback prints the name of the module that caused the error and the point where the error occurred, followed by the name of the routine that called it, and the point of the call, and so on, tracing the call back to the main program. For example:

```
(Error handler) <- BAD_ROUTINE+22 <- ITS_CALLER+33 <- MAIN_PROG+444
```

The numbers are octal offsets within the code of the module. If the stack has been corrupted, the traceback may terminate early or may not be done at all. In this case, the offending routine and address can be found from the standard message, which precedes the traceback.

Table A-2 lists the library subroutine errors, the library subroutines related to each error, and the error conditions causing the errors. The parameter types referenced in Table A-2 are as follows:

r = REAL*4

x = EXTENDED PRECISION (REAL*6)

d = DOUBLE PRECISION (REAL*8)

i = INTEGER*2

j = DOUBLE INTEGER (INTEGER*4)

c = COMPLEX*8 or COMPLEX*16, (real(*c*), imag (*c*))

Generic forms are not listed; for example, LOG (*r*) is listed as ALOG(*r*), SQRT(*d*) is listed as DSQRT(*d*), and *r***d* is listed as *d***d*.

Table A-2. Library Subroutine Errors

Error Message	Expression Used In Program	Error Condition

02-UN	ALOG(r)
	ALOG10(r)
	CLOG(c)
	DLOG(d)
	DLOG10(d)
		$r \leq 0$
		$r \leq 0$
		$c = (0,0)$
		$d \leq 0$
		$d \leq 0$

03-UN	SQRT(r)
	DSQRT(x)
	DSQRT(d)
		$r < 0$
		$x < 0$
		$d < 0$

04-UN	r**r	base = 0, exponent ≤ 0 or base < 0, exponent $\neq 0$

05-OR	SIN(r)
	COS(r)
	CSIN(c)
	CCOS(c)
	CEXP(c)
		r or real(c) outside $[-8192 \times \pi, +8191.75 \times \pi]$

	DSIN(d)
	DCOS(d)
		d outside $[-2^{23}, +2^{23}]$

06-UN	r**i	base = 0, exponent ≤ 0
06-OR	r**j	exponent outside $[-32768, +32767]$

07-OF	EXP(r)
	DEXP(d)
	EXP(c)
		r, d, or real(c) > 88.03

	r**r d**d	overflow
08-UN	i**i j**j	base=0, exponent ≤ 0
08-OF	i**i, j**j	overflow

09-OR	TAN(r)
	DTAN(x)
		r or x outside $[-8192 \times \pi, +8191.75 \times \pi]$

	DTAN(d)	d outside $[-2^{23}, +2^{23}]$

Table A-2. Library Subroutine Errors (continued)

Error Message	Expression Used In Program	Error Condition
10-OF	DEXP(x)	$x > 88.03$
	x**x	overflow

11-UN	DLOG(x)
	DLOG10(x)
		$x \leq 0$

12-UN	x**i d**i	base = 0, exponent ≤ 0
13-UN	x**x d**d	base < 0 or base = 0, exponent ≤ 0
14-UN	c**i	base = (0,0), exponent ≤ 0
15-UN	DATAN2($d1$,$d2$)	$d1 = d2 = 0$
21-UN	ASIN(r)	$r > 1$
22-UN	ACOS(r)	$r > 1$
23-OR	SINH(r)	$r > 88.722839$

	CSIN(c)
	CCOS(c)
		imag(c) > 88.722839

24-OR	COSH(r)	$r > 88.722839$
26-UN	ACOSH(r)	$r < 1$
27-UN	ATANH(r)	$r \geq 1$
31-UN	DASIN(d)	$d > 1$
32-UN	DACOS(d)	$d > 1$
33-OR	DSINH(d)	$d > 88.722839$
34-OR	DCOSH(d)	$d > 88.722839$
36-UN	DACSH(d)	$d < 1$
37-UN	DATNH(d)	$d > 1$
41-OR	CTAN(c)	real(c) outside $[-4096 \times \pi, +4095.875 \times \pi]$
600	Character assignment statement	Left and right sides of character assignment overlap.
601	Character substring	Substring first or last character position out of bounds.

Input/Output Run-Time Errors

If IOSTAT=*ios* is present, the input/output error code is stored in *ios.* If END= is present and an EOF error occurs, or ERR= is present and any other error occurs, control transfers to *label*, where the program can decode and handle the error if desired. If an error occurs but IOSTAT and END= or ERR= are not present, the program is aborted with a run-time error.

The input/output run-time errors returned in the IOSTAT variable or displayed on the user's terminal are listed in Tables A-3 through A-8.

Tables A-4 through A-8 include most values currently possible for IOSTAT in a FORTRAN 77 program. Some of the errors do not apply to all operating systems. Refer to the appropriate system reference manual for applicable system error messages and information about FMP, DS, and EXEC.

Sometimes FMP and DS return very large error numbers. This causes FORTRAN 77 to generate large error numbers that differ from the FMP number. For example, DS file error −999 causes run-time error 1499. Although positive FMP error numbers do not normally occur, when they do occur they can cause run-time error numbers that are less than 500. If a very unusual error number appears, it is probably due to a mismatch between the system RPL file and microcode. See your System Manager.

Note that errors 600 and 601 are not I/O errors, but character string errors. See Table A-5.

Table A-3. Input/Output Run-Time Errors

−1	**IOSTAT MEANING**	An EOF was read on a sequential file, or the end of an internal file was reached.
450	**IOSTAT MEANING**	Invalid FORTRAN unit number (less than zero) or system unit number (greater than 255).
451	**IOSTAT MEANING**	Unrecognized STATUS value; legal values in an OPEN statement are OLD, NEW, SCRATCH, and UNKNOWN.
452	**IOSTAT MEANING**	No file name (FILE=) given; file names are required when STATUS is OLD or NEW.
453	**IOSTAT MEANING**	File name (FILE=) supplied; no file names are allowed when STATUS is SCRATCH.
454	**IOSTAT MEANING**	Unrecognized ACCESS value; legal values are SEQUENTIAL, DIRECT, and BLOCKS.
455	**IOSTAT MEANING**	Unrecognized FORM value; legal values are FORMATTED and UNFORMATTED.

456 **IOSTAT MEANING** The value for MAXREC, RECL, BUFSIZE, or all is negative or zero. These parameters must have positive values.

457 **IOSTAT MEANING** Unrecognized BLANK value; legal values are NULL and ZERO.

458 **IOSTAT MEANING** The maximum number of scratch files, 99, were in use, so this OPEN of another scratch file could not be done.

459 **IOSTAT MEANING** This file has already been opened and connected to a different unit; a file can be connected to only one unit at a time.

460 **IOSTAT MEANING** The OPEN specified direct access, but the file to be opened was sequential access (not type 1 or 2).

461 **IOSTAT MEANING** The OPEN specified sequential access, but the file to be opened was direct access (type 1 or 2).

462 **IOSTAT MEANING** The file was not found, and STATUS was OLD. Same as error 506.

463 **IOSTAT MEANING** Unrecognized STATUS value; legal values in a CLOSE statement are KEEP and DELETE.

464 **IOSTAT MEANING** An ENDFILE was attempted on a direct access file; such files do not have EOFs and ENDFILE is illegal.

465 **IOSTAT MEANING** Invalid file name given in FILE=. Same as error 515.

466 **IOSTAT MEANING** All connections specified in the $FILES are in use; no more OPENs can be done until a CLOSE is done.

467 **IOSTAT MEANING** All disk file connections specified in the $FILES directive are in use; no more disk file OPENs can be done until a disk file CLOSE is done.

468 **IOSTAT MEANING** The record length given in RECL does not match the actual record length of the file.

469 **IOSTAT MEANING** The file position given in FFLOC is odd; only even byte positions are allowed, because the file system cannot position a file to an odd byte position.

470 **IOSTAT MEANING** Unrecognized USE value; legal values are EXCLUSIVE, NONEXCLUSIVE, and UPDATE.

471 **IOSTAT MEANING** The system unit number used is not accessible from this program; that is, it is not in the session SST.

472 **IOSTAT MEANING** Could get neither read nor write permission for this file.

473 **IOSTAT MEANING** (not currently used)

474 **IOSTAT MEANING** A record number (REC=) was supplied in the READ or WRITE, but the unit was not connected to a direct access file.

475 **IOSTAT MEANING** A RECL value must or must not be supplied. RECL must be used *only* if ACCESS is DIRECT, or BLOCKS.

476 **IOSTAT MEANING** (not currently used)

477 **IOSTAT MEANING** A NODE value was supplied (other than −1), but the $FILES directive did not contain the DS keyword (FMGR file system only).

478 **IOSTAT MEANING** An OPEN of a unit that was already connected tried to change attributes other than BLANK.

479 **IOSTAT MEANING** An OPEN was tried with $FILES 0,0; or the library routines to support OPEN were not loaded correctly.

480 **IOSTAT MEANING** A CLOSE was tried with $FILES 0,0; or the library routines to support CLOSE were not loaded correctly.

481 **IOSTAT MEANING** An INQUIRE was tried with $FILES 0,0; or the library routines to support INQUIRE were not loaded correctly.

482 **IOSTAT MEANING** The library routines to support BACKSPACE, ENDFILE, and REWIND were not loaded correctly.

483 **IOSTAT MEANING** This INQUIRE statement tried to inquire about a disk file, but the $FILES directive did not specify any disk connections. At least one disk connection must be specified, even if no OPEN of a disk file will be done.

484 **IOSTAT MEANING** This OPEN statement tried to open a disk file, but the $FILES directive did not specify any disk connections.

485 **IOSTAT MEANING** This OPEN statement specifies a direct access file or uses RECL, but the file name supplied is an LU number.

486 **IOSTAT MEANING** Attempt to use DNODE, which is illegal in FORTRAN 77 programs. Use an OPEN statement to connect to a remote unit.

487 **IOSTAT MEANING** ZBUF, ZLEN, or secondary/tertiary address supplied in this READ or WRITE statement, but unit is connected to a disk file.

488 **IOSTAT MEANING** REC supplied in this READ or WRITE statement is negative.

489 **IOSTAT MEANING** (not currently used)

490 **IOSTAT MEANING** (not currently used)

491 **IOSTAT MEANING** This FORMAT has an invalid field width (*w*), number of digits (*d*), minimum number of digits (*m*), or size of exponent output (*e*).

492 **IOSTAT MEANING** This FORMAT does not begin with a left parenthesis, or has too many levels of parentheses.

493 **IOSTAT MEANING** Unrecognized format character, or use of a negative value in a format (except scale), or no conversions given in the format but I/O list was not empty.

494 **IOSTAT MEANING** Illegal character in a numeric input field.

495 **IOSTAT MEANING** Numeric input field has an ill-formed number or logical value, or an octal number is too large.

496 **IOSTAT MEANING** Discrepancy in record size, I/O list length, internal buffer size, or all. Could be due to:

- Input record (or amount that fits in internal buffer) was not large enough to satisfy an unformatted READ list. If unknown-length records are being read with an unformatted READ, the error can be trapped with IOSTAT and the actual record length recovered using ITLOG.
- Output record, as specified in unformatted WRITE list or a FORMAT, was too large to fit in the internal buffer. See library routine LGBUF in Appendix B.
- Output record, as specified in unformatted WRITE list or a FORMAT statement, was larger than the record size for the direct access file to which this unit is connected.

497 **IOSTAT MEANING** A format specifier is illegal for the type of the list item that matches it.

Table A-4 lists the FMP and DS FMP errors. The last one to two digits of the error number correspond to the absolute value of the FMP number. For example, if the FMP error number is −6, the run-time error number is 506. If the FMP error number is −16, the run-time error number is 516.

For error messages which do not appear in this table, refer to the appropriate system reference manual for applicable system error messages and information about FMP, DS, and EXEC.

Table A-4. FMP Errors and DS FMP Errors

501	**IOSTAT MEANING**	Disk error.
502	**IOSTAT MEANING**	Duplicate file name.
504	**IOSTAT MEANING**	Too many records (more than $2^{31}-1$) in a type 2 file in RTE-6/VM or RTE-A.
505	**IOSTAT MEANING**	Record length illegal.
506	**IOSTAT MEANING**	File not found.
507	**IOSTAT MEANING**	Illegal security code or illegal WRITE to LU 2 or LU 3.
508	**IOSTAT MEANING**	File OPEN or LOCK rejected.
512	**IOSTAT MEANING**	EOF or SOF error.
513	**IOSTAT MEANING**	Cartridge locked.
514	**IOSTAT MEANING**	Directory full.
515	**IOSTAT MEANING**	Illegal file name.
516	**IOSTAT MEANING**	Illegal file type.
519	**IOSTAT MEANING**	Illegal access on a system disk.
525	**IOSTAT MEANING**	Bad FCODE (internal RFAM error).

526	**IOSTAT MEANING**	Bad entry number in RFAM; DCB destroyed.
528	**IOSTAT MEANING**	Too many open DS files at remote node.
529	**IOSTAT MEANING**	Internal RFAM tables invalid.
530	**IOSTAT MEANING**	Disk not mounted to caller's session.
532	**IOSTAT MEANING**	Cartridge not found.
533	**IOSTAT MEANING**	No room on cartridge.
540	**IOSTAT MEANING**	Disk not in SST.
541	**IOSTAT MEANING**	No room in SST.
546	**IOSTAT MEANING**	Greater than 255 extents.
547	**IOSTAT MEANING**	No session LU available for SPOOL file.

Other error numbers in the range 501–999 (except 600 and 601) are unexpected FMP errors. The FMP error number is (500 – <*FORTRAN_error_number*>); for example, FORTRAN error 730 is really FMP error –230. Refer to the RTE-A or RTE-6/VM Programmer's Reference Manual for a list of all FMP errors.

Table A-5 lists the character string errors.

Table A-5. Character String Errors

600	**IOSTAT MEANING**	Left and right sides of character assignment overlap (not an I/O error).
601	**IOSTAT MEANING**	Substring out of bounds (not an I/O error).

Table A-6 lists the I/O errors. The last two digits of the error number correspond to the I/O error number. For example, if the I/O error is I005, the run-time error number is 1005.

Table A-6. I/O Errors

1000	**IOSTAT MEANING**	An illegal class number was specified. Outside table, not allocated, or bad security code.
1001	**IOSTAT MEANING**	Not enough parameters were specified.
1002	**IOSTAT MEANING**	An illegal logical unit number was specified.
1003	**IOSTAT MEANING**	Illegal EQT referenced by LU in I/O call (select code=0).
1004	**IOSTAT MEANING**	An illegal user buffer was specified. Extends beyond RT/BG area or not enough system available memory to buffer the request.
1005	**IOSTAT MEANING**	An illegal disk track or sector was specified.
1006	**IOSTAT MEANING**	A reference was made to a protected track or to unassigned LG tracks.
1007	**IOSTAT MEANING**	The driver has rejected the call.
1009	**IOSTAT MEANING**	The LG tracks overflowed.
1010	**IOSTAT MEANING**	Class get call issued while one all already outstanding.
1011	**IOSTAT MEANING**	A type 4 program made an unbuffered I/O request to a driver that did not do its own mapping.
1012	**IOSTAT MEANING**	An I/O request specified a logical unit not defined for use by this session. The format for IO12 is: SES LU =*xx* IO12 PROG ADDRESS: where: *xx* = session LU not in SST

1013 **IOSTAT MEANING** An I/O request specified an LU that was either locked to another program or pointed to an EQT that was locked to another program.

1014 **IOSTAT MEANING** An I/O request was issued with the no-suspend option.

1015 **IOSTAT MEANING** Buffer size of a type 6 program is greater than what will fit in the user map.

1016 **IOSTAT MEANING** CPU backplane failure or I/O extender timing failure.

1020 **IOSTAT MEANING** Read attempted on write-only spool file.

1021 **IOSTAT MEANING** Read attempted past end-of-file.

1022 **IOSTAT MEANING** Second attempt to read JCL card from batch input file by other than FMGR. Revise program and rerun.

1023 **IOSTAT MEANING** Write attempted on read-only spool file.

1024 **IOSTAT MEANING** Write attempted beyond end-of-file; usually, spool file overflow.

1025 **IOSTAT MEANING** Attempt to access spool LU that is not currently set up.

1026 **IOSTAT MEANING** I/O request made to a spool that has been terminated by the GASP KS command.

Table A-7 lists the DS errors that can occur during remote FMGR file access. These errors are generated only when your program was linked using the $FOLDF library and you used the $FILES directive with the DS option. The 11*xx* errors are generated when the FILE specifier in the OPEN statement is an LU number; the corresponding numbers in parenthesis (55*x*) are generated when the FILE specifier is a FMGR file name.

The last digit of the error number corresponds to the DS error number. For example, if the DS error is DS02, the error number is 1102 (552). For detailed DS error information, refer to the *DS/1000-IV User's Manual,* part number 91750-90012, or the *NS-ARPA Error Message and Recovery Manual,* part number 91790-90045.

Table A-7. DS I/O Errors

1100 (550)	**IOSTAT MEANING**	Local node is quiescent.
1101 (551)	**IOSTAT MEANING**	Communication line parity, protocol failure, 'STOP' received, cable disconnected, or other hardware error.
1102 (552)	**IOSTAT MEANING**	Communication line timeout error (DVA65 links only).
1103 (553)	**IOSTAT MEANING**	Illegal record size.
1104 (554)	**IOSTAT MEANING**	Illegal nodal address, node address not in nodal routing vector (NRV).
1105 (555)	**IOSTAT MEANING**	Request timeout.
1106 (556)	**IOSTAT MEANING**	Illegal request or monitor not active.
1107 (557)	**IOSTAT MEANING**	System table error.
1108 (558)	**IOSTAT MEANING**	Remote busy or resource unavailable.
1109 (559)	**IOSTAT MEANING**	Illegal or missing parameters.

Table A-8 lists miscellaneous run-time errors.

Table A-8. Miscellaneous Run-Time Errors

*xx*99	**IOSTAT MEANING**	Partially malformed error messages.
1999	**IOSTAT MEANING**	All RTE errors that do not have the form IO*xx* or DS*xx* and that are not FMP errors are mapped to 1999.
1000–1099	**IOSTAT MEANING**	RTE I/O errors IO00 through IO99.
1100–1199	**IOSTAT MEANING**	DS errors DS00 through DS99.

B

Intrinsic and Library Functions

This appendix lists and describes the intrinsic and library functions of FORTRAN 77 on the HP 1000.

FORTRAN 77 Intrinsic Functions

Tables B-1 through B-6 list the intrinsic functions of FORTRAN 77. The tables give the definition (and syntax, if different) 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.

In Tables B-1 through B-6, the following definitions apply:

- Real means REAL*4.
- Double means REAL*8. In 66 mode, Double can also mean REAL*6 (except where † is indicated).
- Wherever Integer appears, both INTEGER*2 and INTEGER*4 apply.
- Wherever Complex appears, both COMPLEX*8 and COMPLEX*16 apply.

Some functions (such as type conversion functions and NINT) return a type different from that of their arguments. When the result type is Integer or Double, the size of the result depends on the I, J, X, and Y compiler options. The size of the result is independent of the context in which the result is used. For example, in `K=NINT(x)`, `NINT` returns an INTEGER*2 value when the I option is used, even if `k` is type INTEGER*4.

A function can be called with either its generic name (if it has one) or its specific name. If it is called with its generic name, FORTRAN 77 determines the specific function from the argument type(s).

Table B-1. Arithmetic Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute value	abs(a) |a| [See note 5]	1	ABS	IABS ABS DABS CABS —	Integer Real Double COMPLEX*8 COMPLEX*16	Integer Real Double Real Double
Remaindering	a−int(a/b)*b mod(a,b) [See note 1]	2	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double
Transfer of sign	sign(a,b) |a| if b≥0 −|a| if b < 0	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Positive difference	dim(a,b) a−b if a > b 0 if a ≤ b	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Double precision product	dprod (a,b) a*b	2		DPROD	Real	Double
Choosing largest value	max(a,b,...)	≥2	MAX	MAX0 AMAX1 DMAX1	Integer Real Double	Integer Real Double
				AMAX0 MAX1	Integer Real	Real Integer
Choosing smallest value	min(a,b,...)	≥2	MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double
				AMIN0 MIN1	Integer Real	Real Integer
Imaginary part of a complex argument	imag (a) ai [See note 5]	1	IMAG	AIMAG —	COMPLEX*8 COMPLEX*16	Real Double
Conjugate of a complex argument	conjg(a) (ar, −ai) [See note 7]	1	CONJG	CONJG DCONJG †	Complex	Complex

† Indicates that the function is an extension to the ANSI 77 standard.

Table B-2. Bit Manipulation Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Bit test †	btest(a,b) [See notes 19 and 20]	2		BTEST †	Integer	LOGICAL*2
Bit set †	ibset(a,b) [See notes 19 and 21]	2		IBSET †	Integer	Integer
Bit clear †	ibclr(a,b) [See notes 19 and 22]	2		IBCLR †	Integer	Integer
Bit field move †	mvbits(a,b,c,d,e) [See note 18]	5		MVBITS †	Integer	None (subroutine)
Logical shift †	ishft(a,b) [See note 8]	2		ISHFT †	Integer	Integer
Circular shift †	ishftc(a,b,c) [See note 16]	3		ISHFTC †	Integer	Integer
Bit field extraction †	ibits(a,b,c) [See note 17]	3		IBITS †	Integer	Integer
Logical product †	iand(a,b)	2		IAND†	Integer	Integer
Logical sum†	ior(a,b)	2		IOR †	Integer	Integer

Exclusive OR†	ixor(a)
	ieor(a)
		2		IXOR †
				IEOR †
					Integer
					Integer
						Integer
						Integer

Complement†	not(a)	1		NOT†	Integer	Integer

† Indicates that the function is an extension to the ANSI 77 standard.

Table B-3. Character Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Conversion to character	char (a) [See note 13]	1		CHAR	Integer	Character
Conversion to integer	ichar (a) [See note 13]	1		ICHAR	Character	Integer
Length	Length of character string len(a)	1		LEN	Character	Integer
Index of a substring	index(a,b) Location of substring b in string a [See note 14]	2		INDEX	Character	Integer
Lexically greater than or equal	lge(a,b) $a \geq b$ [See note 15]	2		LGE	Character	LOGICAL*2
Lexically greater than	lgt(a,b) $a > b$ [See note 15]	2		LGT	Character	LOGICAL*2
Lexically less than or equal	lle(a,b) $a \leq b$ [See note 15]	2		LLE	Character	LOGICAL*2
Lexically less than	llt(a,b) $a < b$ [See note 15]	2		LLT	Character	LOGICAL*2

Table B-4. Numeric Conversion Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to integer INT(a) [See note 1]	1	INT	— INT IFIX IDINT —	Integer Real Real Double Complex	Integer Integer Integer Integer Integer
	Conversion to real real(a) [See note 2]	1	REAL	FLOAT — SNGL —	Integer Real Double Complex	Real Real Real Real
	Conversion to double precision dble(a) [See note 2]	1	DBLE	— — — —	Integer Real Double Complex	Double Double Double Double

	Conversion to COMPLEX *8
	cmplx(a)
	cmplx(a,b)
	[See note 4]
		1 or 2		CMPLX	Real
					[See note 4]
						COMPLEX *8

	Conversion to COMPLEX *16
	dcmplx(a)
	dcmplx(a,b)
	[See Note 4]
		1 or 2		DCMPLX †	Double
					[See note 4]
						COMPLEX *16

Truncation	aint(a) [See note 1]	1	AINT	AINT DINT	Real Double	Real Double
Nearest whole number	anint(a) aint(a+0.5) if a ≥ 0 aint(a−0.5) if a < 0	1	ANINT	ANINT DNINT	Real Double ‡	Real Double ‡
Nearest integer	nint(a) int(a+0.5) if a ≥ 0 int(a−0.5) if a < 0	1	NINT	NINT IDNINT	Real Double ‡	Integer Integer ‡

† Indicates that the function is an extension to the ANSI 77 standard.

‡ Indicates that the function is not defined for extended precision (REAL*6) arguments. For more information about extended precision, see Chapter 8.

Table B-5. Transcendental Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Sine	sin(a) [See note 6]	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Cosine	cos(a) [See note 6]	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent	tan(a) [See note 6]	1	TAN	TAN DTAN CTAN †	Real Double Complex	Real Double Complex
Arcsine	asin(a) [See note 6]	1	ASIN	ASIN DASIN	Real Double ‡	Real Double ‡
Arccosine	acos(a) [See note 6]	1	ACOS	ACOS DACOS	Real Double ‡	Real Double ‡
Arctangent	atan(a) [See note 6]	1	ATAN	ATAN DATAN	Real Double	Real Double

	atat2(a,b)
	arctan(a/b)
	[See note 12]
		2	ATAN2	ATAN2
				DATAN2
					Real
					Double
						Real
						Double

Hyperbolic sine	sinh(a)	1	SINH	SINH DSINH	Real Double ‡	Real Double ‡
Hyperbolic cosine	cosh(a)	1	COSH	COSH DCOSH	Real Double ‡	Real Double ‡
Hyperbolic tangent	tanh(a)	1	TANH	TANH DTANH	Real Double ‡	Real Double ‡
Hyperbolic arcsine	asinh(a)	1	ASINH	ASINH † DASINH †	Real Double ‡	Real Double ‡
Hyperbolic arccosine	acosh(a)	1	ACOSH	ACOSH † DACOSH †	Real Double ‡	Real Double ‡
Hyperbolic arctangent	atanh(a)	1	ATANH	ATANH † DATANH †	Real Double ‡	Real Double ‡
Square root	sqrt(a) $\sqrt{a}$	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Exponential	exp(a) e^{a}	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural logarithm	log(a)	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common logarithm	log10(a)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double

† Indicates that the function is an extension to the ANSI 77 standard.
‡ Indicates that the function is not defined for extended precision (REAL*8) arguments. For more information about extended precision, see Chapter 8.

Table B-6. Miscellaneous Functions

Intrinsic Function	Description	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Parameter count †	No. of parameters actually passed to this subroutine or function pcount() [See note 23]	0		PCOUNT		INTEGER*2
Sense switch register†	issw(a) [See note 10]	1		ISSW	INTEGER*2	INTEGER*2
EXEC REIO† DEXEC, XREIO and XLUEX	EXEC REIO [See note 11]			EXEC REIO		

† Indicates that the function is an extension to the ANSI 77 standard.

Table B-7. Compatibility Functions

Compatibility Function Name	Preferred Function Name
ALOGT	ALOG10
DLOGT	DLOG10
DATN2	DATAN2
DDINT	DINT

Notes for Tables B-1 through B-7

1. For *a* of type integer or double integer, INT(*a*)=*a*. For *a* of type real, extended precision, or double precision, there are two cases:

   a. If |*a*| < 1, int(*a*)=0;

   b. If |*a*| ≥ 1, int(*a*) is the integer whose magnitude is the largest integer less than or equal to |*a*| and whose sign is the same as the sign of *a*.

   For example:

   ```
   int(-3.7) = -3
   ```

   For *a* of type complex, INT(*a*) is the value obtained by applying the above rule to the real part of *a*.

   For *a* of type real, IFIX(*a*) is the same as INT(*a*).

2. REAL(*a*) and DBL(*a*) convert their arguments to real and double precision, respectively. The result is rounded if necessary, except that REAL(*a*), where a is double integer, drops the least significant bits of *a* if necessary to fit it in a real value.

   For *a* of type integer, FLOAT(*a*) is the same as REAL(*a*).

3. The result of the function IBCLR(*a*,*b*) is equal to the value of *a* with bit number *b* set to 0. If *b* ≥ 16/32, the result is *a*. If *b* is a constant expression ≥ 16/32, a warning is issued but the code is correct.

4. CMPLX can have one or two arguments. If there is one argument, it can be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and can be of type integer, real, or double precision.

   For *a* of type complex, CMPLX(*a*) is *a*. For *a* of type integer, real, or double precision, CMPLX(*a*) is the complex value whose real part is REAL(*a*) and whose imaginary part is 0. For *a* of type double complex, CMPLX(*a*) is:

   ```
   CMPLX(REAL(a),IMAG(a))
   ```

   CMPLX(*a*,*b*) is the complex 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, DCMPLX(*a*) is

   ```
   DCMPLX(DBLE(a),IMAG(a))
   ```

5. A complex value is expressed as an ordered pair of reals or doubles (*ar*,*ai*), where *ar* is the real part and *ai* is 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 $(ar,-ai)$; refer to note 5 above.

8. As a MIL-STD-1753 extension to the ANSI 77 standard, 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), then the result is 0. Bits shifted out from the left or right end are lost, and zeros are shifted in from the opposite end. The type of the result is the same as the type of *a*.

9. If too few parameters are passed to a CDS subprogram, the program may abort with an error. See "$OPTPARMS Directive" in Chapter 7 for further information.

10. ISSW(a) is defined to set the sign bit of the A register equal to bit *a* of the switch register. May be meaningless on some processors.

11. EXEC, REIO, XLUEX, XREIO, and DEXEC can be invoked as functions and return the contents of the A and B registers to check for error indications. Only one alternate return label can be specified. Refer to the appropriate programmer's reference manual for more information.

12. ATAN2(y,x) is in the range $[-\pi,+\pi]$ and is in the correct quadrant.

13. 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. See Appendix C for the ASCII character set.

    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.

    CHAR converts from an integer to a character, using the integer to form the internal representation of the character. The integer value should be in the range 0 to 255; if it is not, the least 8 bits are used. Note that c = CHAR(ICHAR(c)) for all characters c, and i = ICHAR(CHAR(i)) for all integers i in the range 0 to 255.

14. 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.

15. In FORTRAN 77 on the HP 1000, the intrinsic functions LGE, LGT, LLE, and LLT behave exactly the same as the operators .GE., .GT., .LE., and .LT. because the HP 1000 uses the standard ASCII character set. The intrinsics 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 ('3','A')
    ```

    is always true, but

    ```
    ('3' .LT. 'A')
    ```

    may be false on some systems.

16. As a MIL-STD-1753 standard extension to the ANSI 77 standard, ISHFTC(*a*,*b*,*c*) is defined as the rightmost *c* bits of the argument *a* left 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 32 if *a* is INTEGER*4.

17. As a MIL-STD-1753 standard extension to the ANSI 77 standard, bit fields can be extracted from a value. Bit fields 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.

    The function IBITS(*a*,*b*,*c*) extracts a field 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 are set to 0. The value of *b*+*c* must be less than or equal to 16 if *a* is INTEGER*2, or 32 if *a* is INTEGER*4.

18. As a MIL-STD-1753 standard extension to the ANSI 77 standard, the bit move subroutine 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 remains unchanged. All arguments are integer expressions, except *d*, which must be an integer variable or array element. Arguments *a* and *d* can be the same. The values of *b*+*c* and *e*+*c* must be less than or equal to the lengths of *a* and *d*, respectively.

19. 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 3, 20, and 21. 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 bit 0).

20. The function BTEST(*a*,*b*) is a logical function. Bit number *b* 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* ≥ 16/32, the result is false. If *b* is a constant expression ≥ 16/32, a warning is issued but the code is correct.

21. 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* ≥ 16/32, the result is *a*. If *b* is a constant expression ≥ 16/32, a warning is issued but the code is correct.

22. 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* ≥ 16/32, the result is *a*.

23. The PCOUNT function returns the actual number of arguments passed to the current subroutine or function. The hidden result argument in CHARACTER, DOUBLE PRECISION and COMPLEX functions is not counted. Also see the $OPTPARMS directive.

General Type Rules for Intrinsic Functions

1. If the argument type matches the function type as specified in Tables B-1 through B-7, the function maintains the same type.

   For example, if *a* is REAL*8, EXP(*a*) is a REAL*8 value.

2. If the argument type does not match the function type, the compiler default type determines the function result type (defined by the I, J, X and Y compiler options).

   For example, if compiler option J (double integer) is selected, INT(XNUM) results in a double integer value. Also, if an argument *k* is of type double integer, then the generic function ABS(*k*) results in a double integer value.

Input/Output Library Interface Functions

The following library interface functions are used in FORTRAN 77:

NFIOB
ITLOG
ISTAT
LGBUF
FPOST
ITYPE
FFRCL
FLOCF
FPOSN

The above are *not* intrinsic functions. Unless otherwise specified, they have argument and result types of INTEGER*2.

NFIOB is an integer function, requiring no arguments, that returns the number of input/output buffer blocks available for allocation. The following example allocates all the remaining buffer blocks to unit number 88:

Example

```
OPEN (88,FILE='OUTPUT',BUFSIZ=NFIOB())
```

ITLOG is a single-integer function that returns the actual number of characters read by the input/output library during its last input request. (Note that one READ statement may read more than one record and thus require more than one input request.) There must be no input/output operations between the READ and the ITLOG call.

Example

```
CHARACTER*80 line              !Reads max of 80 characters.
READ (100,'(A)',END=99) line !Determines number of characters
ichr=ITLOG()                   !actually read.
```

ISTAT is a no-argument integer function that returns the actual status of the device that was last accessed by an input/output request from the input/output library. The value of the function corresponds to the contents of the A-Register after the latest request was completed. There must be no input/output operations between the request and the ISTAT call.

Syntax

```
ISTAT ()
```

Note If the device has multiple subchannels, or a buffered WRITE was done, the ISTAT value may be meaningless.

LGBUF extends the size of the input/output buffer for binary or formatted input or output.

Syntax

```
CALL LGBUF (a,b)
```

where:

a is the new buffer address (array name).

b is the size of the buffer in words (single integer; 1 to 16383).

Example

```
DIMENSION lbuf(500), line(100)
 :
CALL LGBUF (lbuf,500)
 :
WRITE (8,'(100A2)') (line(i),i = 1,100)
```

The array `lbuf` in the example above must be statically allocated and must not be used for any other purpose.

If a segmented program or a segment of an MLS program uses LGBUF, the array must be in the root segment. In a CDS program, the array must not be on the stack, and it should be in SAVE or COMMON.

Note that when run-time error 496 occurs, the input/output buffer size may have been exceeded. A call to LGBUF to extend the size of the buffer may correct the problem. The default buffer sizes are 134 bytes for formatted input/output and 120 bytes for unformatted input/output.

For device I/O, the actual system call done by REIO will request the full record size needed to fill the buffer (as modified by LGBUF). This may cause problems with some drivers, such as the HPIB driver. Use LGBUF with device I/O carefully.

FPOST causes any buffered data to be written (posted) to the file immediately.

Syntax

```
CALL FPOST(unit,ierr)
```

where:

unit is a FORTRAN unit number.

ierr is the error number.

If the unit is connected to a disk file, FPOST posts any data written to *unit* with WRITE or PRINT and sets *ierr* equal to the FORTRAN error code. 0 is returned for no errors.

Posting means physically writing the disk buffer. This is useful when programs share a file, or to ensure the integrity of a database file. The file can reside on a remote node of a DS network.

ITYPE returns the type of the file connected to the specified unit.

Syntax

```
i = ITYPE(unit)
```

where:

i is a file type.

unit is a FORTRAN unit number.

ITYPE returns the file type associated with *unit*. 0 is returned for type 0 files, system units (devices), and unconnected units.

FFRCL sets the maximum record length for list-directed WRITE and PRINT statements. This is the largest line that a list-directed WRITE or PRINT will produce before starting a new line.

Syntax

```
CALL FFRCL(length)
```

where:

length is an integer value that specifies the record length in bytes (characters). If the length exceeds the internal buffer size (see LGBUF), it will be reduced to the internal buffer size.

The default width is 72 characters; suggested values are 79 for an 80-character terminal, or 132 for a printer.

FLOCF
FPOSN get or set the current position of the disk file that is connected to a given unit. The file need not be direct access.

Syntax

```
       {FLOCF}
CALL {FPOSN} (lu,ierr,record,position)
```

where:

lu is an INTEGER*2 FORTRAN unit number that has been connected to a disk file.

ierr is an INTEGER*2 error return variable, which is set to an IOSTAT value.

record is an INTEGER*4 record number.

position is an INTEGER*4 byte offset from the start of the file.

FLOCF sets *record* and *position* to the current file position. FPOSN sets the file position to *record* and *position*. The actual file positioning is performed using *position*; the value of *record* is used only to keep track of record numbers. If the value of *record* is incorrect, some operations such as BACKSPACE and INQUIRE (NEXTREC=) may fail.

Although the FORTRAN standard treats EOF as a record, the file system does not. FPOSN may not be used to set the (FORTRAN) file position to a point after the EOF record.

Example

```
   integer*4 recnum(100),position(100)
   character flag*10,line*80
    ⋮
   open(1000,file='/dir/filetoindex')
   i = 0
   do while (.true.)
     read(1000,'(a)',end=99) flag
     if (flag .eq.  'index next') then
       i = i + 1
       call flocf(1000,ierr,recnum(i),position(i))
       if (ierr.ne.0) stop 'FLOCF error.'
     endif
   end do
99 continue
     ⋮
   write(1,*) 'Enter index number: '
   read(1,*) index
   call fposn(1000,ierr,recnum(index),position(index))
```

```
      if (ierr.ne.0) stop 'FPOSN error.'
      read(1000,'(a)')  line
      write(1,'(a)') line
```

This program builds an index into a sequential file and randomly positions the file, on demand, to a specified indexed record. The index could have been kept in another file so that the access program would not have to build it each time. Note that the FLOCF call returns the position of the record that will be read next, not the position of the record that was just read.

Random Number Generator Functions

Three random number generator functions can be used by FORTRAN programs: URAN, GRAN, and IRANP. Each can be referred to by the function SSEED.

URAN generates uniform random numbers using a multiplicative congruence method and a register shift method asynchronously. The random numbers generated are single precision real numbers in the range 0 < *random_number* < 1.

Example

```
n = INT (100. * URAN ( ))
```

GRAN generates Gaussian (normal) random numbers with a mean of 0 and standard deviation of 1. The method used is to generate a random (chi-squared) variable and a random angle (using two calls to URAN) to yield a random normal variable. The random number generated is a single precision real number with a range of approximately −5 to +5.

Example

```
randm = GRAN ( )
```

IRANP generates Poisson-distributed random numbers that are one-word integers. The method used is to calculate exponentially distributed random numbers until the product is less than $e^{-\lambda}$. The number of exponentials needed, minus 1, is the random number returned by this function.

Example

```
irandm = IRANP (x)
```

In this example x is a user-defined real number representing λ. λ must be in the range $0 < \lambda \leq 88.72$. If λ is not in this range, the following error values are returned as the function value:

$\lambda \leq 0$	IRANP = −1
$\lambda > 88.72$	IRANP = −2

This method makes an average of λ+1 calls to URAN, so it is suggested that λ be less than 50.

SSEED can be used to seed URAN, GRAN, and IRANP. This function generates an internal number that is used by URAN, GRAN, and IRANP to start a new sequence of random numbers. If SSEED is not called, this internal value defaults to 12345.

Syntax

```
CALL SSEED (iseed)
```

where:

iseed is an arbitrary user-defined positive integer. The least significant bit of *iseed* is ignored. Therefore, an even *iseed* is the same as *iseed*+1.

If SSEED is called with the same *iseed* value each time the program is executed, or not called at all, the same sequence of random numbers is generated each time. To vary this sequence, SSEED should be called with a different *iseed* every time the program is executed.

Command Line Access Subprograms: RCPAR, RHPAR, and FPARM

The library entry points RCPAR, RHPAR, and FPARM copy command line parameters into character variables or integer arrays. Each entry point can be referenced as a subroutine or a function.

Syntax

```
CALL RCPAR(pnum,cvalue)
or
INTEGER RCPAR
LENGTH = RCPAR(pnum,cvalue)

CALL RHPAR(pnum,ivalue,plength)
or
INTEGER RHPAR
LENGTH = RHPAR(pnum,ivalue,plength)

CALL FPARM(p1,p2...pn)
```

where:

pnum is the number of the command line parameter.

cvalue is a character variable, array element, or substring.

ivalue is an integer array name.

plength is the number of characters to be copied.

p1,p2,...pn is one or more character variables, array elements, or substrings.

The specified parameters are copied to the result variables and truncated or padded with blanks on the right as required. Leading blanks are also truncated. If *pnum* is −1, the entire command line is returned.

If the specified parameter is null, the result variable is unchanged. The first time RCPAR, RHPAR, or FPARM is called, it makes an EXEC 14 call to get the command line. The complete command line is stored as part of the program's internal data structures. When an EXEC 14 call is made, the command line is consumed by the system, so subsequent EXEC 14 calls cannot access the command line. Further calls to RCPAR or RHPAR use the internal copy of the command line. A call to the system routine GETST also consumes the command line the same way as an EXEC 14 call. RCPAR, RHPAR, and FPARM do not work if the program makes an EXEC 14 or GETST call before the first RCPAR, RHPAR, or FPARM call, nor does an EXEC 14 or GETST call work after the first RCPAR, RHPAR, or FPARM call.

FPARM is shorthand for repeated RCPAR calls. For example:

```
CALL RCPAR (1,p1)
  ⋮
CALL RCPAR (1,pn)
```

For more information on EXEC 14 and GETST, see the appropriate programmer's reference manual.

See "PROGRAM Statements" in Chapter 3 for another way to get command line parameters. Using formal arguments in a PROGRAM statement is equivalent to calling FPARM.

Example

```
      PROGRAM add
      IMPLICIT NONE

C     This program gets two strings from the command line,
C     converts them to numbers using internal file reads,
C     and displays the sum of the two values.

      CHARACTER*20 a,b
      REAL x,y

      CALL fparm(a,b)           ! Get the parameters

      READ(a,*) x               ! Convert first param
      READ(b,*) y               ! Convert second param

      WRITE(1,*) x+y            ! Display SUM

      END
```

C

HP Character Set

Effect of Control Key *

000-037B | 040-077B | 100-137B | 140-177B

7 6 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				Col.	0	1	2	3	4	5	6	7
4	3	2	1	Row								
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 Lowercase

64 Character Set

96 Character Set

128 Character Set

Example: The representation for the character "K" (column 4, row 11) is

Bit	7	6	5	4	3	2	1
Binary	1	0	0	1	0	1	1
Octal	1		1			3	

Note: * Depressing the Control Key while typing an uppercase letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

Table C-1. Hewlett-Packard Character Set for Computer Systems

This table shows Hewlett-Packard'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 Scandinavian 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]	Meaning
	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	EXT	E_X	End of Text
4	002000	000004	EOT	E_T	End of Transmission
5	002400	000005	ENQ	E_Q	Enquiry
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
15	007400	000017	SI	S_I	Shift In } Character Set
16	010000	000020	DLE	D_L	Data Link Escape
17	010400	000021	DC1	D_1	Device Control 1 (X-ON)
18	011000	000022	DC2	D_2	Device Control 2 (TAPE)
19	011400	000023	DC3	D_3	Device Control 3 (X-OFF)
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]

Table C-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number Sign, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent
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
44	026000	000054	,	Comma, Cedilla
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	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Table C-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	Uppercase Letters
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	
78	047000	000116	N	
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]

Table C-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	‘	Grave Accent[5]
97	060400	000141	a	Lowercase Letters[5]
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	
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 Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

Note 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.

Note 2: Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on an HP 2640 terminal.

Note 3: Delete may be displayed as "_", "@", or space.

Note 4: Normally, the caret and underline are displayed. Some devices substitute the up arrow and the back arrow.

Note 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.

Table C-2. HP 7970B BCD-ASCII Conversion

Symbol	BCD (Octal Code)	ASCII Equivalent (Octal Code)	Symbol	BCD (Octal Code)	ASCII Equivalent (Octal Code)
(space)	20	040	@	14	100
!	52	041	A	61	101
"	37	042	B	62	102
#	13	043	C	63	103
$	53	044	D	64	104
%	57	045	E	65	105
&	†[1]	046	F	66	106
'	35	047	G	67	107
(	34	050	H	70	110
)	74	051	I	71	111
*	54	052	J	41	112
+	60	053	K	42	113
,	33	054	L	43	114
–	40	055	M	44	115
.	73	056	N	45	116
/	21	057	O	46	117
0	12	060	P	47	120
1	01	061	Q	50	121
2	02	062	R	51	122
3	03	063	S	22	123
4	04	064	T	23	124
5	05	065	U	24	125
6	06	066	V	25	126
7	07	067	W	26	127
8	10	070	X	27	130
9	11	071	Y	30	131
:	15	072	Z	31	132
;	56	073	[	75	133
<	76	074	\	36	134
=	17	075	]	55	135
>	16	076	↑	77	136
?	72	077	←	32	137

Note 1: †The ASCII code 046 is converted to the BCD code for a space (20) when writing data onto a 7-track tape.

RTE Special Characters

Mnemonic	Octal Value	Use
SOH (Control A)	1	Backspace (TTY)
EM (Control Y)	31	Backspace (2600)
BS (Control H)	10	Backspace (TTY, 2615, 2640, 2644, 2645)
EOT (Control D)	4	End-of-file (TTY, 2615, 2640, 2644, 2645)

Data Format in Memory

FORTRAN 77 has nine data types:

- Integer
- Double integer
- Real
- Double precision
- Complex
- Double complex
- Logical
- Double logical
- Character

Two further types are available as ANSI 66 compatibility extensions:

- Extended precision
- Hollerith

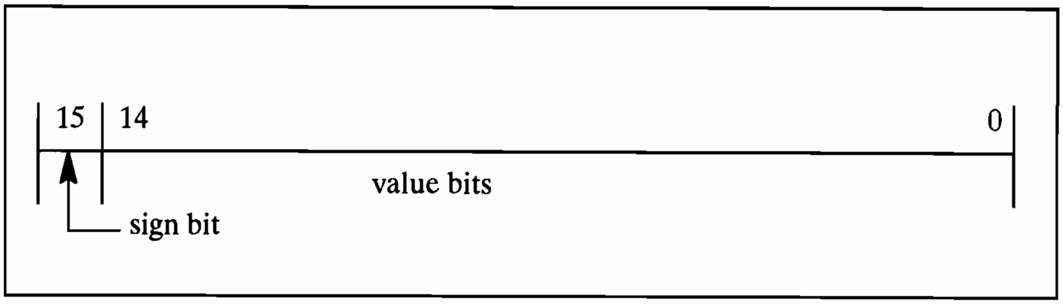
When stored in memory, each type has the format described in this appendix.

Integer Format

An integer datum is always an exact representation of a whole number.

The integer format (INTEGER*2) occupies one 16-bit word and has a range of:

-2^{15} to $+2^{15}-1$ or -32768 to $+32767$

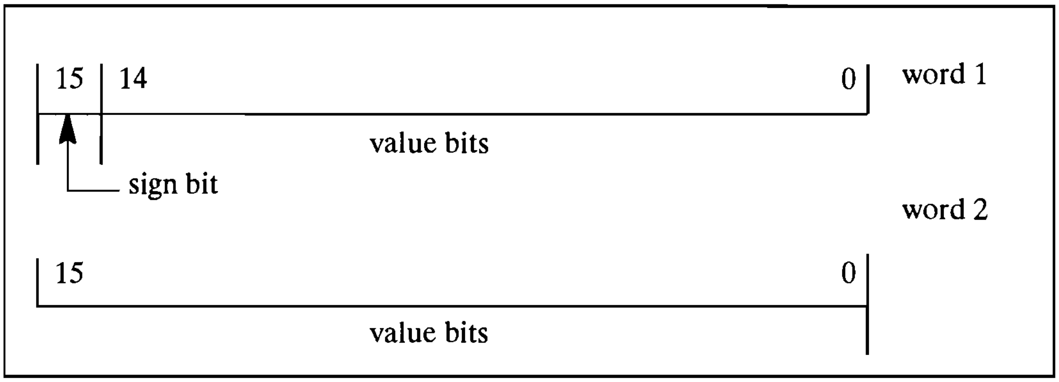


Double Integer Format

An integer datum is always an exact representation of a whole number.

The double integer format (INTEGER*4) occupies two 16-bit words and has a range of:

$$-2^{31} \text{ to } +2^{31}-1 \quad \text{or} \quad -2{,}147{,}483{,}648 \text{ to } +2{,}147{,}483{,}647$$



word 1 – most significant word, memory location M
word 2 – least significant word, memory location M+1

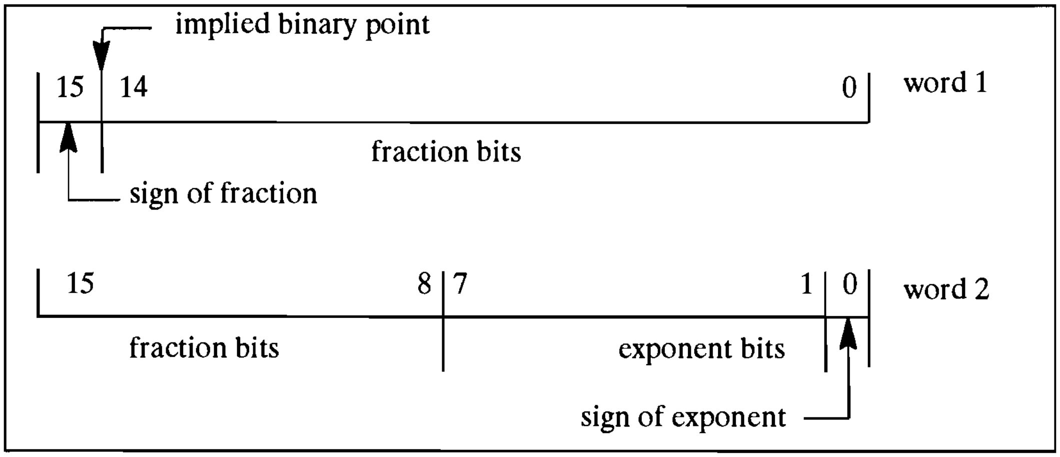
Real Format

A real datum is a processor approximation to a real number.

The real format (REAL*4) occupies two consecutive 16-bit words in memory and has an approximate range of:

$$1.47 \times 10^{-39} \text{ to } 1.70 \times 10^{38}$$

The real format has a 23-bit fraction and a 7-bit exponent. Significance is 6.6 to 6.9 decimal digits, depending upon the magnitude of the leading bits/digits in the fraction (that is, one part in $10^{6.6}$ to $10^{6.9}$).



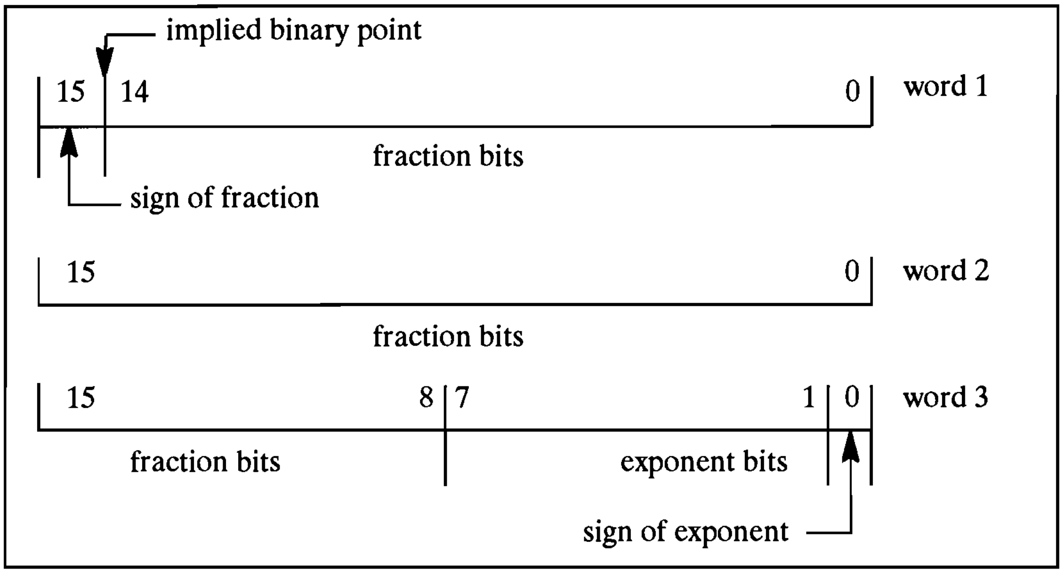
Extended Precision Format

An extended precision datum is a processor approximation to a real number.

The extended precision format (REAL*6 or DOUBLE PRECISION*6) occupies three consecutive 16-bit words in memory, and has an approximate range of:

$$1.47 \times 10^{-39} \text{ to } 1.70 \times 10^{38}$$

The extended precision format has a 39-bit fraction and a 7-bit exponent. Significance is 11.4 to 11.7 decimal digits, depending upon the magnitude of the leading bits/digits in the fraction (that is, one part in $10^{11.4}$ to $10^{11.7}$).



Extended precision is fully described in Chapter 8.

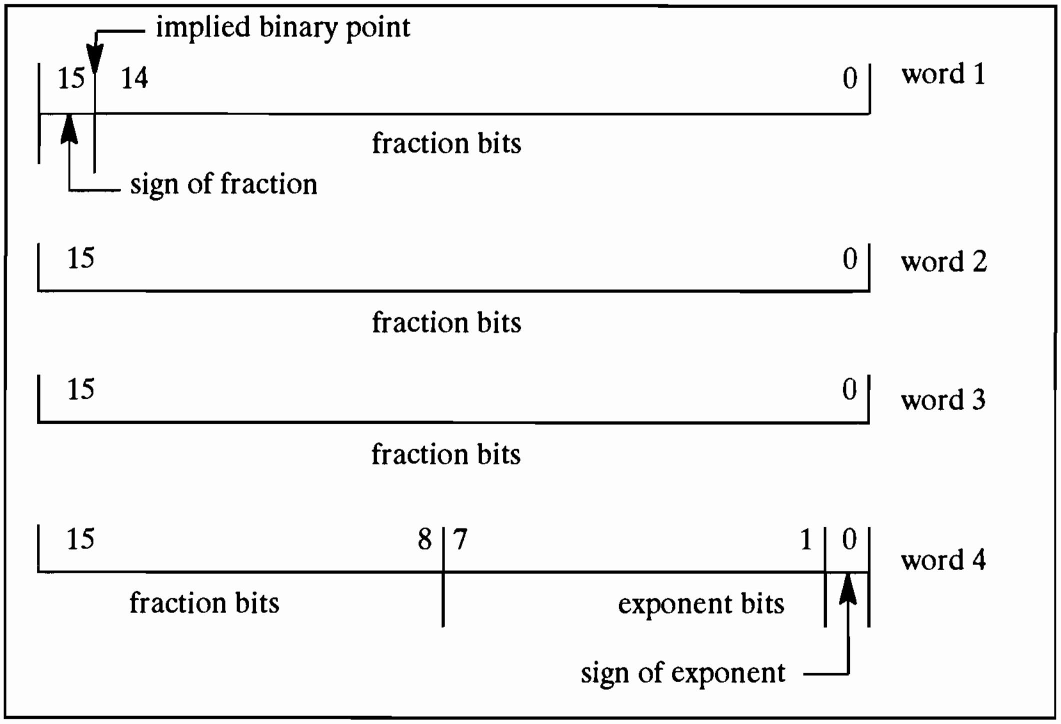
Double Precision Format

A double precision datum is a processor approximation to a real number.

The double precision format (REAL*8 or DOUBLE PRECISION) occupies four consecutive 16-bit words in memory, and has an approximate range of:

$$1.47 \times 10^{-39} \text{ to } 1.70 \times 10^{38}$$

This format has a 55-bit fraction and a 7-bit exponent. Significance to the user is 16.3 to 16.6 decimal digits, depending upon the magnitude of the leading bits/digits in the fraction (that is, one part in $10^{16.3}$ to $10^{16.6}$).



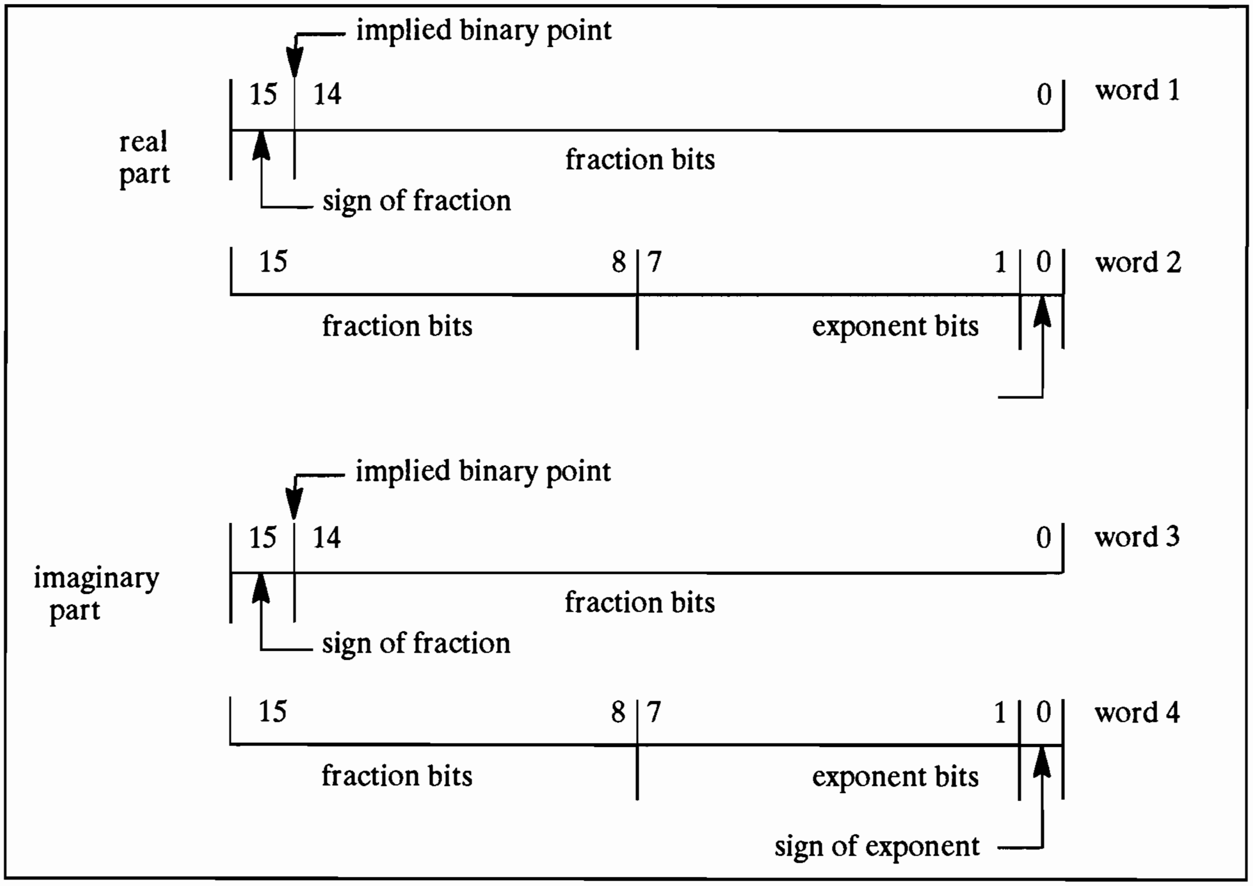
Complex Format

A complex datum is a processor approximation to a complex number.

The complex format (COMPLEX*8) occupies four consecutive 16-bit words in memory. Both the real and imaginary parts have an approximate range of:

$$1.47 \times 10^{-39} \text{ to } 1.70 \times 10^{38}$$

Both the real and the imaginary parts have 23-bit fractions and 7-bit exponents; each has the same significance as a real number.



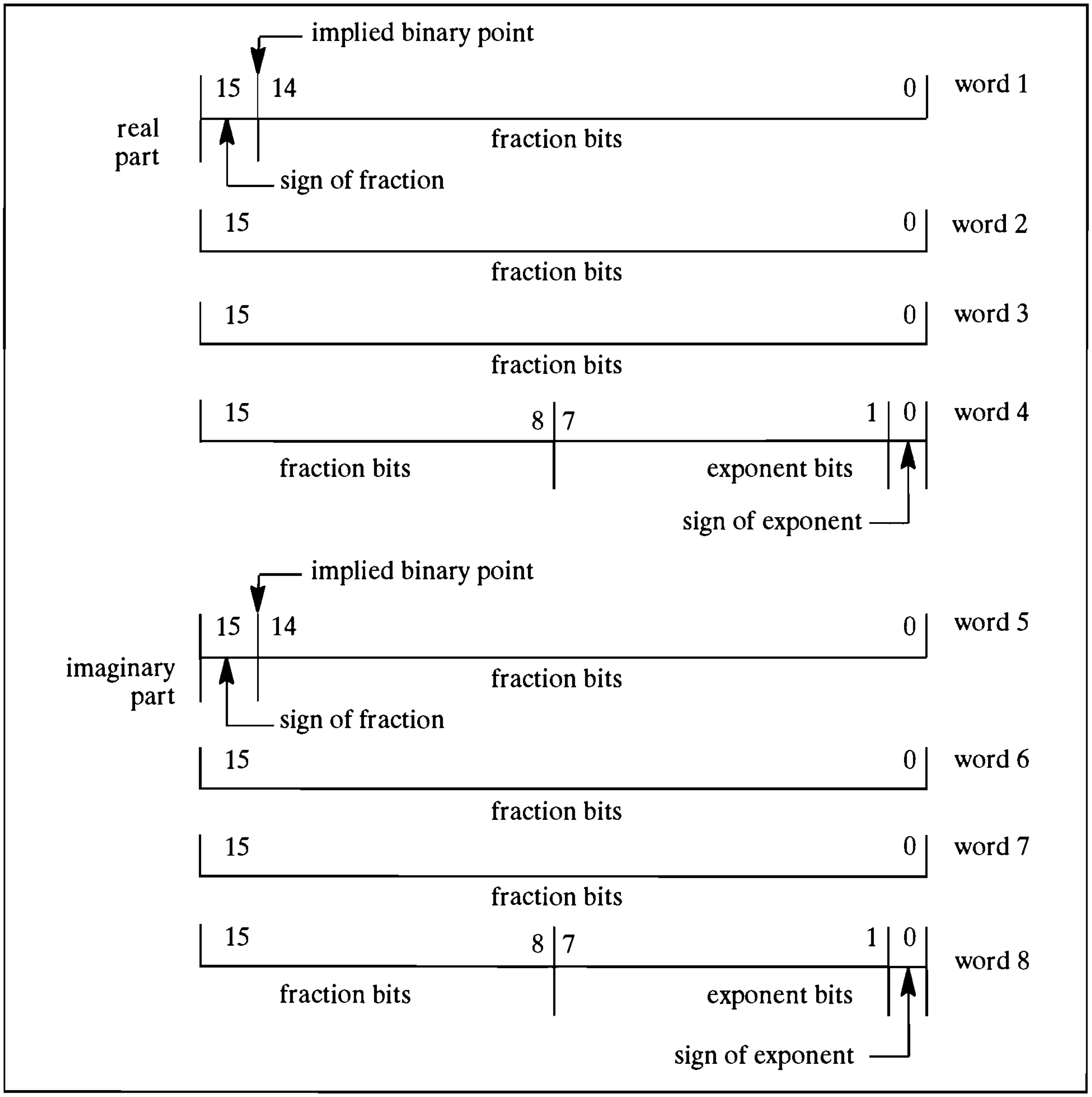
Double Complex Format

A double complex datum is a processor approximation of a complex number.

The double complex format (COMPLEX*16 or DOUBLE COMPLEX) occupies eight consecutive 16-bit words in memory. Both real and imaginary parts have an approximate range of:

$$1.47 \times 10^{-39} \text{ to } 1.70 \times 10^{38}$$

Both the real and imaginary parts have 55-bit functions and 7-bit exponents; each has the same significance as a double precision number.



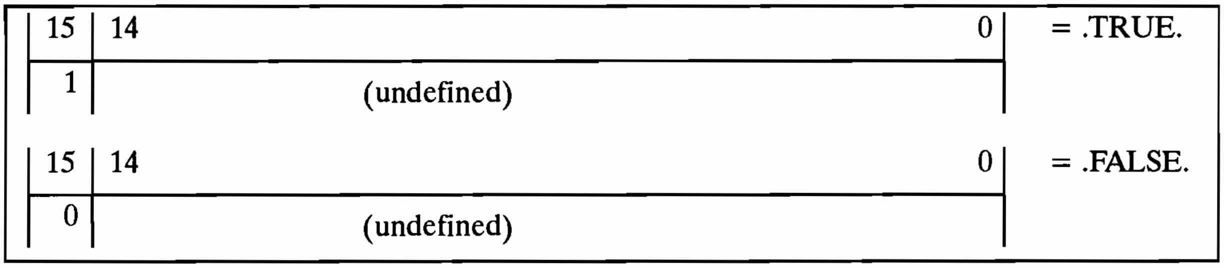
Logical Format

A logical datum is a representation of true or false, where the sign bit determines the truth value, with:

1 = true

0 = false

The logical format (LOGICAL*2) occupies one 16-bit word in memory. Bit 15 determines the truth value. The lower 15 bits are undefined.



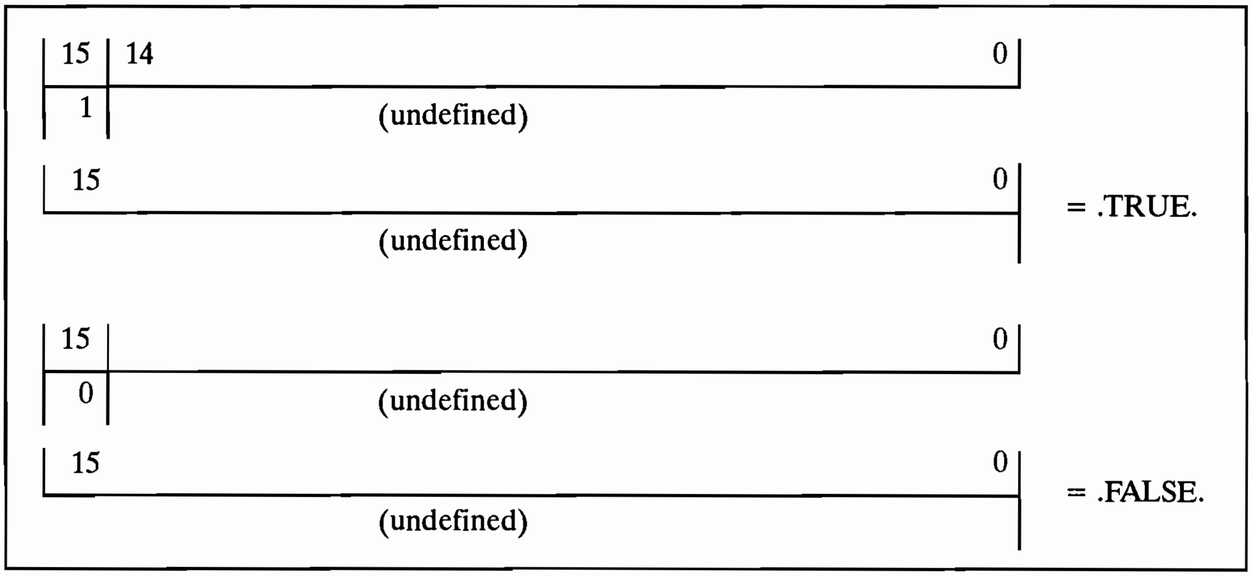
Double Logical Format

A double logical datum is a representation of true or false, where the sign bit determines the truth value, with:

1 = true

0 = false

The double logical format (LOGICAL*4) occupies two 16-bit words in memory. Bit 15 of the first word determines the truth value. The lower 31 bits are undefined.

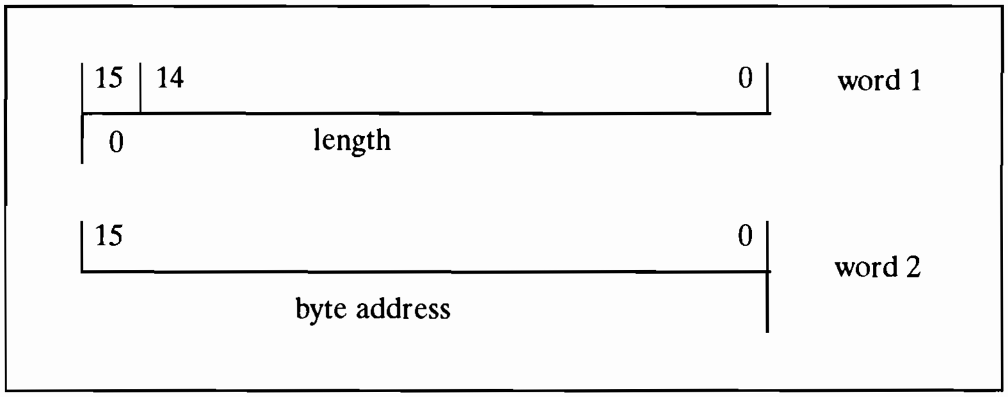


Character Format

A character datum is a character string taken from the ASCII character set. ASCII characters occupy 1 byte (8 bits) of a 16-bit word, and are packed two to a word in memory.

Character variables and constants can start or end, or both, in the middle of a word. The other byte of the word may be used by the compiler as part of another variable or constant, or it may be unused.

When character items are passed as actual arguments, the argument address points to a descriptor in the format given below. Descriptors must never be modified; only the data the descriptor points to may be changed:



Hollerith Format

Hollerith constants are described in Chapter 8. A Hollerith constant has the same format when stored in memory as a character datum. If the item is declared to have an odd number of characters, it is padded with a blank in the lower byte of the last word.

Hollerith constants start on word boundaries. When passed as actual parameters, the word address is used (no descriptor).

E

FORTRAN Comparisons

In this appendix the FORTRAN 77 compiler is first compared with the ANSI 77 standard by listing the FORTRAN 77 extensions to the standard. Then considerations affecting conversion from FORTRAN 4X to FORTRAN 77 are discussed, followed by a list of the FORTRAN 77 features that are not part of FORTRAN 4X.

Extensions to the Standard

FORTRAN 77 fully implements the ANSI 77 standard for FORTRAN. FORTRAN 77 also contains many extensions to this standard. This appendix categorizes and lists these extensions. Complete descriptions are given in the appropriate sections of this manual.

Extensions marked with an asterisk (*) are also available in FORTRAN 4X.

Extensions for Backward Compatibility

The compatibility extensions make the FORTRAN 77 compiler backward compatible with FORTRAN 4X. These extensions consist of the 66 mode features and compatibility features discussed in Chapter 8.

MIL-STD-1753 Extensions

The FORTRAN 77 compiler fully implements the Military Standard Definition (MIL-STD-1753) of extensions to the ANSI 77 standard. These extensions are:

- BLOCK DO loops using END DO.
- DO WHILE loops.
- INCLUDE statement (also $INCLUDE directive).*
- IMPLICIT NONE statement.*
- The following bit manipulation intrinsics:

```
IAND*   IOR*    IBITS   IBSET   ISHFT*  MVBITS
NOT*    IEOR*   BTEST   IBCLR   ISHFTC
```

- Octal and hexadecimal constants in DATA statements.

Other Extensions

These are the system-dependent extensions and all other extensions to the ANSI 77 standard:

- Double complex data type (as approved by the IFIP WG 2.5 Numerical Software Group).*
- Integer data type (1 word storage).*
- Logical data type (1 word storage).*
- Underscore (_) in symbolic names.
- Lowercase letters as part of the FORTRAN 77 character set.*
- Symbolic names greater than six characters.
- Equivalence of character and noncharacter items.
- Character and noncharacter items in same common block.
- Byte length specified in type statements; for example, INTEGER*4. (Byte length in CHARACTER type statements is part of the ANSI 77 standard.)*
- EMA statement.*
- Compiler directives.*
- Secondary and tertiary HP-IB addressing as part of the unit value in READ and WRITE statements.*
- The following input/output statement keywords:

```
BUFSIZ* USE*
MAXREC* ZBUF*
NODE*   ZLEN*
```

- Extended range DO loops (that is, jumping into a DO loop).*
- Formal arguments in ENTRY statements can be referenced before their first occurrence in a parameter list.
- In some cases in which output formatting overflow would occur, the scale factor, number of digits, or both are modified to produce readable results.*
- Ew.*d*E*e* and Gw.*d*E*e* formats omit the E on output if necessary to fit the exponent.
- A character expression involving concatenation of an item of type CHARACTER*(*) can be used in a relational expression.
- D*w*.d*E*e format, and the octal formats K*w*, @*w*, and O*w*.
- The logical operators, .AND., .OR., .NOT., .EQV., .NEQV., .XOR., and .EOR., can be applied to integer data to perform bit masking and manipulation.*
- An exclamation point (!) can be used to denote an end-of-line comment.*

- For all statements, there is no limit to the number of continuation lines.*
- .XOR. and .EOR. can be used the same way as the .NEQV.operator.*
- A character string (without delimiting quotes) can appear in PROGRAM, FUNCTION, SUBROUTINE, and BLOCK DATA statements.*
- A character substring can be used in an implied DO loop in a DATA statement.
- The following intrinsics are included:

```
ASINH*  ACOSH*  ATANH*  IXOR*   REIO*   DASINH*
DATANH* PCOUNT* EXEC*   ISSW*   CTAN*   XLUEX*
DEXEC*  DACOSH* XREIO*
```

Comparison of FORTRAN 4X and FORTRAN 77

FORTRAN 77 is completely backward compatible with FORTRAN 4X. To compile a FORTRAN 4X program with FORTRAN 77, change the program's control statement so that it begins with FTN66. This will ensure that the program is compiled under 66 mode. If the 66 mode default is in effect, no changes are needed to the program.

FORTRAN 77 includes many features not found in FORTRAN 4X. These features are:

- All extensions (MIL-STD-1753 and other) that are not marked with an asterisk in "MIL-STD-1753 Extensions" and "Other Extensions" above.
- 77 mode handling of ANSI 77 and FORTRAN 4X conflicts.
- Character data type.
- The following character manipulation intrinsics:

```
ICHAR   LEN     LGE     LLE     CHAR    INDEX   LGT     LLT
```

- SAVE statement.
- INTRINSIC statement.
- Implied DO loops in DATA statements.
- ENTRY statement.
- An asterisk (*) as the last upper bound in an array declaration.
- Real or double precision variable as a DO loop index.
- Constant expressions in specification statements.
- Expressions for variable dimensioning of arrays.
- CMPLX intrinsic allowing integer, double integer, real, double precision, or complex arguments.
- DPROD intrinsic.
- Block data subprograms are unnecessary in FORTRAN 77. Note: a FORTRAN 4X block data subprogram will not be found during a library search in a FORTRAN 77 program.

Cross-Reference Table

The FORTRAN 77 compiler provides a cross-reference table of symbolic names and labels used in the source program. If requested, the cross-reference table is always the last listing produced for each compiled program unit. This chapter explains how to request and read a cross-reference table and includes an example cross-reference listing of a program and its cross-reference table.

Requesting a Cross-Reference Table

To request a cross-reference table, include the optional parameter C in the FORTRAN control statement or command line. Chapter 7 describes the format and parameters of the FORTRAN control statement.

Cross-Reference Table Format

Each symbol (symbolic name or label) is printed followed by the line numbers in which the symbol appears. Multiple references in one line to the same symbol are noted. Statement labels are preceded by the pound sign (#) character.

Up to 10 line numbers are printed per line of the cross-reference table. The line numbers are listed in ascending order.

A cross-reference table is not complete for lines that contain compilation errors, since compilation terminates at the point in the line where the error is detected.

Example Program

Here is an example program that is compiled with a cross-reference table request (that is, with C in the command line). The example program is followed by the listing and cross-reference table.

```
    PROGRAM csort

C       This program builds a sorted character array of
C       names and prints them in alphabetical order.

        PARAMETER (max_names = 5)
        CHARACTER*12 name_list(max_names),

        DO i=1,mac_names

C   READ a name.

           WRITE
    iii    FORMAT('Input a name of 12 characters or less')
           READ(i,*) name

C   Move "larger" names up and insert this one in order.

           DO j=1,2,-1
               IF (name_list(j-1) .GT. name) THEN
                   name_list(j) = name_list(j-1)
               ELSE
                   GOTO 200
               END IF
           END DO
  200      name_list(j) = name
        END DO

C   Print the sorted names.

        WRITE(1,*) 'The names in alphabetical order are:'
        WRITE(1,'(2x,a12)') name_list
        END
```

```
Page 1                        Opts: 77/LCYI        Thu Nov 19, 1981 4:59 pm
                                                   CSORT.FTN::SCRATCH

     2    PROGRAM csort
     3
     4 C  This program builds a sorted character array of
     5 C  names and prints them in alphabetical order.
     6
     7       PARAMETER (max_names = 5)
     8       CHARACTER*12 name_list(max_names),
     9
    10       DO i=1,mac_names
    11
    12  C READ a name.

    14           WRITE
    15   iii     FORMAT('Input a name of 12 characters or less')
    16           READ(i,*) name
    17
    18 C  Move "larger" names up and insert this one in order.
    19
    20           DO j=1,2,-1
    21               IF (name_list(j-1) .GT. name) THEN
    22                   name_list(j) = name_list(j-1)
    23               ELSE
    24                   GOTO 200
    25               END IF
    26           END DO
    27   200     name_list(j) = name
    28       END DO
    29
    30 C  Print the sorted names.
    31
    32       WRITE(1,*) 'The names in alphabetical order are:'
    33       WRITE(1,'(2x,a12)') name_list
    34       END

Module CSORT          No errors      DATA:  25        Blank Common: None
FTN7X 5000/861229     No warnings    CODE: 214        Stack size:     53
```

```
Page 2   CSORT            Opts: 77/LCYI     Thu Nov 19, 1981 4:59 pm
                                            CSORT.FTN::SCRATCH

Cross-reference list

 Symbol          References

  #111  . . . . . . . .14    15

  #200  . . . . . . . .24    27

   I    . . . . . . . .10    20

   J    . . . . . . . .20    21   22    22   27

  MAX NAMES . . . . . . 7     8   10
  NAME            . . . 8    16   21    27

  NAME_LIST     . . . . 8    21   22    22   27    33
```

G

FORTRAN 77 Syntax Charts

The charts in this appendix describe the syntax of the FORTRAN 77 language as specified in this manual. The charts are in *railroad normal form*, designed for human readability, not as an exact specification of the syntax. For example, the description of expressions does not reflect the precedence of operators. Certain syntactic features are not represented in the charts. These features include:

- The use of blanks.
- The capability to write statements on initial lines and continuation lines.
- Compiler directives (lines beginning with a $).
- Comment lines and end-of-line comments.
- Context-dependent features, such as data type requirements, uniqueness and completeness of labels used, actual and dummy argument matching, requirements for specification statements, and restrictions on the use of statements in a particular context. Restrictions of this kind are described in the body of the manual.

If there is a discrepancy between the syntax charts of this appendix and the language as specified in the manual, the language syntax is that specified by the manual.

The charts were produced by a FORTRAN 77 program, using HP's Graphics/1000-II graphics software and an HP 9872 plotter.

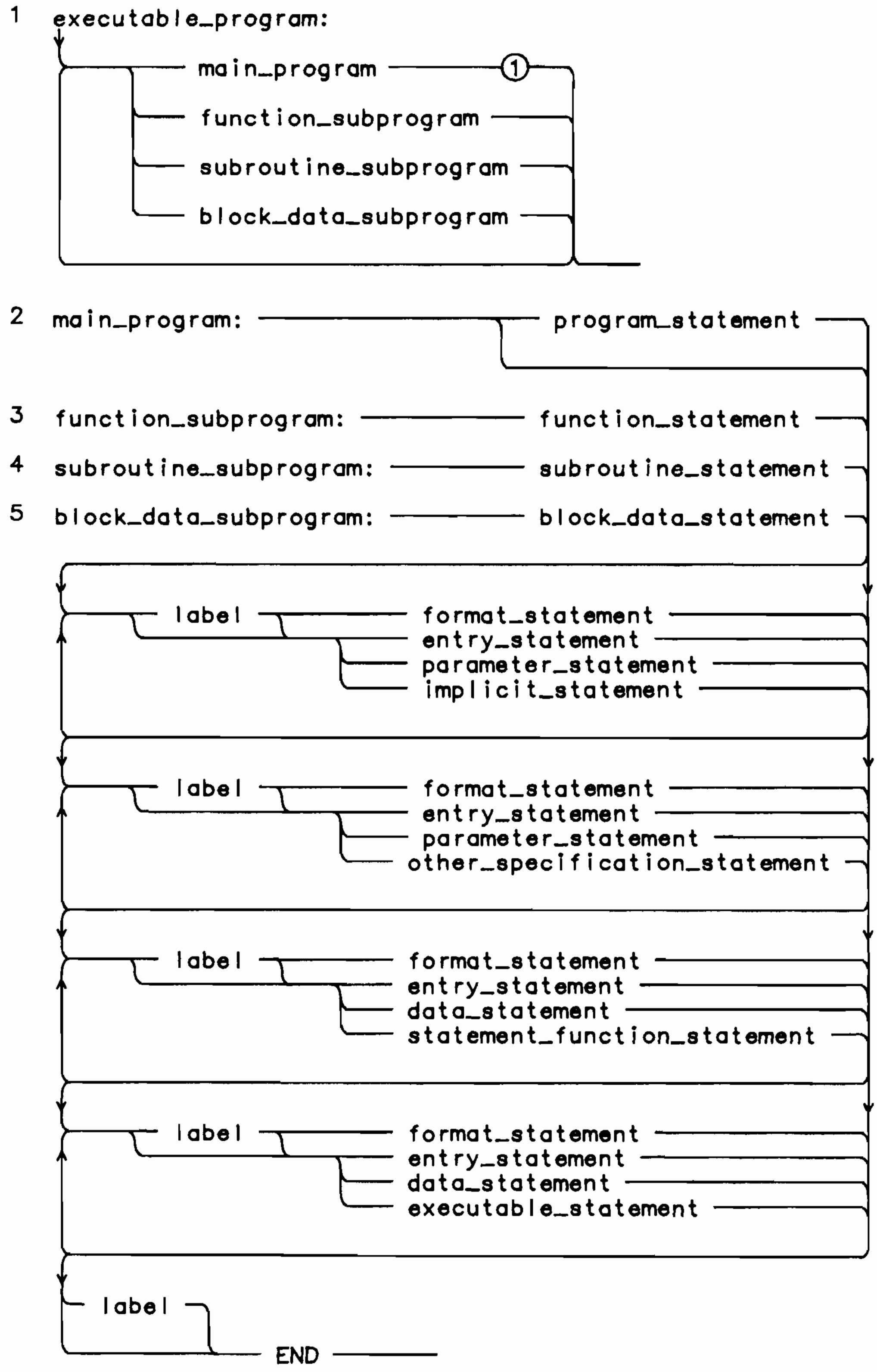
Syntax Chart Conventions

In the charts, sequences of lowercase letters and embedded underscore characters (_) represent syntactic entities. Uppercase letters and special characters must appear as written; however, uppercase or lowercase characters can be used interchangeably in programs.

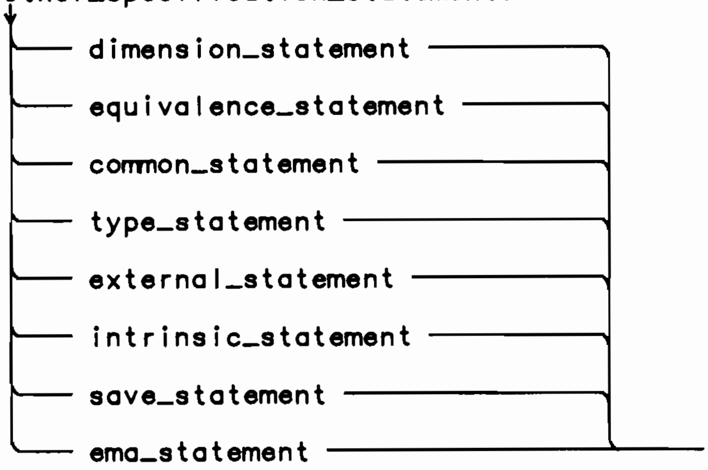
In general, names of syntactic items are identical to those used in the manual. A few names have been shortened (for example, "statement label" to "label").

The charts look like railroad tracks (hence the term "railroad normal form"). Alternative paths are specified by "switches" in the path. A number *n* in a half-circle indicates that the path can be traversed at most *n* times. A number *n* in a circle indicates that the path must be traversed exactly *n* times.

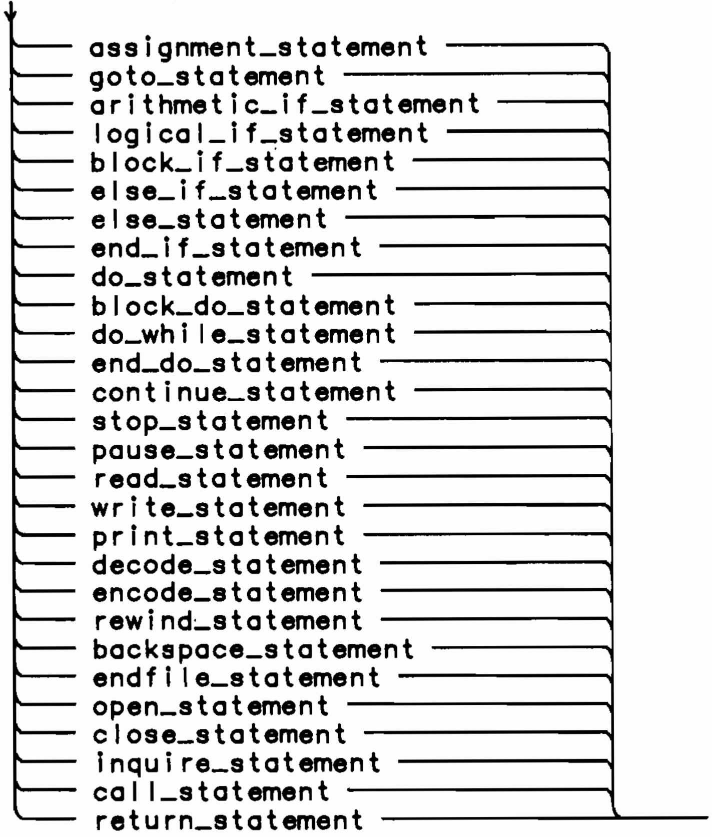
Syntax Charts:



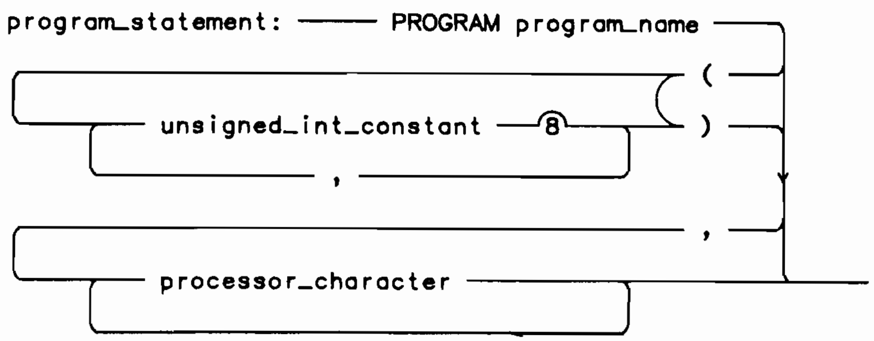
6 other_specification_statement:



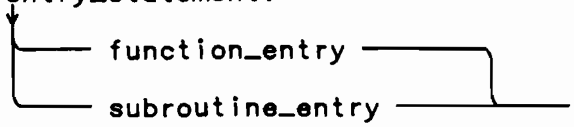
7 executable_statement:



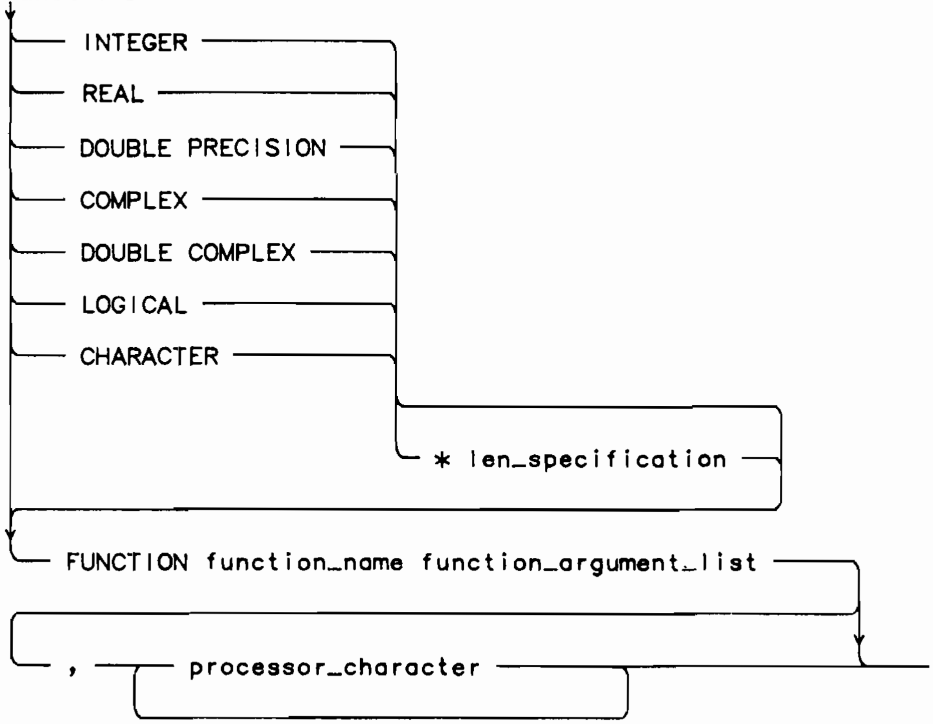
8 program_statement:



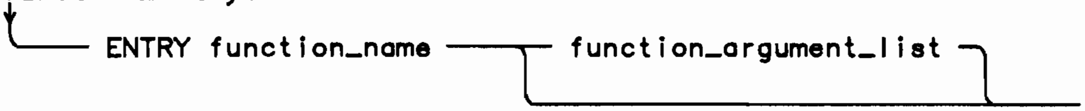
9 entry_statement:



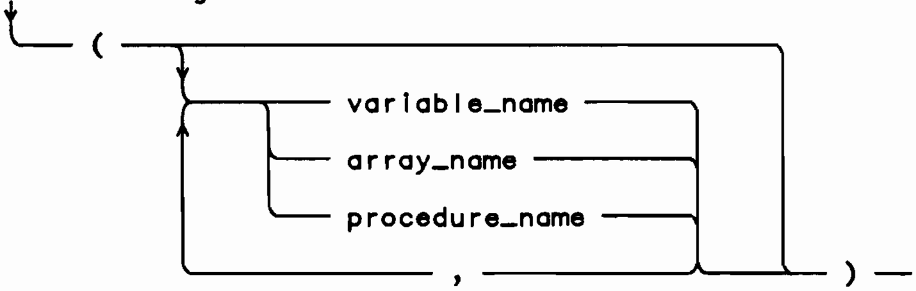
10 function_statement:



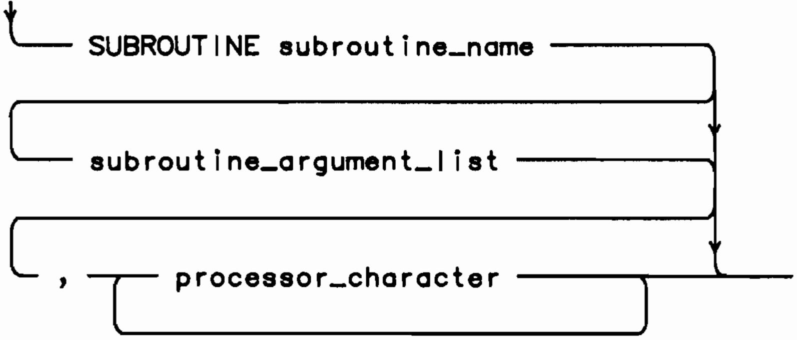
11 function_entry:



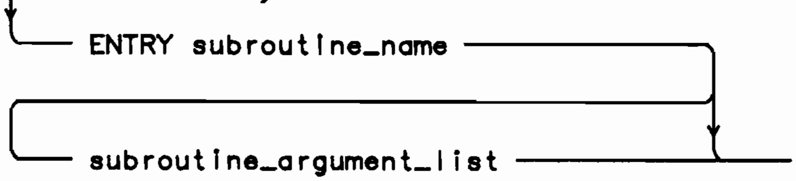
12 function_argument_list:



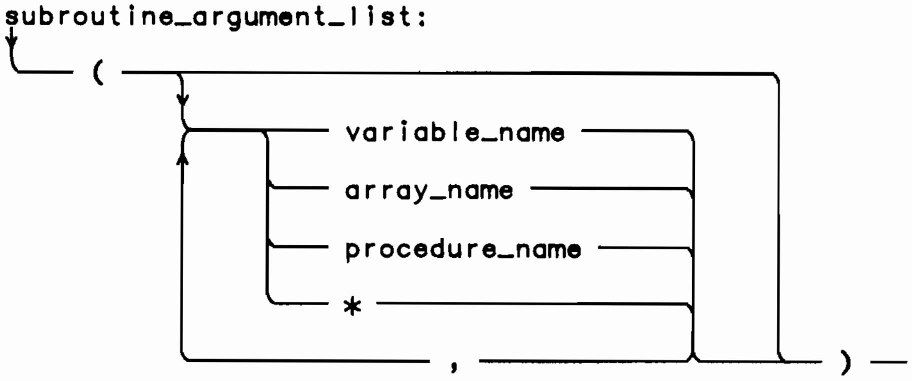
13 subroutine_statement:



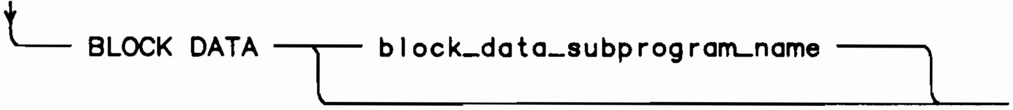
14 subroutine_entry:



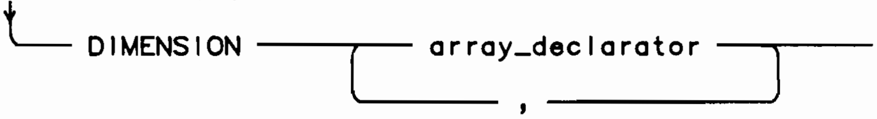
15 subroutine_argument_list:



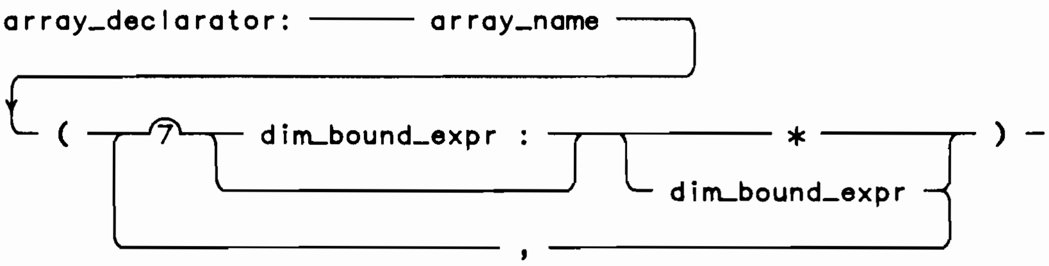
16 block_data_statement:



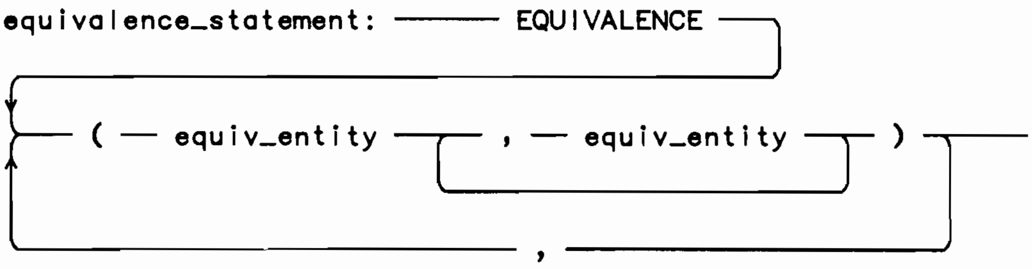
17 dimension_statement:



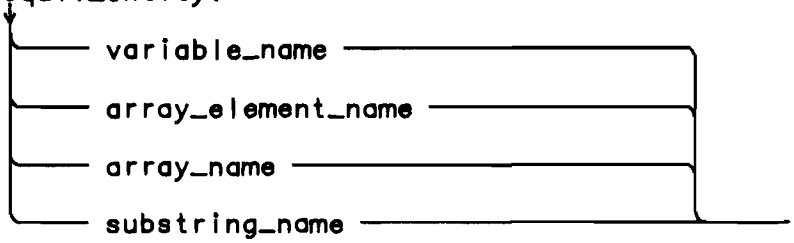
18 array_declarator:



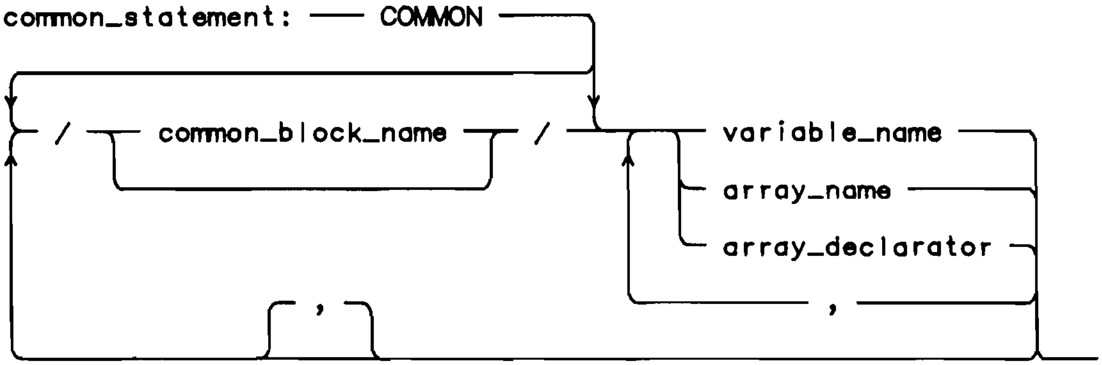
19 equivalence_statement:



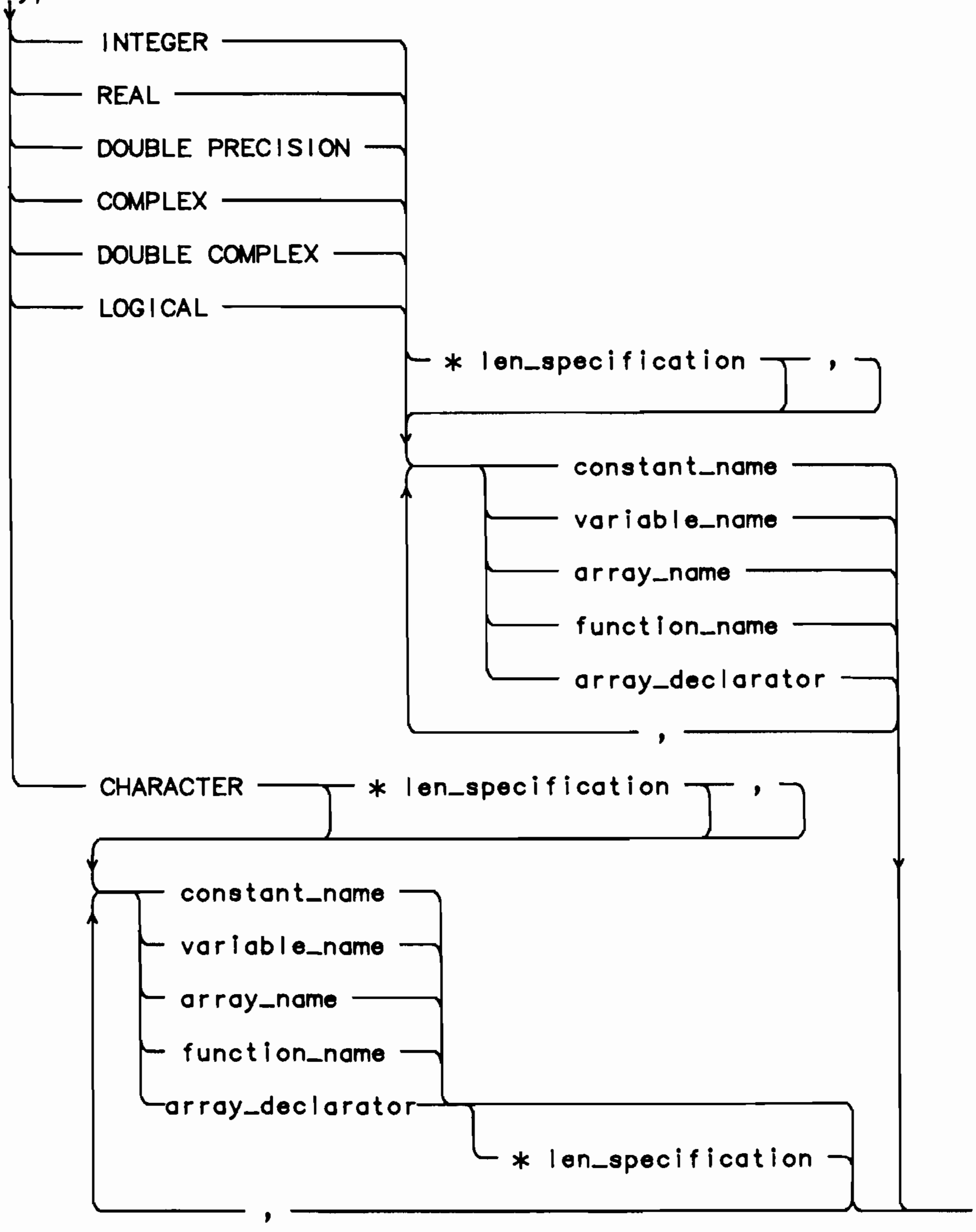
20 equiv_entity:



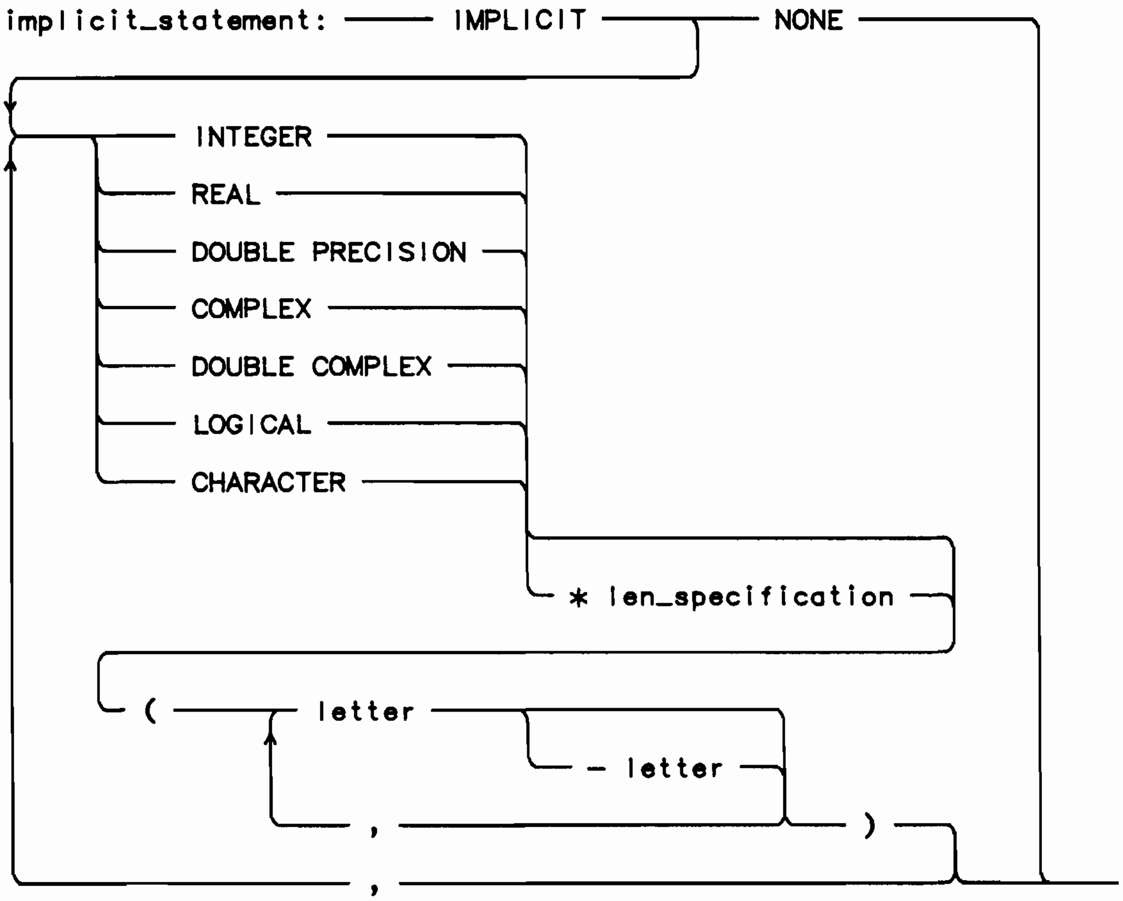
21 common_statement:



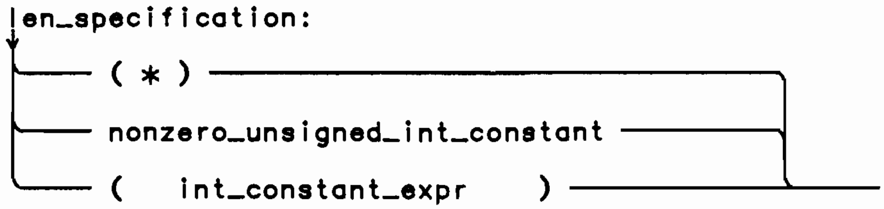
22 type_statement:



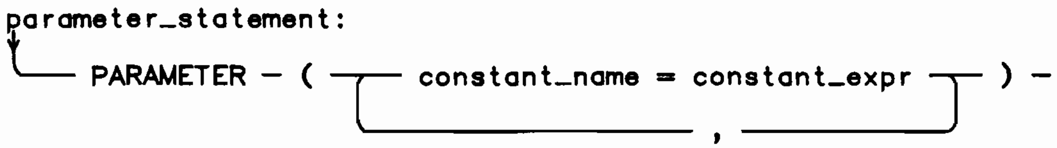
23


implicit_statement:
IMPLICIT
NONE
INTEGER
REAL
DOUBLE PRECISION
COMPLEX
DOUBLE COMPLEX
LOGICAL
CHARACTER
* len_specification
(
letter
- letter
,
)
,

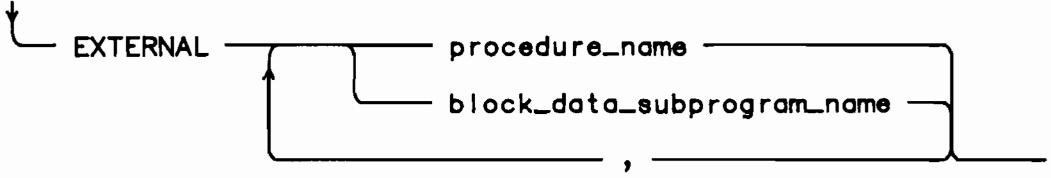
24


len_specification:
(*)
nonzero_unsigned_int_constant
(int_constant_expr)

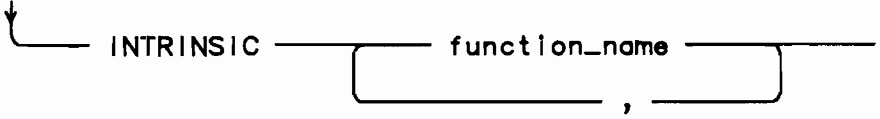
25


parameter_statement:
PARAMETER - (
constant_name = constant_expr
,
) -

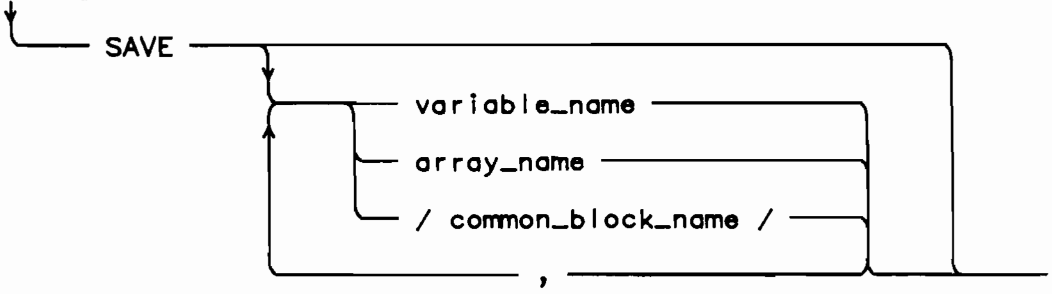
26 external_statement:



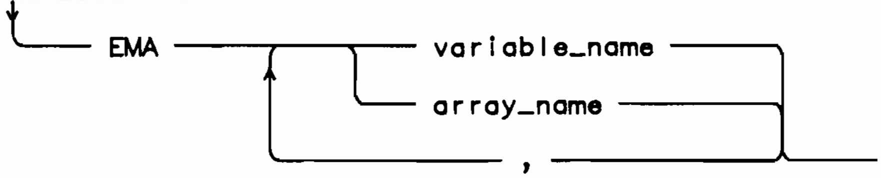
27 intrinsic_statement:



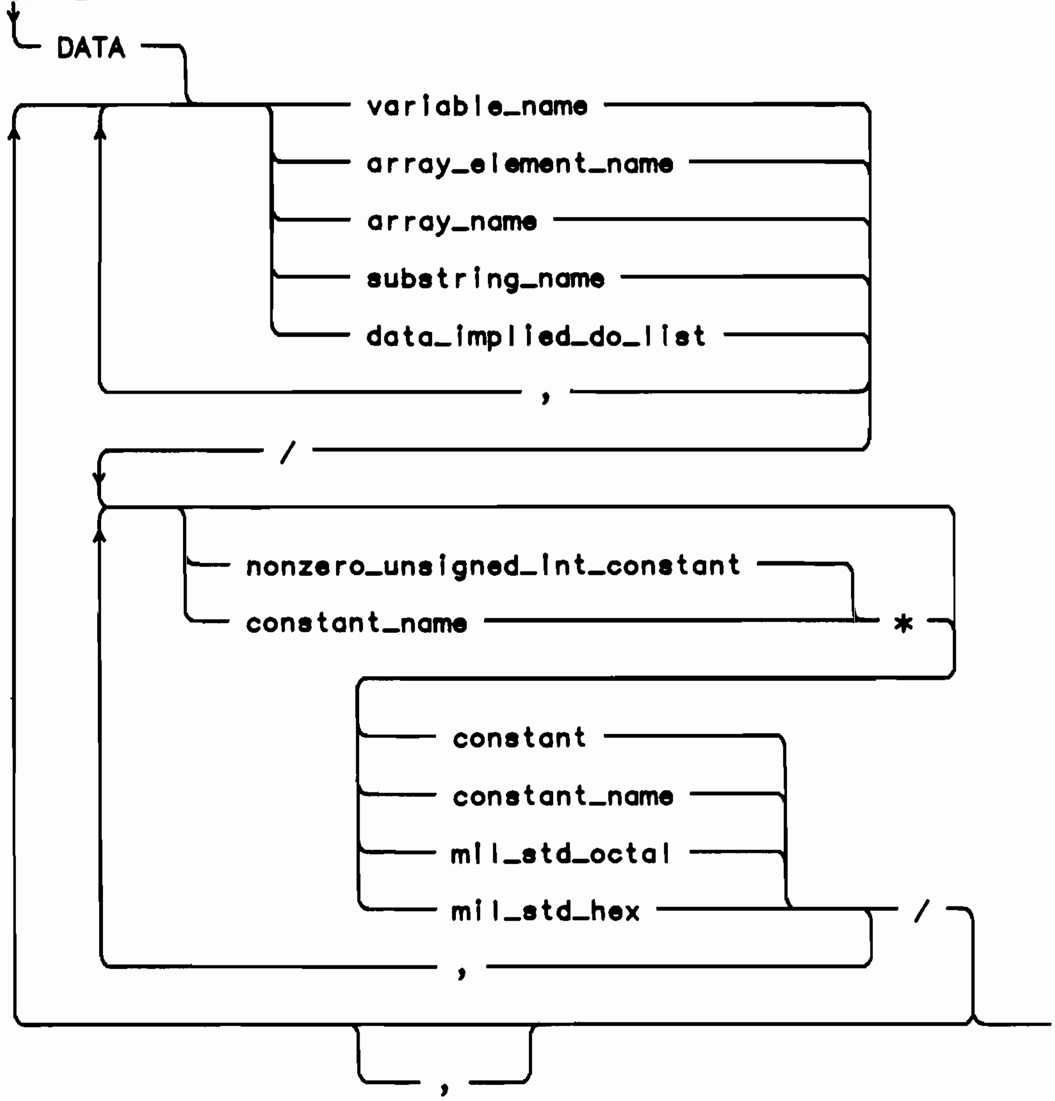
28 save_statement:



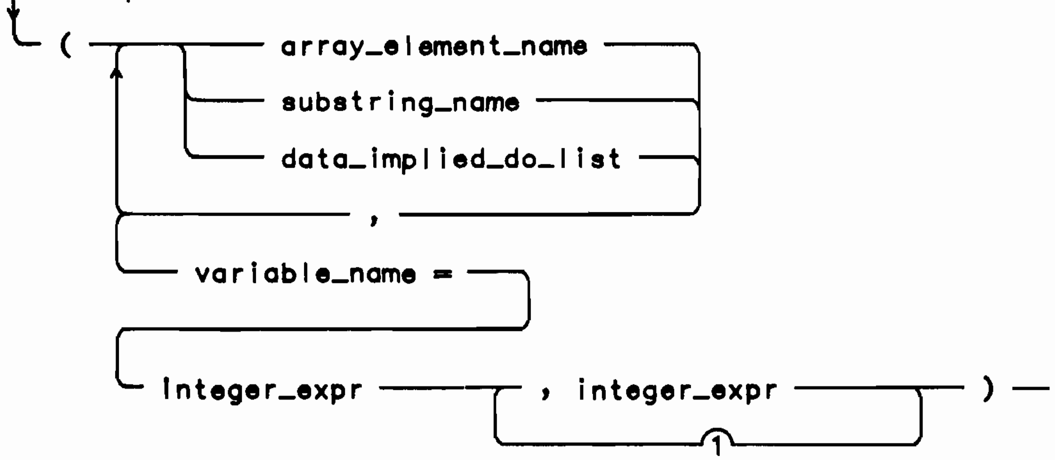
29 ema_statement:



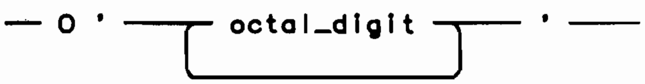
30 data_statement:



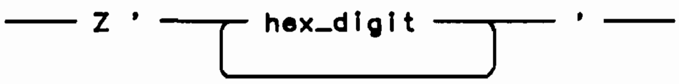
31 data_implied_do_list:



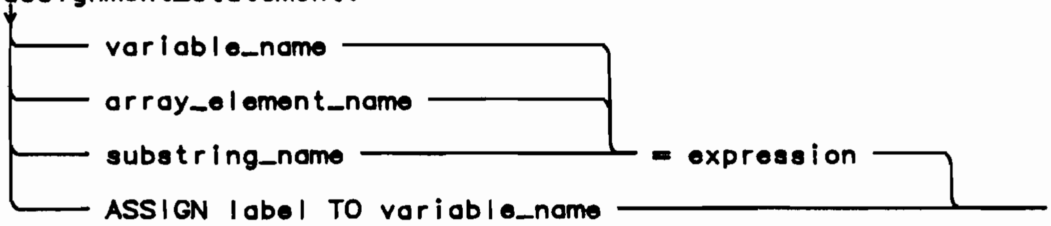
32 mil_std_octal:



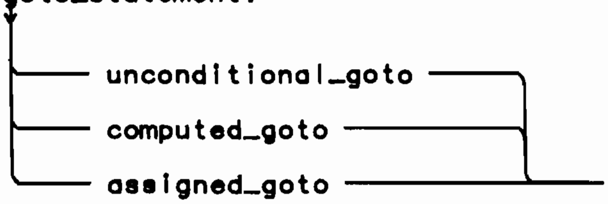
33 mil_std_hex:



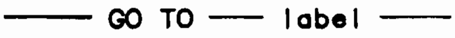
34 assignment_statement:



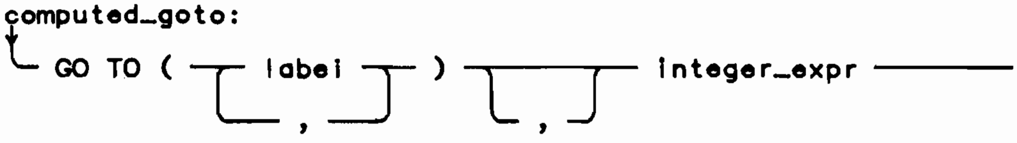
35 goto_statement:



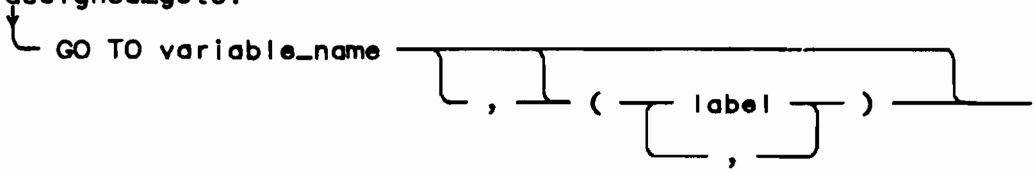
36 unconditional_goto:



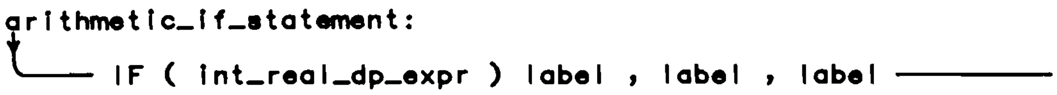
37 computed_goto:



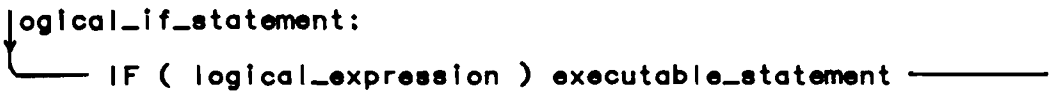
38 assigned_goto:



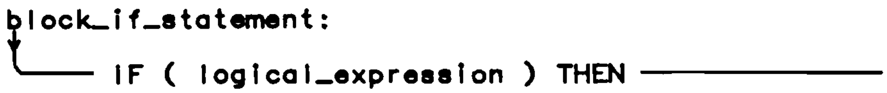
39



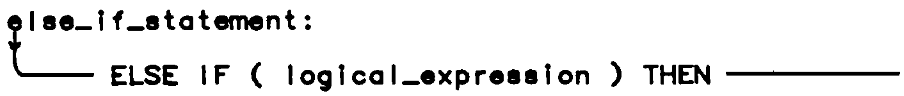
40



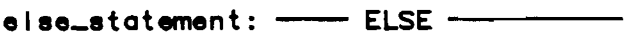
41



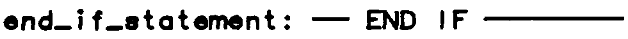
42



43

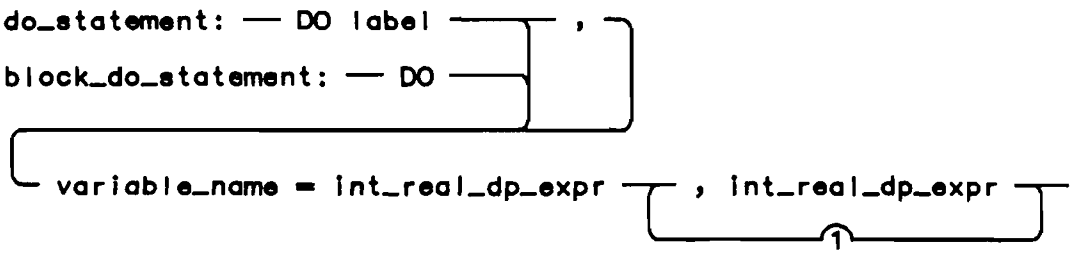


44

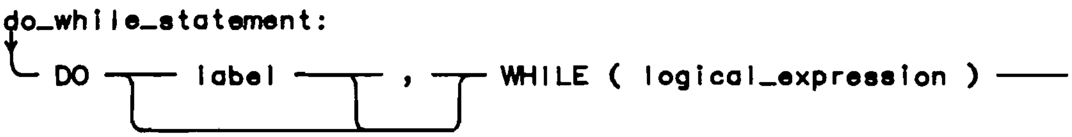


45

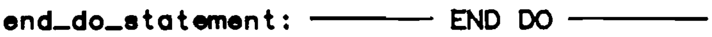
46



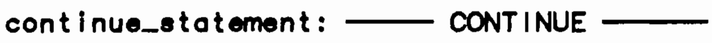
47



48

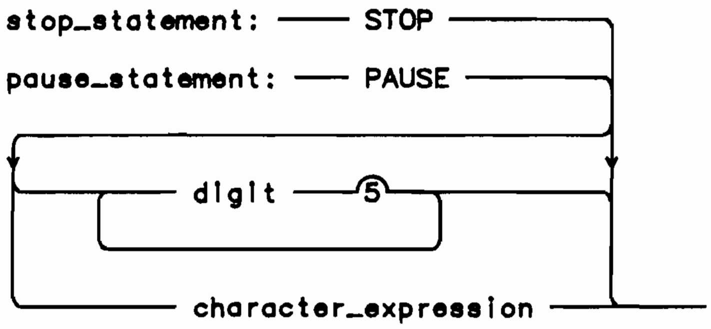


49



50

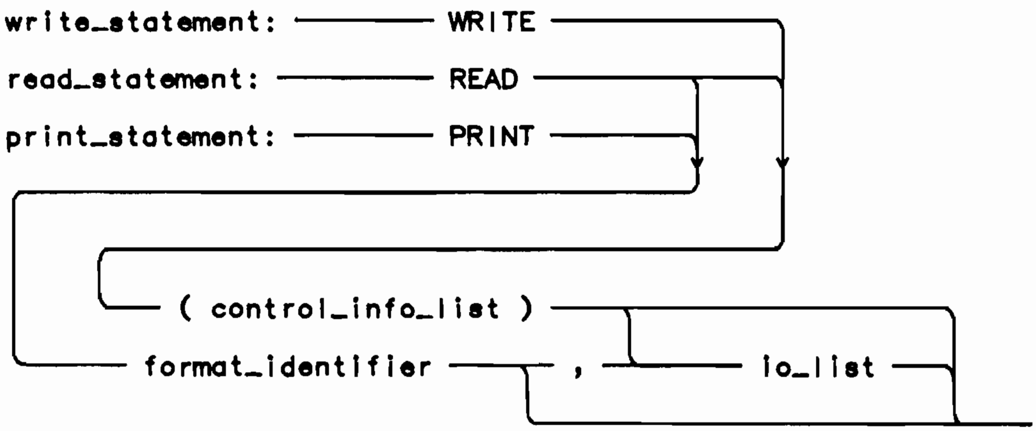
51

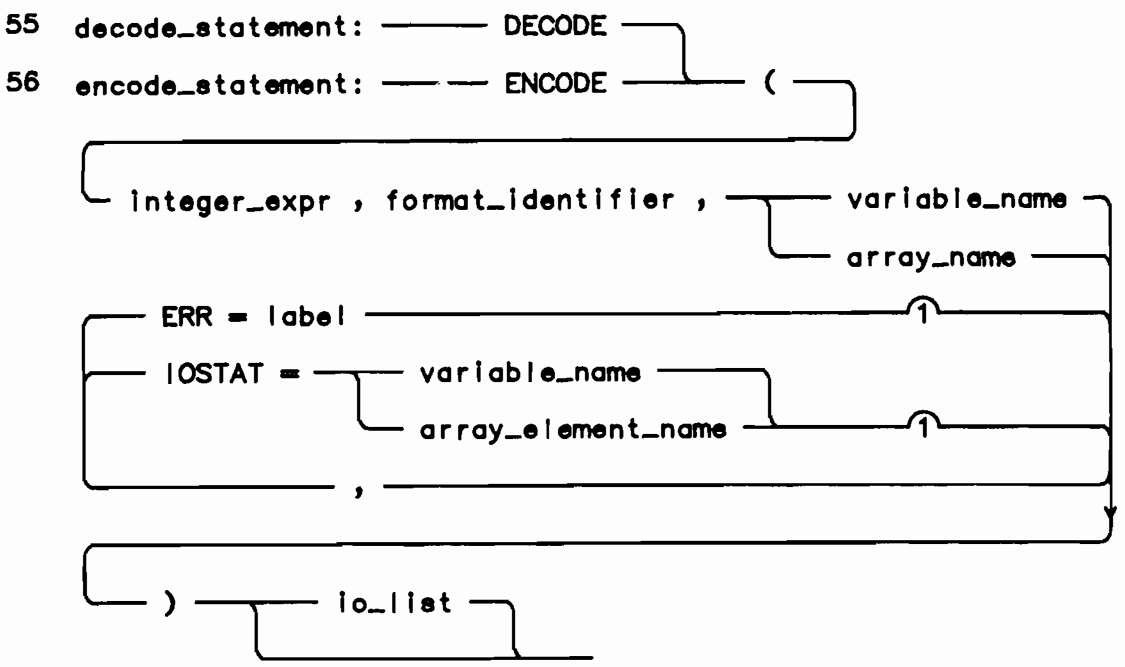


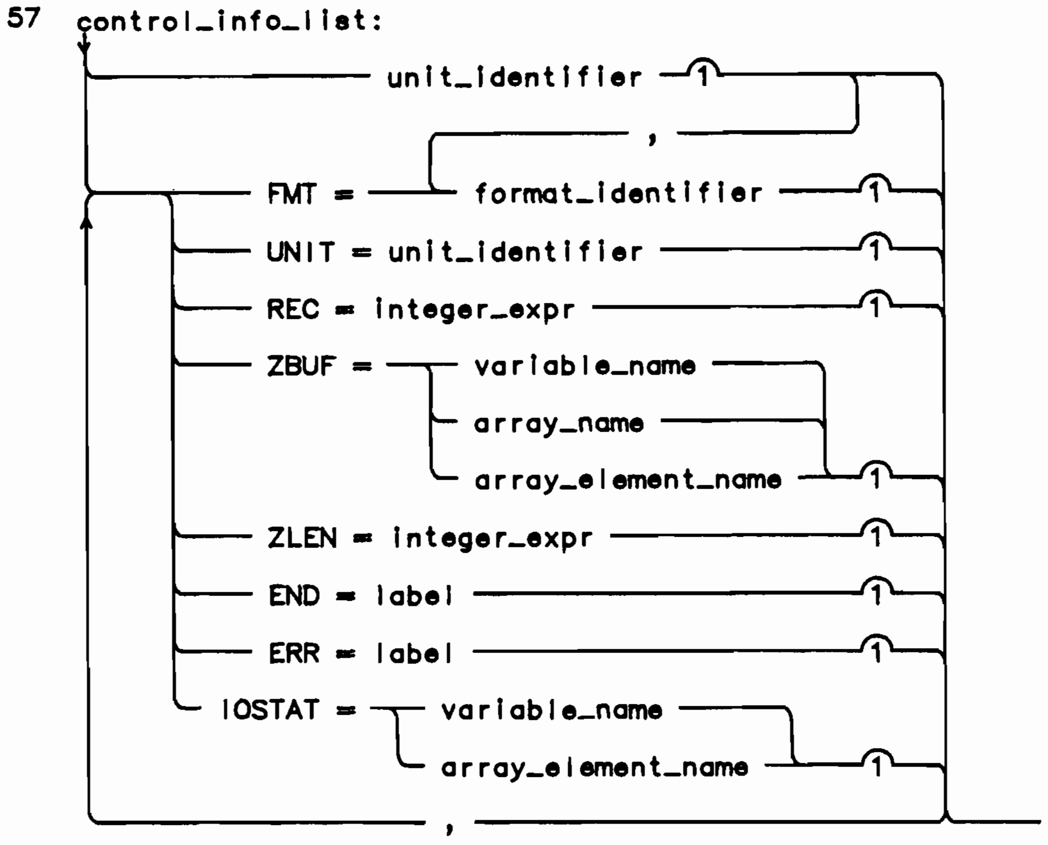
52

53

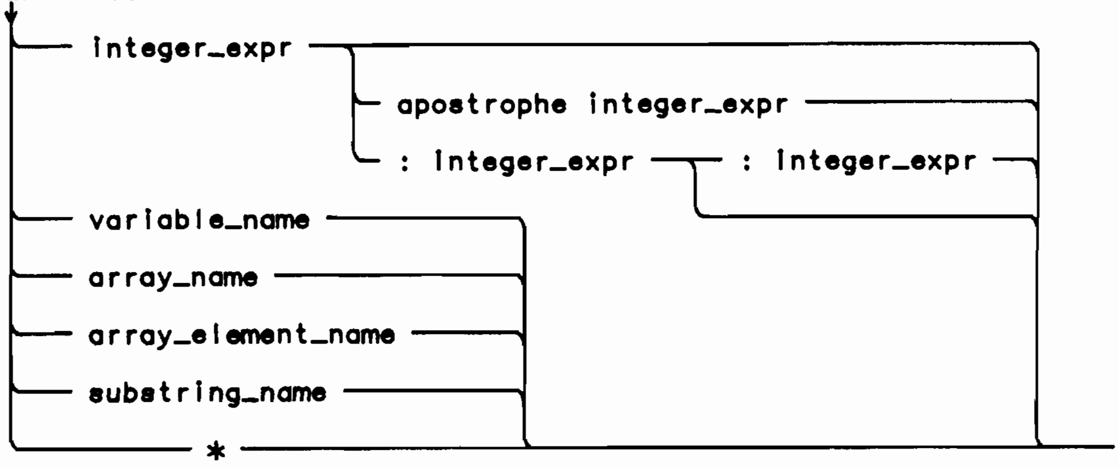
54



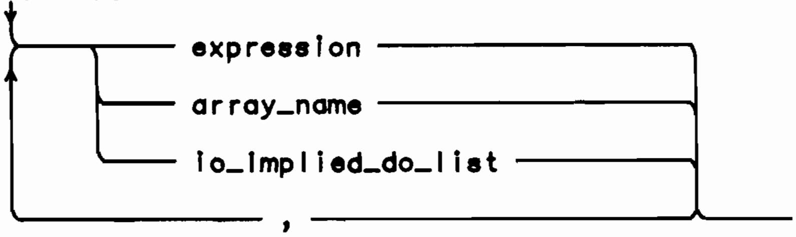

55 decode_statement:
DECODE
56 encode_statement:
ENCODE
(
integer_expr , format_identifier ,
variable_name
array_name
ERR = label
1
IOSTAT =
variable_name
array_element_name
1
,
)
io_list


57 control_info_list:
unit_identifier
1
,
FMT =
format_identifier
1
UNIT = unit_identifier
1
REC = integer_expr
1
ZBUF =
variable_name
array_name
array_element_name
1
ZLEN = integer_expr
1
END = label
1
ERR = label
1
IOSTAT =
variable_name
array_element_name
1
,

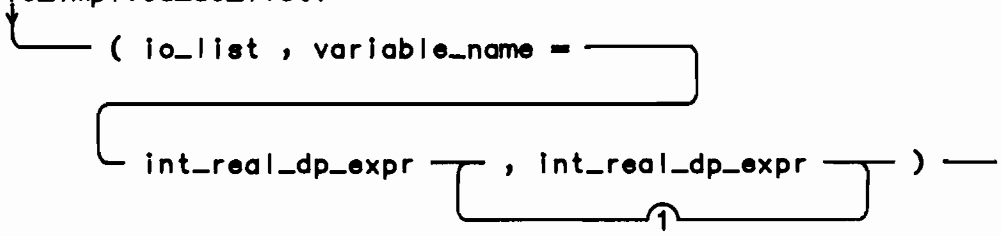
58 unit_identifier:



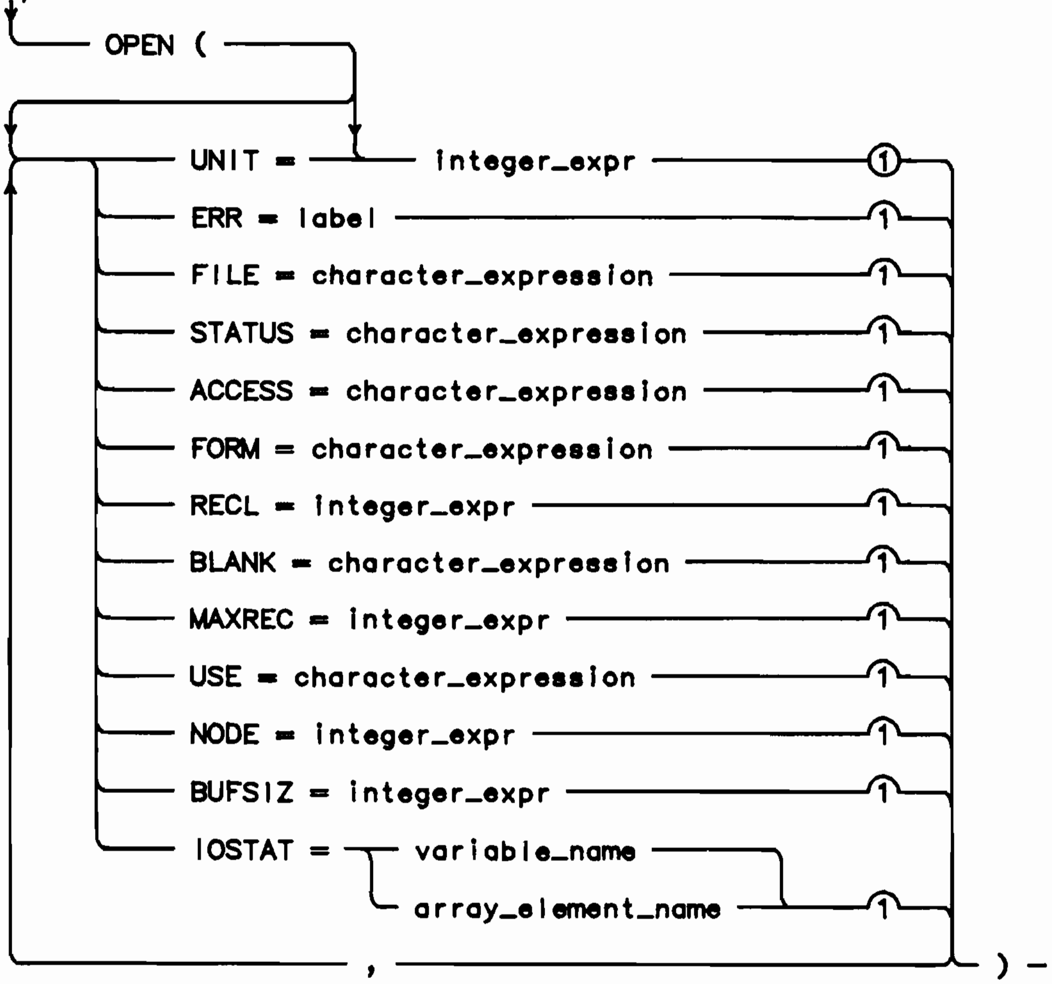
59 io_list:



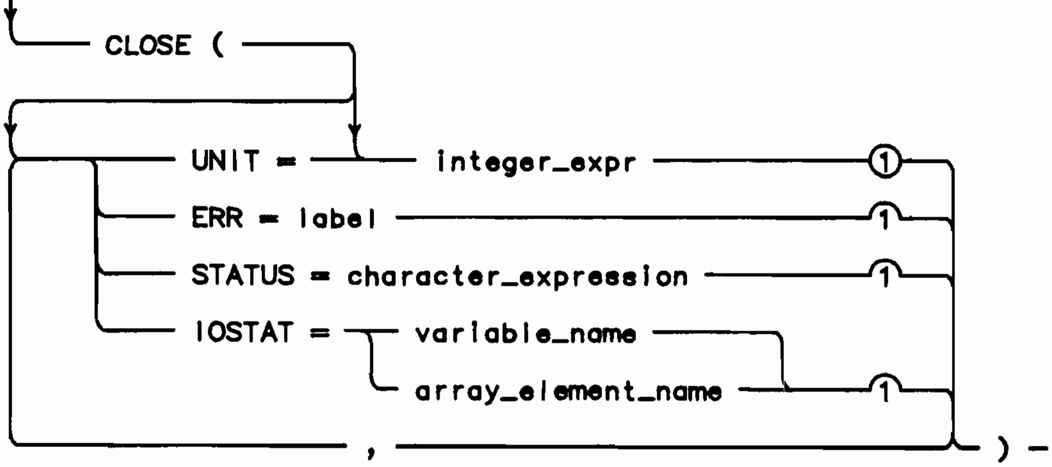
60 io_implied_do_list:



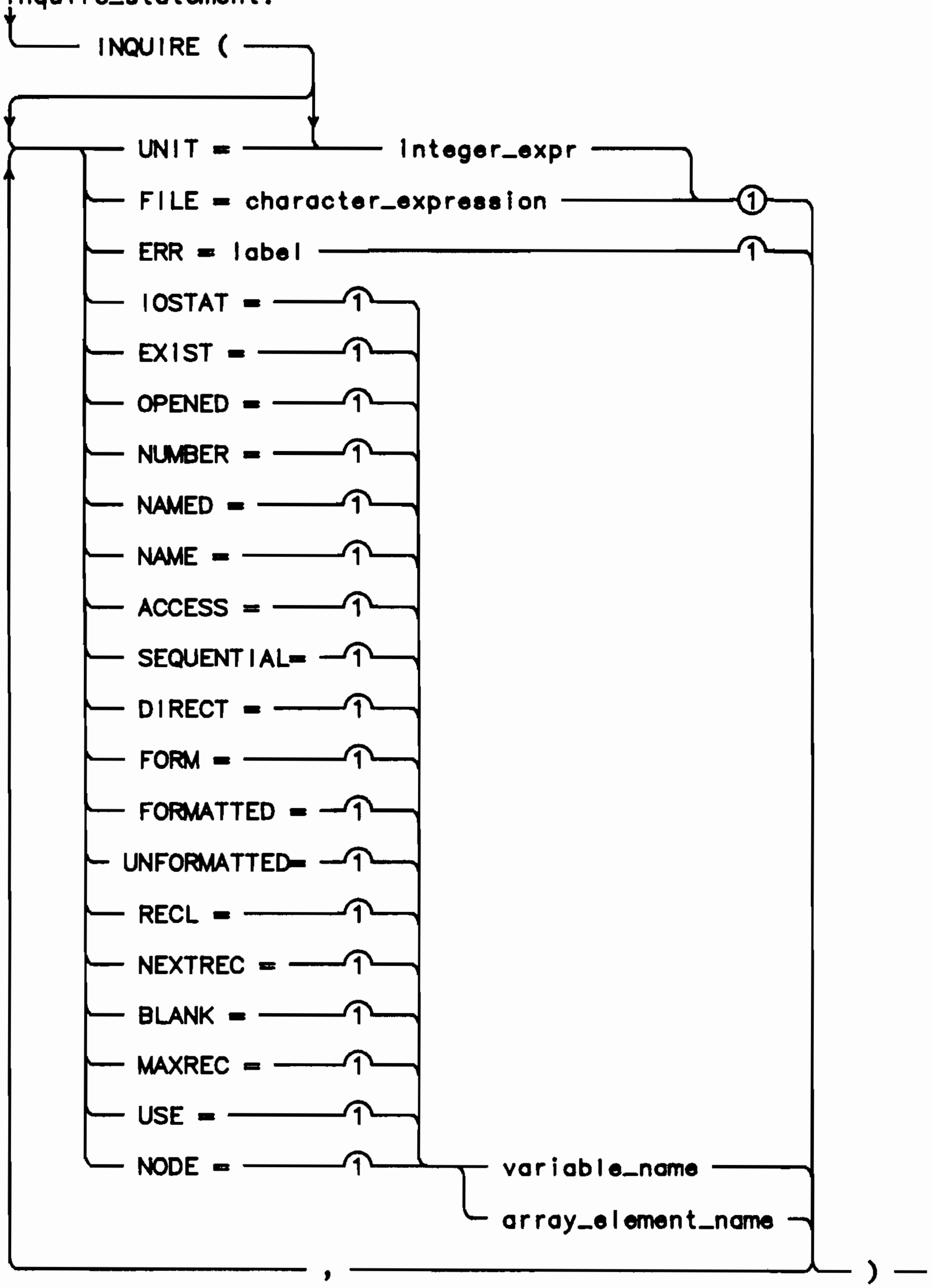
61 open_statement:


OPEN (
UNIT =
integer_expr
ERR = label
FILE = character_expression
STATUS = character_expression
ACCESS = character_expression
FORM = character_expression
RECL = integer_expr
BLANK = character_expression
MAXREC = integer_expr
USE = character_expression
NODE = integer_expr
BUFSIZ = integer_expr
IOSTAT =
variable_name
array_element_name
1
,
) -

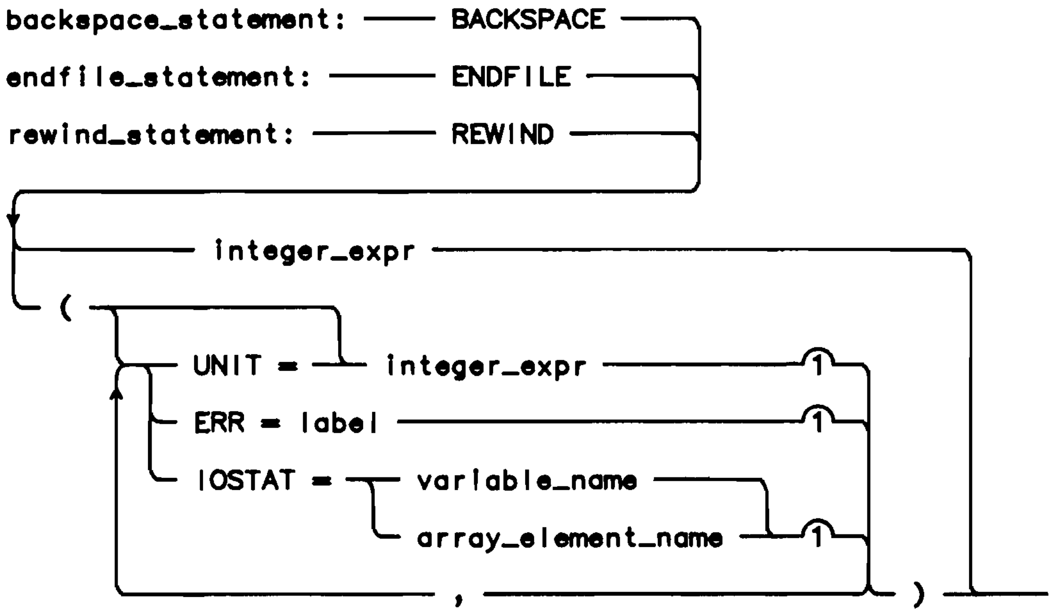
62 close_statement:



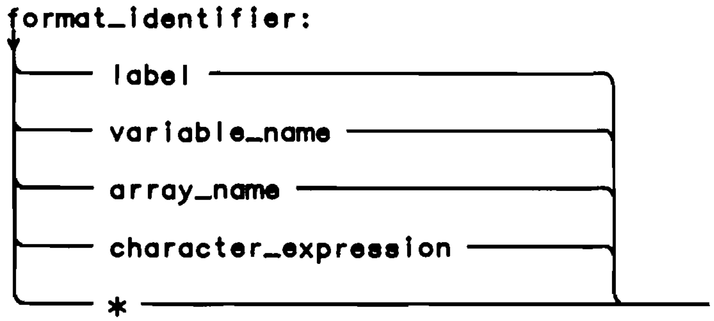
63 inquire_statement:



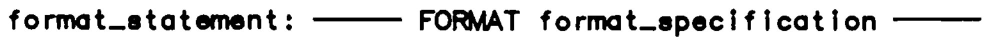
64 65 66



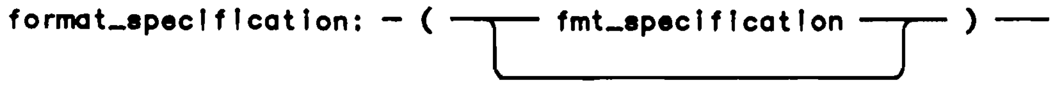
67



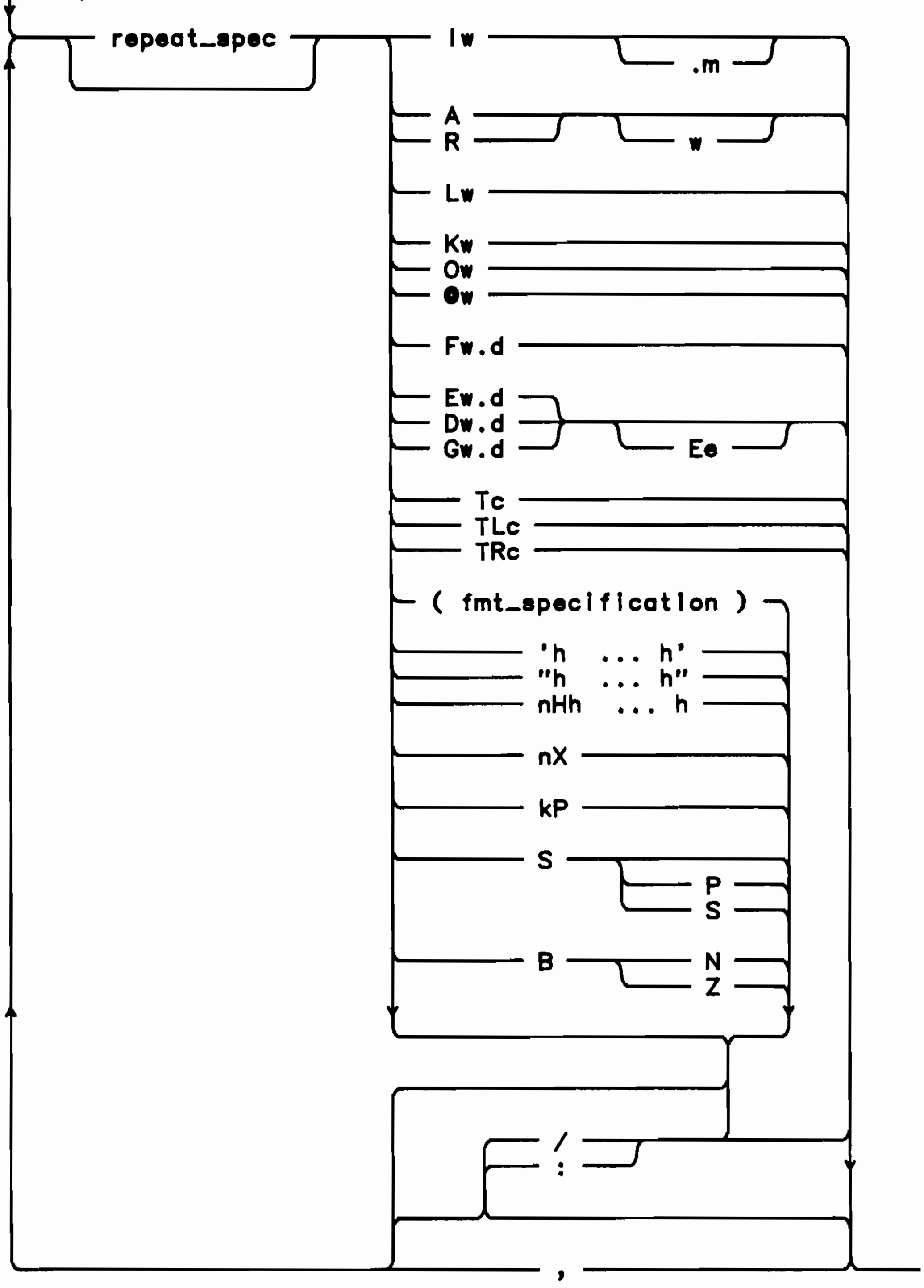
68



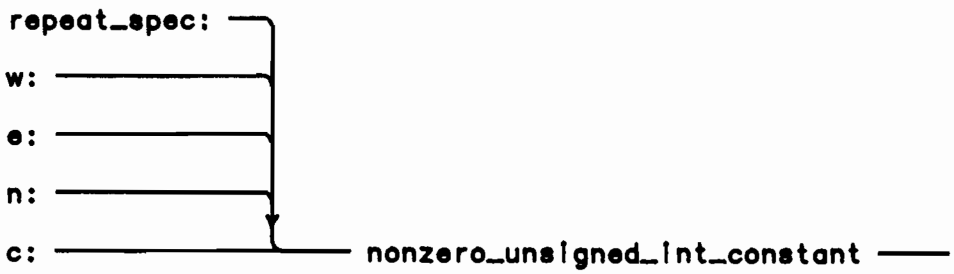
69



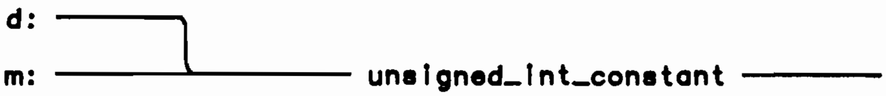
70 fmt_specification:



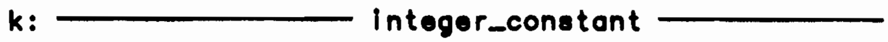
71–75



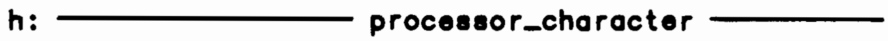
76–77



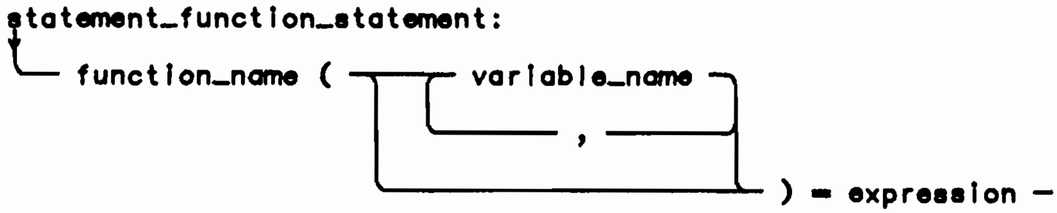
78



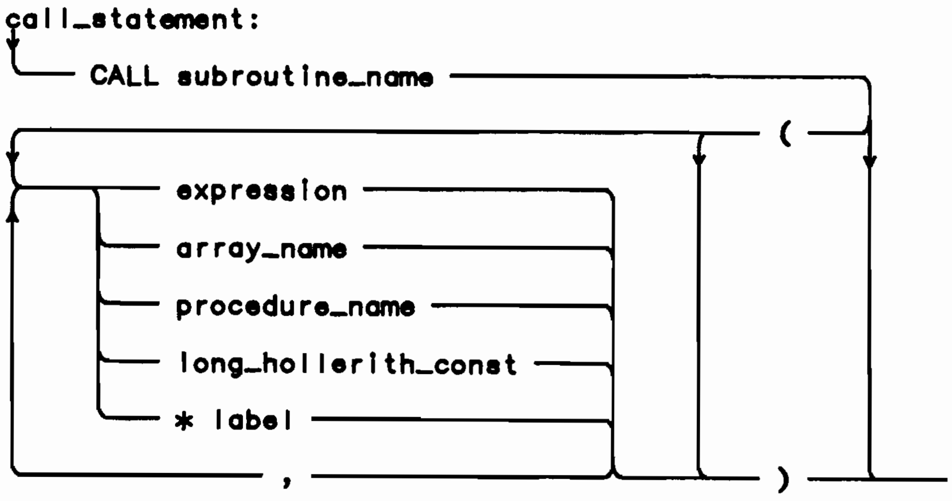
79



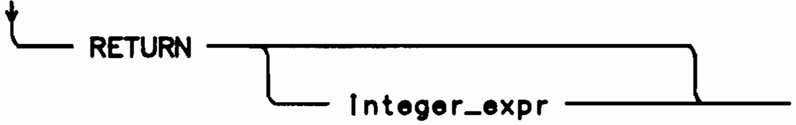
80



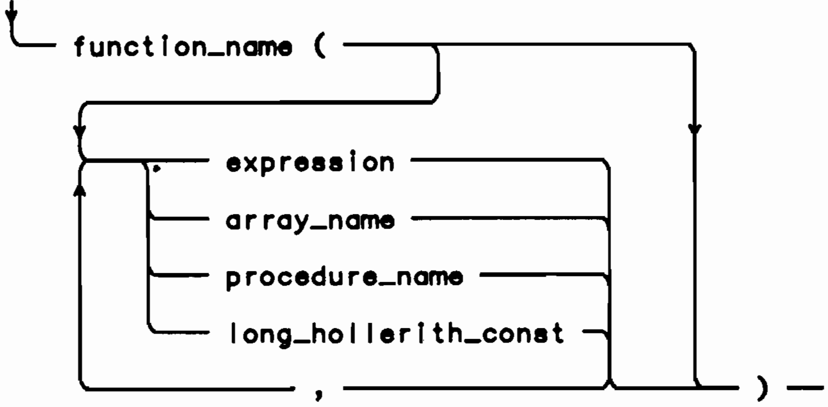
81



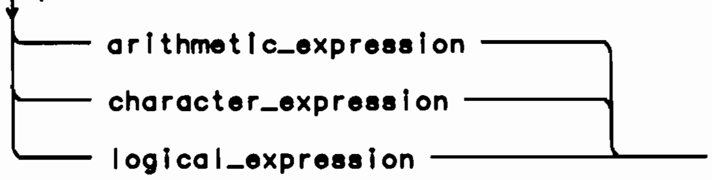
82 return_statement:



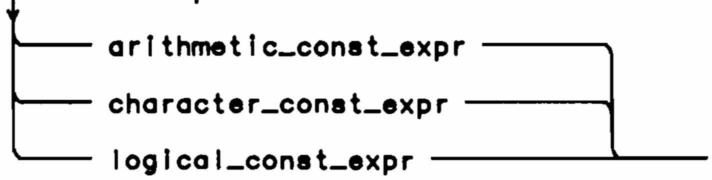
83 function_reference:

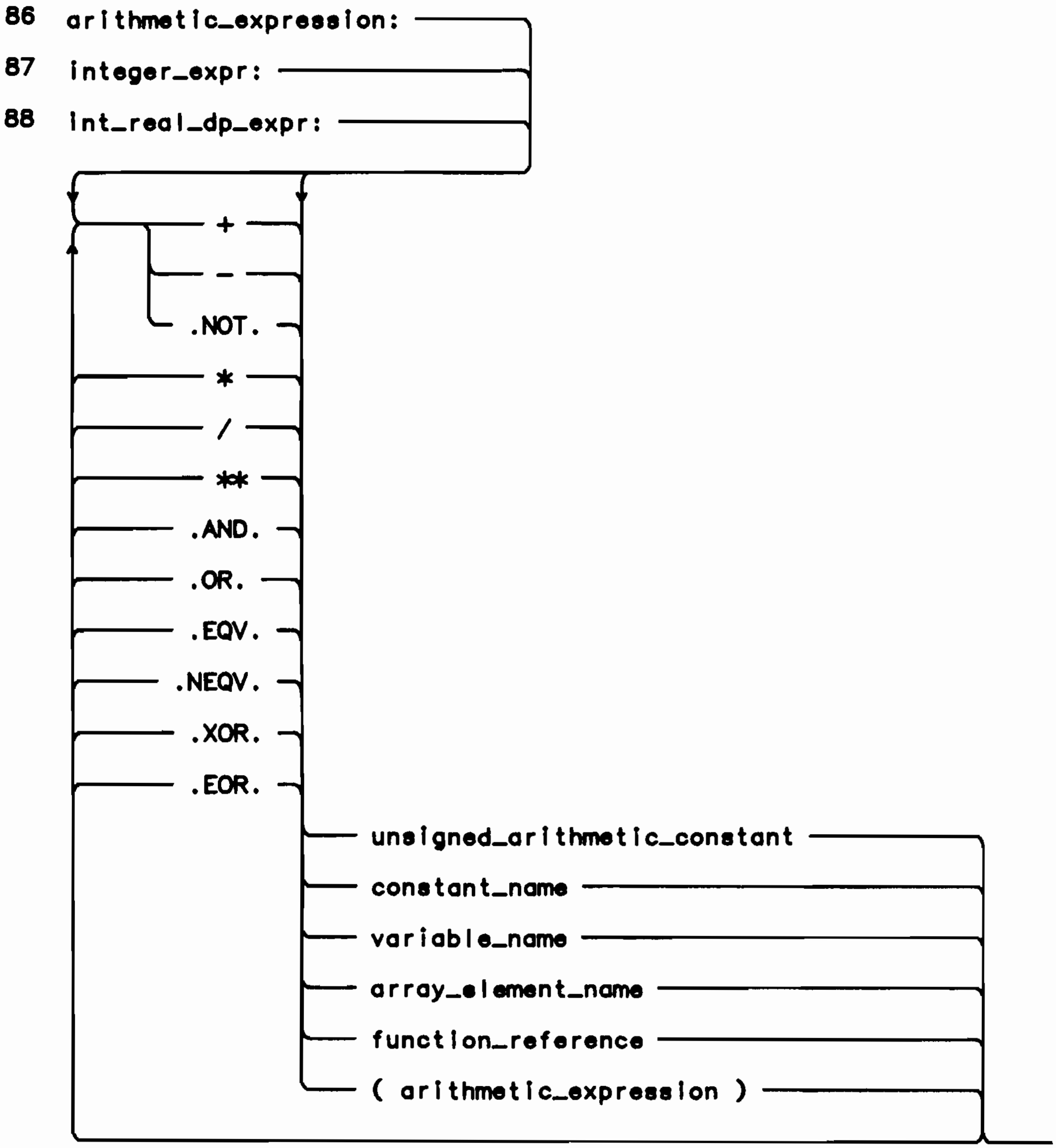


84 expression:

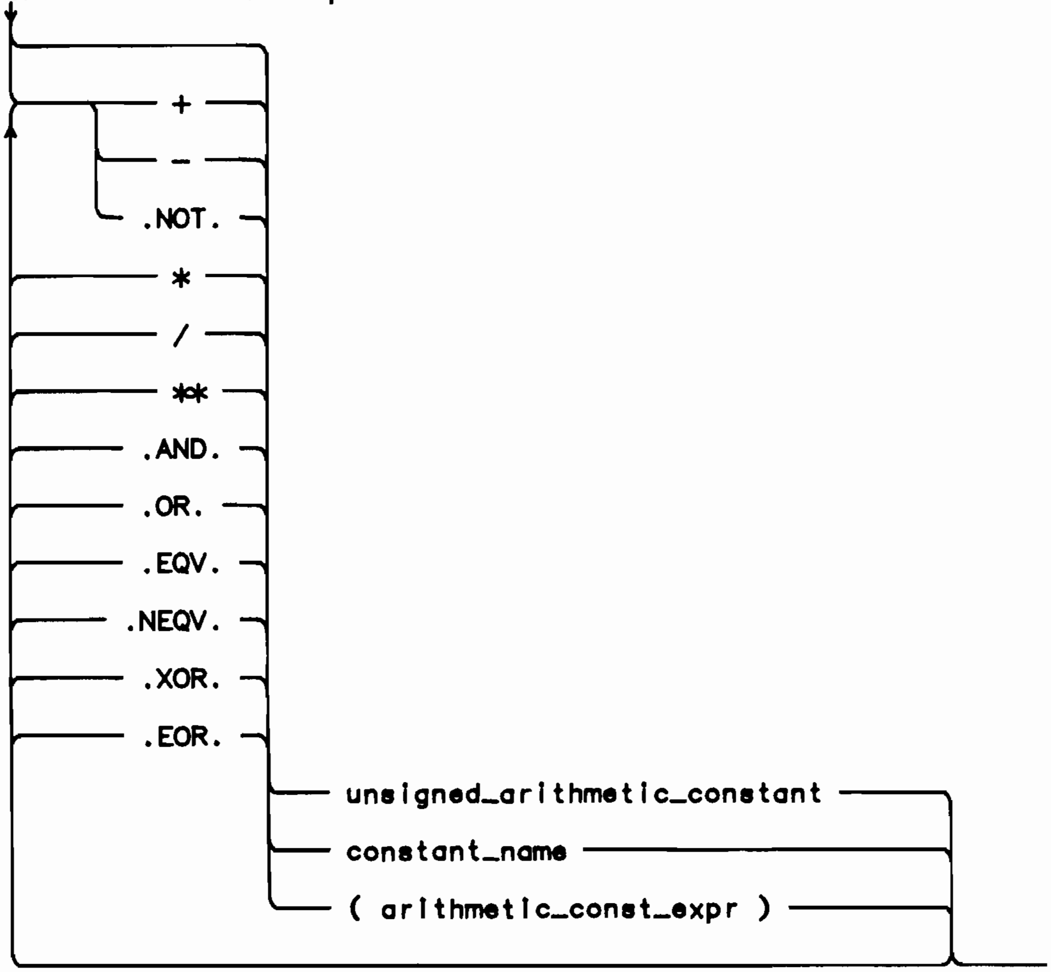


85 constant_expr:

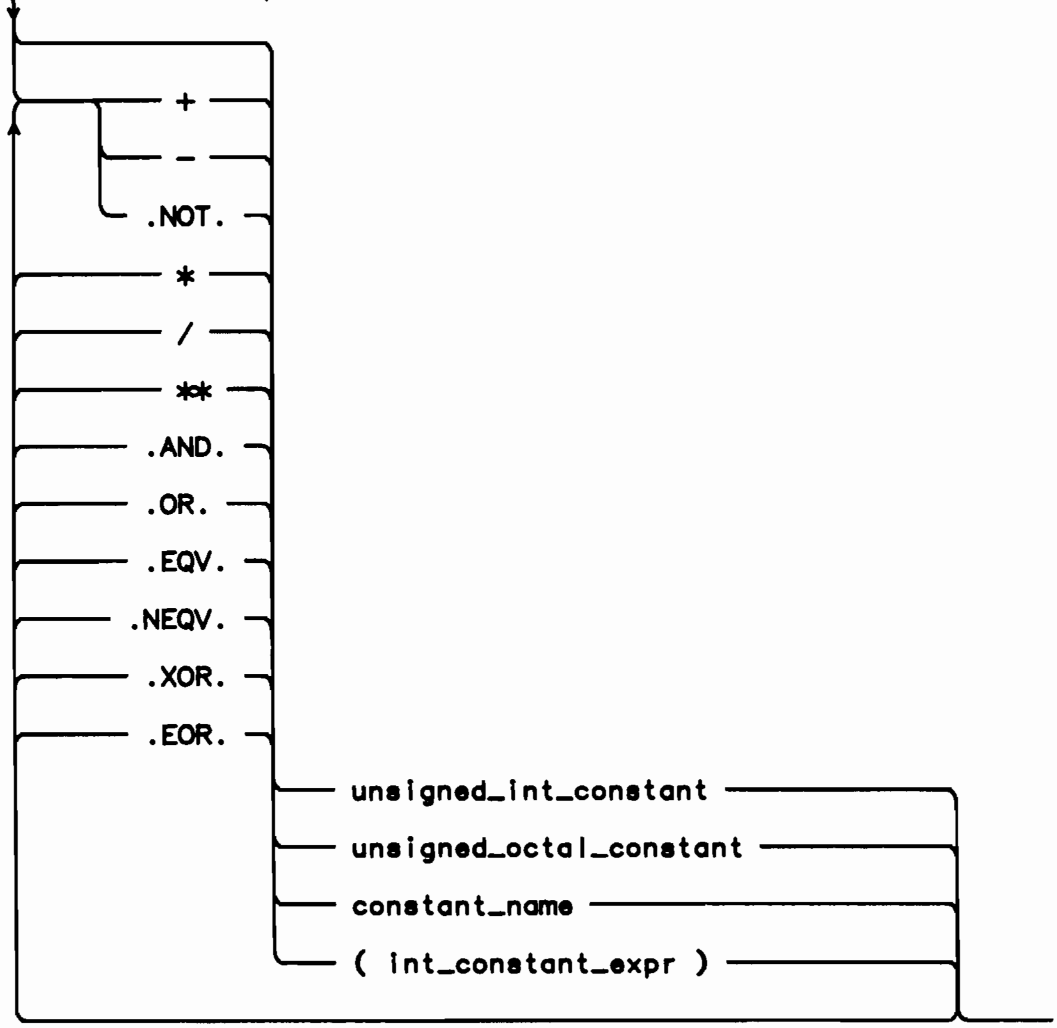



86 arithmetic_expression:
87 integer_expr:
88 int_real_dp_expr:
+
-
.NOT.
*
/
**
.AND.
.OR.
.EQV.
.NEQV.
.XOR.
.EOR.
unsigned_arithmetic_constant
constant_name
variable_name
array_element_name
function_reference
(arithmetic_expression)

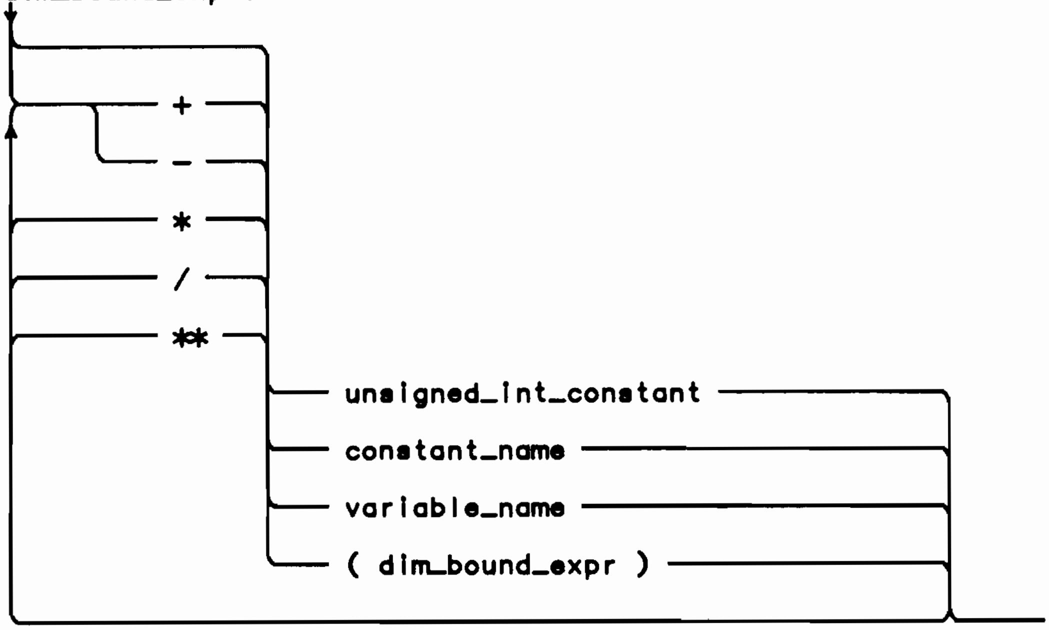
89 arithmetic_const_expr:



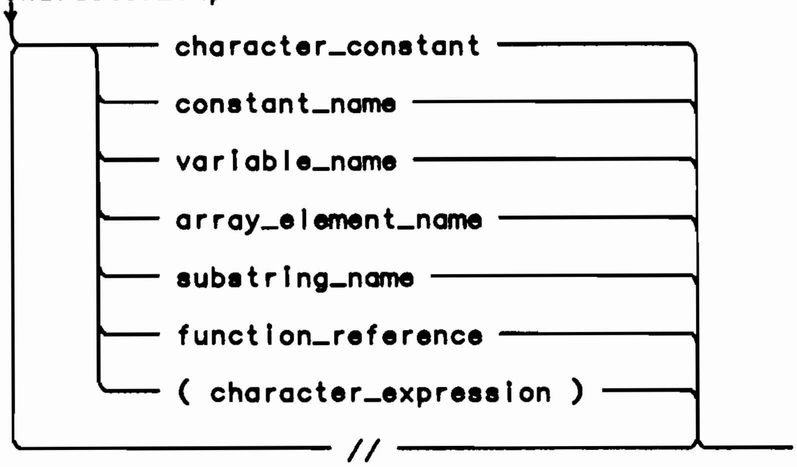
90 int_constant_expr:



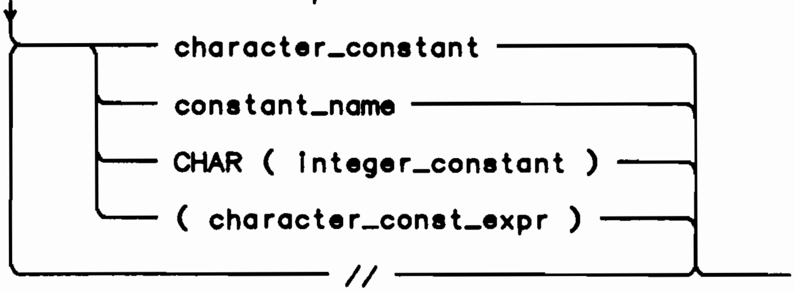
91 dim_bound_expr:



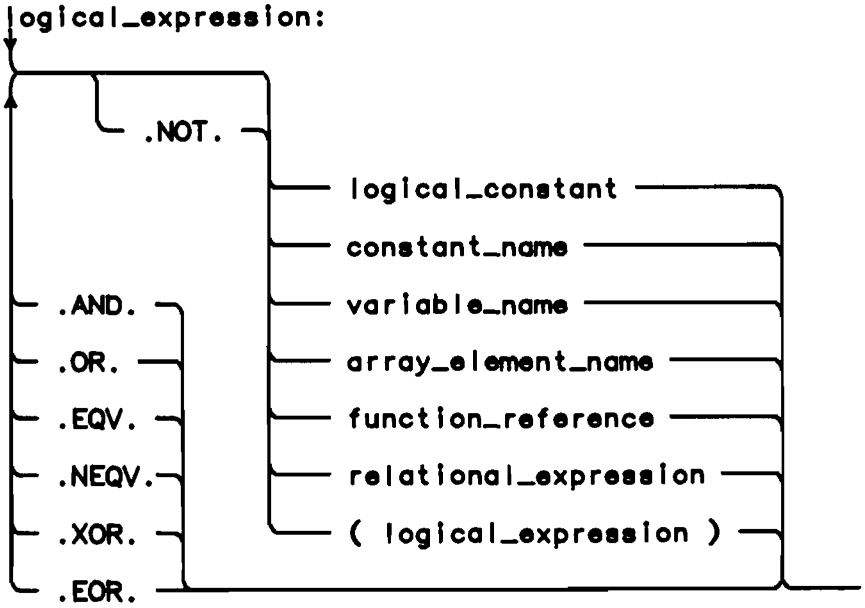
92 character_expression:



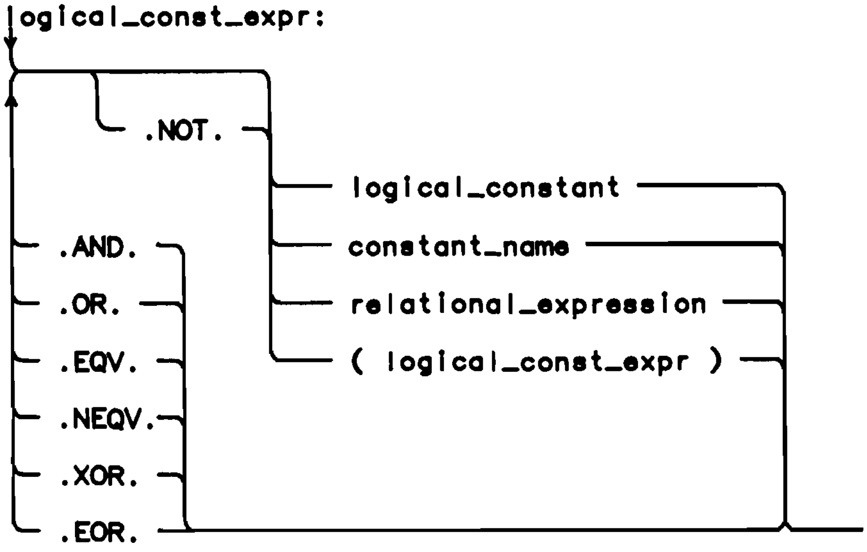
93 character_const_expr:



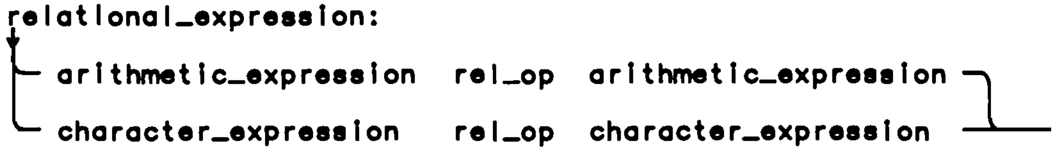
94


logical_expression:
.NOT.
logical_constant
constant_name
variable_name
array_element_name
function_reference
relational_expression
(logical_expression)
.AND.
.OR.
.EQV.
.NEQV.
.XOR.
.EOR.

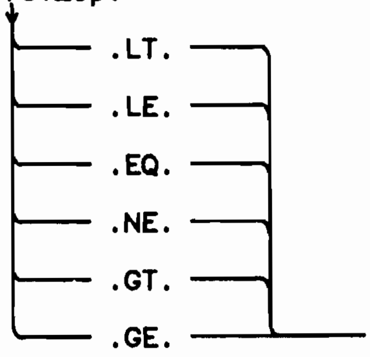
95


logical_const_expr:
.NOT.
logical_constant
constant_name
relational_expression
(logical_const_expr)
.AND.
.OR.
.EQV.
.NEQV.
.XOR.
.EOR.

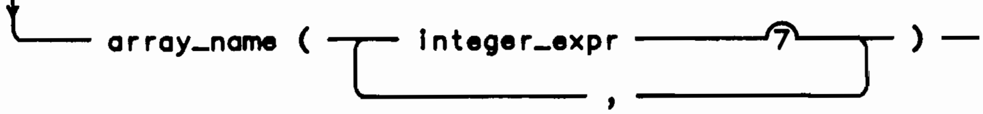
96


relational_expression:
arithmetic_expression rel_op arithmetic_expression
character_expression rel_op character_expression

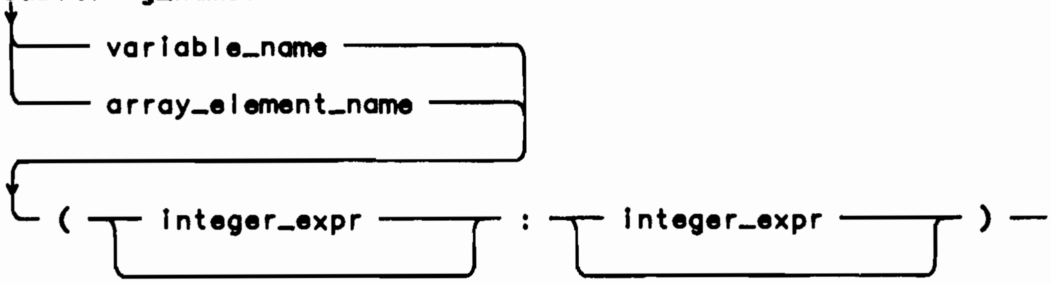
97 rel_op:



98 array_element_name:



99 substring_name:



100 constant_name:

101 variable_name:

102 array_name:

103 common_block_name:

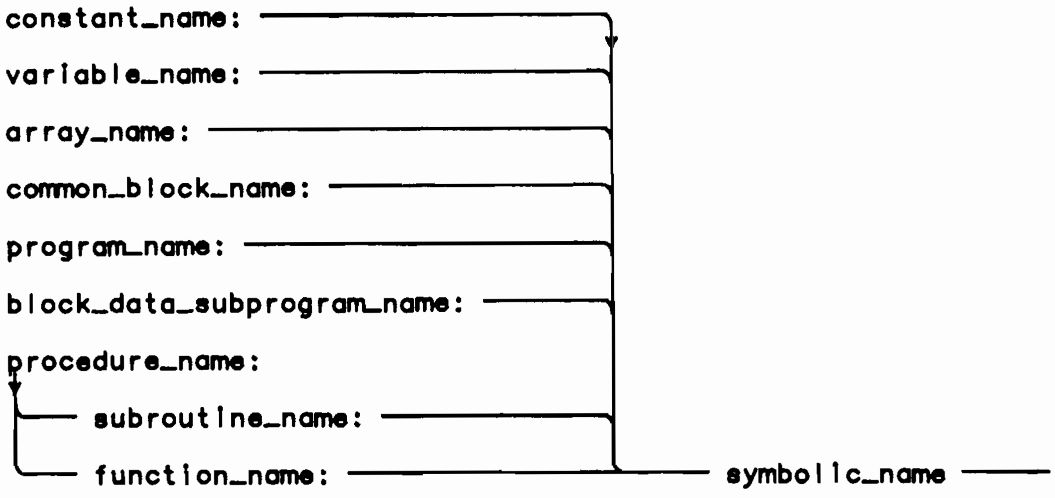
104 program_name:

105 block_data_subprogram_name:

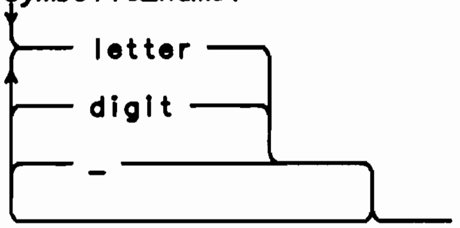
106 procedure_name:

107 subroutine_name:

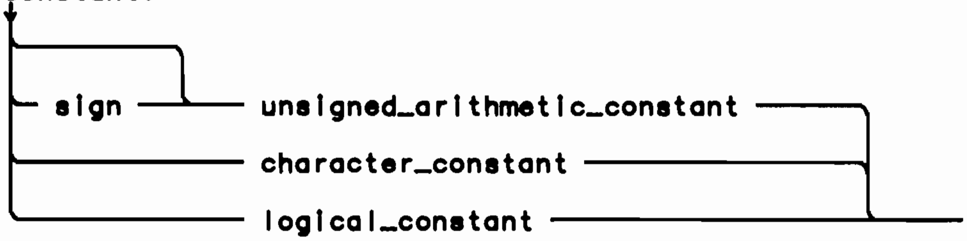
108 function_name:



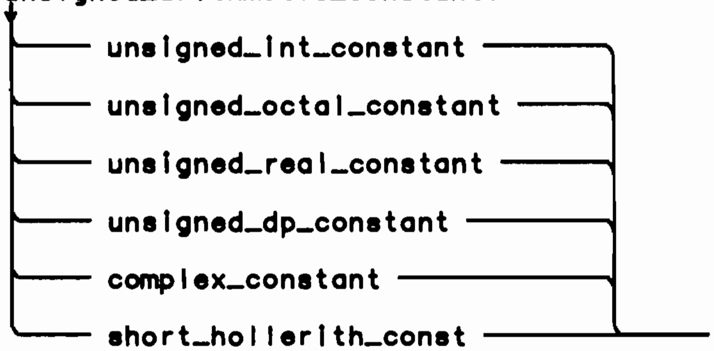
109 symbolic_name:



110 constant:



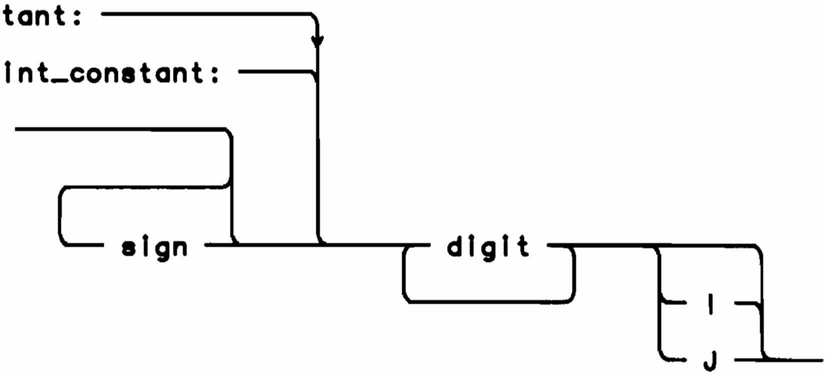
111 unsigned_arithmetic_constant:



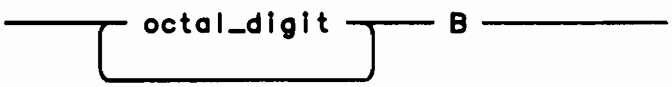
112 unsigned_int_constant:

113 nonzero_unsigned_int_constant:

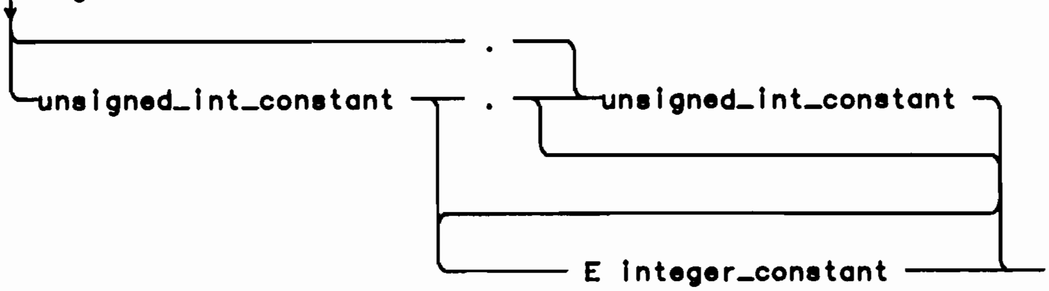
114 integer_constant:



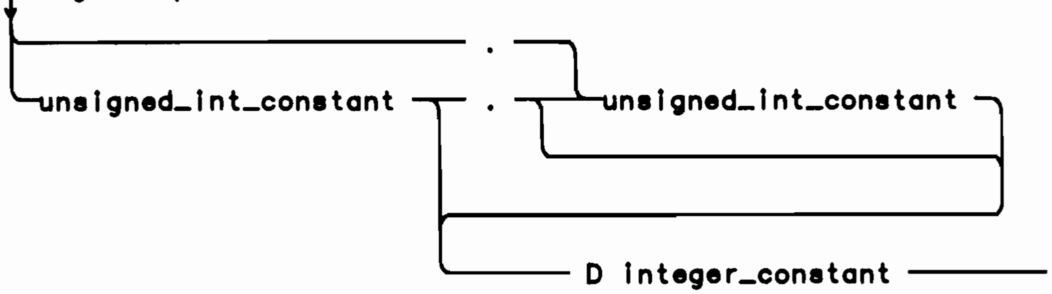
115 unsigned_octal_constant:



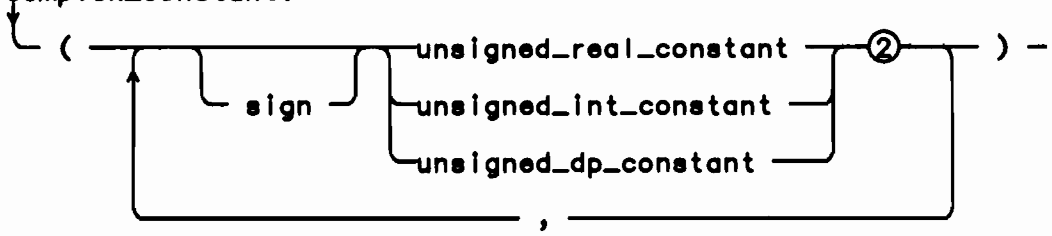
116 unsigned_real_constant:



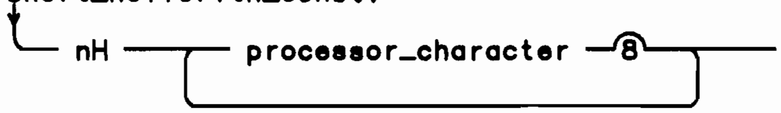
117 unsigned_dp_constant:

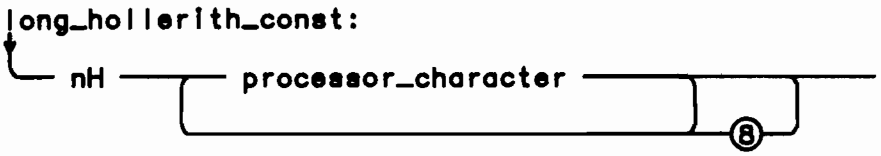


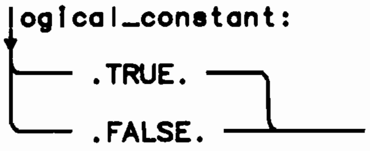
118 complex_constant:

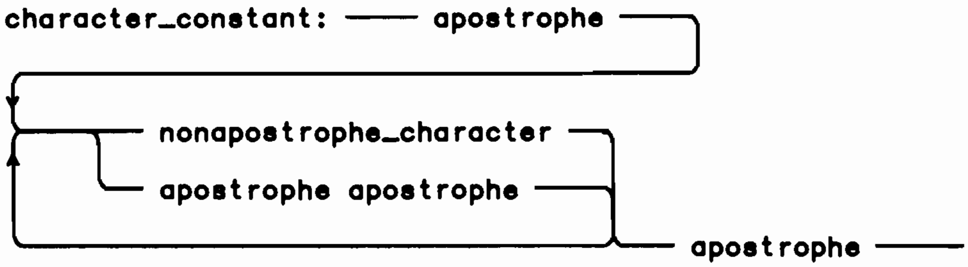


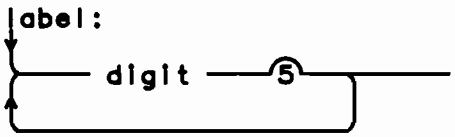
119 short_hollerith_const:

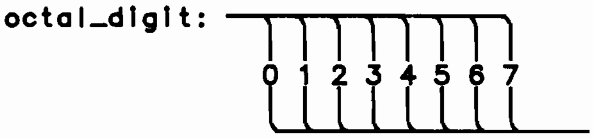


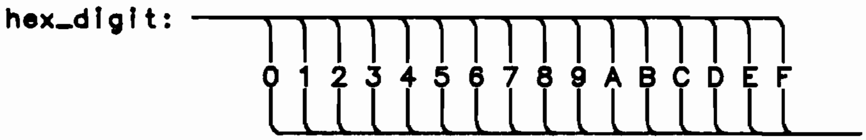
120


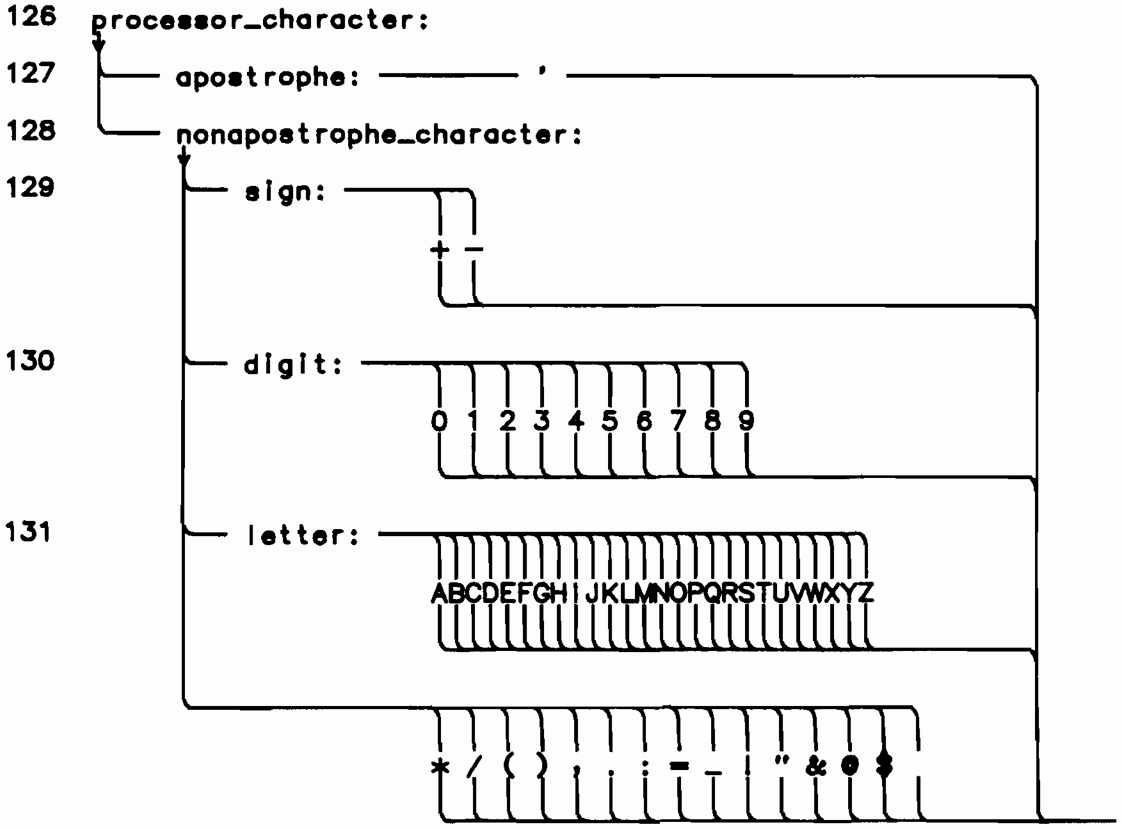
121


122


123


124


125



126 processor_character:
127 apostrophe: '
128 nonapostrophe_character:
129 sign:
+ –
130 digit:
0 1 2 3 4 5 6 7 8 9
131 letter:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
* / ( ) , . : = _ ! " & @ $

Cross-Reference to Syntax Charts

121 logical_constant: 94, 95,110
94 logical_expression: 40, 41, 42, 47, 84, 94
40 logical_if_statement: 7
120 long_hollerith_const: 81, 83

77 m: 70
2 main_program: 1
33 mil_std_hex: 30
32 mil_std_octal: 30

74 n: 70,1
128 nonapostrophe_character: 122
113 nonzero_unsigned_int_constant: 24, 30, 75

124 octal_digit: 32,115
61 open_statement: 7
6 other_specification_statement: 5

25 parameter_statement: 5
51 pause_statement: 7
54 print_statement: 7
106 procedure_name: 12, 15, 26, 81, 83
126 processor_character: 8, 10, 13, 79,119,120
104 program_name: 8
8 program_statement: 2

53 read_statement: 7
97 rel_op: 96
96 relational_expression: 94, 95
71 repeat_spec: 70
82 return_statement: 7
66 rewind_statement: 7

28 save_statement: 6
119 short_hollerith_const: 111
129 sign: 110,114,118
80 statement_function_statement: 5
50 stop_statement: 7
15 subroutine_argument_list: 13, 14
14 subroutine_entry: 9
107 subroutine_name: 13, 14, 81 13 subroutine_statement: 4
4 subroutine_subprogram: 1 99 substring_name: 20, 30, 31, 34, 58, 92
109 symbolic_name: 108

22 type_statement: 6

36 unconditional_goto: 35 58 unit_identifier: 57
111 unsigned_arithmetic_constant: 88, 89,110

117 unsigned_dp_constant: 111,118
112 unsigned_int_constant: 8, 77, 90, 91,111,116,117,118
115 unsigned_octal_constant: 90,111
116 unsigned_real_constant: 111,118

101 variable_name: 12, 15, 20, 21, 22, 28, 29, 30, 31, 34, 38, 46, 56, 57, 58, 60, 61, 62, 63, 66, 67, 80, 88, 91, 92, 94, 99

72 w: 70
52 write_statement: 7

CDS Usage

Code and Data Separation (CDS) is available exclusively under the RTE-A Operating System with the VC+ System Extension Package. CDS is not available under any other HP operating system, and programs compiled for other systems should always be non-CDS.

Most programs work equally well with or without code and data separation. A few existing programs make improper assumptions about the initial values of data in main programs or subprograms because of static storage use; since CDS programs use less static storage and more dynamic storage, such programs may fail when first compiled in CDS mode. This chapter clarifies these considerations and explains a few features that are unique to one mode or the other.

CDS Overview

Details of the CDS mechanisms and instructions are found in the *RTE-A Programmer's Reference Manual*, part number 92077-90007, and the *RTE-A System Design Manual*, part number 92077-90013. The overview given here profiles the information necessary to use CDS with FORTRAN programs. Since EMA works the same in CDS and non-CDS programs, EMA is not discussed.

Non-CDS programs have only one memory area: the area with instructions and data, called the *program space* or the *non-EMA space*. Instructions and data are freely mixed in these programs, and a program with errors can easily overwrite its own instructions (for example, with a bad array subscript). All data is allocated statically; that is, it has its own private area of memory, always in the same place, and its area is never used for anything else. As a side effect of this, variables in subprograms have the same value on entry to the subprogram as they had at the last exit.

Subprogram calls in non-CDS programs are made by a machine language instruction that saves the address of the call in a location just before the subprogram. The standard entry mechanism in that subprogram copies the addresses of the parameters to a list that is also just before the subprogram. Both of these methods cause code modification, in that the memory area containing the instructions of the subprogram is changed. The parameters are put into statically allocated memory.

CDS programs replace the program memory area with two areas: *code space* and *data space*. Instructions are put into code space, and data is put into data space. A program cannot accidentally change its instructions, since it can't access its code space. Some data, such as common blocks and items in DATA statements, are statically allocated in data space. Other data is dynamically allocated.

CDS programs have a *stack*. This is an area in data space that is allocated on demand when each subprogram is entered, and released when it returns, in LIFO (last in, first out) manner. Each subprogram allocates as much stack space as it needs.

Some local variables and arrays in CDS subprograms are allocated on the stack. When the subprogram exits, these variables and arrays are lost; the next subprogram to be called re-uses that memory for something else. These variables do not keep their values from one call to the next.

Subprograms in CDS mode are called with a different machine language instruction that puts the address of the call and the parameters on the stack. No code is modified. Since each subprogram allocates space for its return point and variables when it is called, recursion works; recursive calls do not cost any more than nonrecursive calls.

Static vs. Dynamic Memory

Since local variables may be allocated on the stack, they behave differently in CDS programs than in non-CDS. Uninitialized variables have random values, instead of starting out at zero as they often do in non-CDS programs. Variables do not retain their values from one subprogram call to the next. The ANSI 77 standard explicitly states that variables become undefined at subprogram exit, but many non-CDS programs assume that the values are not lost.

When such programs are compiled in CDS mode, they must include SAVE statements, which declare that certain variables must stay defined when the routine exits. An example is a routine that counts the number of times it is called, or a report generator that appends strings to an output line on each call until the line is printed.

One way to get such programs running quickly is to use blank SAVE statements. These declare that all variables are to be allocated statically. This has the disadvantage of increasing the amount of data space used; a large program may not fit in memory with blank SAVE statements, though it fits easily without them.

Restrictions

For compatibility, CDS subprograms can call non-CDS subprograms. However, the reverse does *not* work; non-CDS subprograms can call only other non-CDS subprograms. If a large program is converted to CDS piecemeal, the main program must be converted first, then the subprograms that the main program calls, and so on, in a top-down fashion.

There is an obscure FORMAT capability that allows a READ statement to read ASCII data into a Hollerith field in the format itself, changing the format (but not any variables) so that the next time it is used in a WRITE, the Hollerith data has the value from the READ.

Since data space is the critical resource in CDS programs, FORTRAN puts the ASCII data of format statements in code space. When a READ or WRITE statement is used, the library copies the format (using special instructions) from code space to a buffer in data space, and uses the copy for the READ or WRITE. When data is read into a Hollerith field in a READ, the data modifies the copy of the format, not the original. The next time the format is used, it has its original value again. Therefore, programs that rely on reading into format statements cannot use CDS. (The ANSI 77 standard prohibits reading into Hollerith formats.)

The CALL CODE feature, which was obsoleted by the introduction of ENCODE and DECODE in FTN4X, does not work in CDS mode. This is because the CODE routine actually looks ahead in the program to find the READ or WRITE statement and decodes the first instruction from it; that instruction is not even accessible in CDS mode, and if it were, it might work differently in CDS mode and be misinterpreted by CODE. Since CODE is not a reserved name, and could be the name of a legitimate subroutine, FORTRAN 77 does not issue a warning when a CALL CODE is done in CDS mode. The CODE routine may set up an internal conversion using a random address; for WRITE statements, this can have the effect of randomly overwriting memory. In short, be careful to avoid CALL CODE in CDS programs. Because it has been replaced by ENCODE and DECODE and internal files, all CALL CODE statements should be changed to newer statements, even in non-CDS code.

Recursion

Recursion works only in CDS mode. All local variables are put on the stack, unless they are initialized in DATA statements. All local equivalence groups are also put on the stack, unless an item in a group is initialized in a DATA statement. Care should be taken to ensure that variables that must be unique to each call are put on the stack; one way to check is with a T option listing, which lists the location of each variable in the subprogram.

A subprogram can call itself directly or indirectly. If a subprogram calls itself directly, the type and number of parameters are checked.

Variables that are to be the same for each subprogram invocation (that is, static) can be kept off the stack in CDS by use of DATA statements; however, the preferred method is SAVE statements. SAVE statements ensure program portability, while DATA statements do not because other compilers may implement DATA statements by executing assignments when the subprogram is entered the first time, and then reuse their memory area between subprogram calls.

Recursion is an extension of the ANSI 77 standard.

Libraries

CDS programs must be linked with the CDS versions of the system libraries. Normally these libraries are selected automatically by the linker. If you specify system libraries explicitly, be sure to specify the CDS versions of the libraries. See the appropriate system reference manual for further information about system libraries.

Index