TOPS–10/TOPS–20 FORTRAN Language Manual

AA–N383B–TK

May 1985

This document describes the language elements of FORTRAN–10 and FORTRAN–20.

This manual supersedes the *TOPS–10/20 FORTRAN Language Manual*, order number AA–N383A–TK.

OPERATING SYSTEM: TOPS–10 V7.02
TOPS–20 V4.1, V5.1

SOFTWARE: FORTRAN–10 V10
FORTRAN–20 V10

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center. Outside the United States, orders should be directed to the nearest DIGITAL Field Sales Office or representative.

Northeast/Mid–Atlantic Region

Digital Equipment Corporation
PO Box CS2008
Nashua, New Hampshire 03061
Telephone:(603)884–6660

Central Region

Digital Equipment Corporation
Accessories and Supplies Center
1050 East Remington Road
Schaumburg, Illinois 60195
Telephone:(312)640–5612

Western Region

Digital Equipment Corporation
Accessories and Supplies Center
632 Caribbean Drive
Sunnyvale, California 94086
Telephone:(408)734–4915

digital equipment corporation • marlboro, massachusetts

First Printing, February 1983
Revised, May 1985

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The following are trademarks of Digital Equipment Corporation:

digital™

DEC
DECmate
DECsystem–10
DECSYSTEM–20
DECUS
DECwriter
DIBOL
MASSBUS
PDP
P/OS
Professional
Q–BUS
Rainbow
RSTS
RSX
RT
UNIBUS
VAX
VMS
VT
Work Processor

The postage-prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 CHARACTERS AND LINES

CHAPTER 3 CONSTANTS

CHAPTER 4 SYMBOLIC NAMES, VARIABLES, AND ARRAYS

CHAPTER 5 EXPRESSIONS

CHAPTER 6 EXECUTABLE AND NONEXECUTABLE STATEMENTS

CHAPTER 7 SPECIFICATION AND DATA STATEMENTS

CHAPTER 8 ASSIGNMENT STATEMENTS

CHAPTER 9 CONTROL STATEMENTS

FIGURES

TABLES

PREFACE

The FORTRAN-10/20 Language Manual reflects the software as of Version 10 of the FORTRAN-10/20 compiler, the FORTRAN-10/20 Object Time System (FOROTS), and the FORTRAN-10/20 debugging program (FORDDT).

This manual describes the FORTRAN language as implemented for the TOPS-10 operating system (FORTRAN-10) and the TOPS-20 operating system (FORTRAN-20). Any differences between FORTRAN-10 and FORTRAN-20 are noted in this manual.

Since this is a reference manual, we assume that you have used FORTRAN before. If you haven't, you should read one of the many introductory FORTRAN texts.

CONVENTIONS

The following conventions are used throughout the manual:

Braces { } indicate that a choice must be made from one of the enclosed lines.

Brackets [ ] indicate an optional feature.

Ellipsis ... or . . . indicate the omission of information from a programming example or that items in a command line can be optionally repeated.

Lowercase letters indicate variable information you supply in a command string.

UPPERCASE LETTERS indicate fixed (or literal) information that you must enter as shown in a command string.

b̸ indicates a blank.

The standard for FORTRAN is the American National Standards Institute (ANSI) FORTRAN, X3.9-1978 (also known as FORTRAN-77). FORTRAN-10/20 extensions and additions to ANSI FORTRAN are in blue print in this manual.

MANUALS REFERENCED

The following manuals are referenced from TOPS-10 publications:

- TOPS-10 Operating System Commands Manual
- SOS Reference Manual
- TOPS-10 Monitor Calls Manual
- TOPS-10 Hardware Reference Manual
- TOPS-10 LINK Reference Manual
- TOPS-10 SORT/MERGE User's Guide
- TOPS-10 FORTRAN Installation Guide

The following manuals are referenced from TOPS-20 publications:

- TOPS-20 Commands Reference Manual
- TOPS-20 EDIT Reference Manual
- TOPS-20 User's Guide
- TOPS-20 Monitor Calls Manual
- TOPS-20 Link Reference Manual
- TOPS-20 SORT/MERGE User's Guide
- TOPS-20 FORTRAN Installation Guide

The following TOPS-10/TOPS-20 manual are referenced:

- FORTRAN-10/20 and VAX-11 FORTRAN Compatibility Manual
- TOPS-10/TOPS-20 FORTRAN Pocket Guide
- TOPS-10/TOPS-20 COBOL-74 Language Manual
- TOPS-10/20 BLISS Language Guide
- TOPS-10/20 Common Math Library Manual

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

The FORTRAN language, as implemented on the TOPS-10 and TOPS-20 operating systems, is compatible with and encompasses the standard described in "American National Standard FORTRAN, X3.9-1978" (referred to as the FORTRAN-77 standard) at the full-language level.

FORTRAN-10/20 provides many extensions and additions to the FORTRAN-77 standard that greatly enhance the usefulness of FORTRAN and increase its compatibility with FORTRAN languages implemented by other computer manufacturers. The extensions and additions to the standard FORTRAN-77 are printed in this manual in blue print.

A FORTRAN source program consists of a set of statements constructed using the language elements and the syntax described in this manual. A given FORTRAN statement performs any one of the following functions:

1. It causes operations such as multiplication, division, and branching to be carried out.
2. It specifies the type and format of the data being processed.
3. It specifies the characteristics of the source program.

FORTRAN statements are composed of keywords (words that are recognized by the compiler) used with elements of the language set: constants, variables, and expressions. There are two basic types of FORTRAN statements: executable and nonexecutable.

Executable statements specify the actions of the program; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and the kind of subprograms that may be included in the program. The compilation of executable statements results in the creation of executable code in the object program. Nonexecutable statements provide information only to the compiler; they do not create executable code.

1.2 MANUAL ORGANIZATION

This manual is divided into three parts: Language Elements, Statements, and Language Usage. Sections 1.2.1 through 1.2.3 contain general descriptions for each of these three parts. Section 1.2.4 contains a general description of the appendixes.

1.2.1 FORTRAN Language Elements (Part I)

Part I of this manual describes the fundamental elements of FORTRAN programs, including (by chapter):

Chapter 2, CHARACTERS AND LINES, describes FORTRAN source program characters and lines. The FORTRAN compiler interprets your source program and translates it into machine code (executable code).

Chapter 3, CONSTANTS, describes FORTRAN data types and constants. FORTRAN enables you to manipulate information (data) in a variety of ways. This chapter describes the techniques for defining FORTRAN constants of various data types.

Chapter 4, SYMBOLIC NAMES, VARIABLES, AND ARRAYS, describes symbolic names, variables, and arrays in FORTRAN. The symbolic name is used in a variety of ways in a source program; this chapter describes the conventions for using symbolic names to define both variables and arrays.

Chapter 5, EXPRESSIONS, introduces and describes FORTRAN operators. Expressions in FORTRAN are formed using variables, constants, and operators.

1.2.2 FORTRAN Statements (Part II)

Part II of this manual describes all the statements in the FORTRAN language. The following list describes (by chapter) the information presented in Part II:

Chapter 6, EXECUTABLE AND NONEXECUTABLE STATEMENTS, defines the term "FORTRAN Statement", and describes the "Compilation Control Statements".

Chapter 7, SPECIFICATION AND DATA STATEMENTS, describes all the statements within the category "Specification and Data Statements". The specification statements enable you to explicitly define the data types of variables used within your program; the DATA statement enables you to create initially defined constants within your program.

Chapter 8, ASSIGNMENT STATEMENTS, describes all the statements within the category "Assignment Statements". FORTRAN assignment statements enable you to assign values to variables, and to assign statement labels to symbolic names.

Chapter 9, CONTROL STATEMENTS, describes all the statements within the category "Control Statements". The default execution sequence of lines in a FORTRAN program is each line from left-to-right, and all lines from top-to-bottom. You use the FORTRAN control statements to alter the default execution sequence, to stop or pause during program execution, or to mark the end of an executable program.

Chapter 10, DATA TRANSFER STATEMENTS, describes the data transfer category of "FORTRAN Input/Output (I/O) Statements". As the term implies, a data transfer statement moves data from one place to another.

Chapter 11, FILE-CONTROL AND DEVICE-CONTROL STATEMENTS, describes file-control and device-control categories of "FORTRAN I/O Statements". The file-control statements enable you to associate a unit number with a file. Device-control statements enable you to position a storage medium (for example, magnetic tape) on a connected unit.

Chapter 12, FORMATTED DATA TRANSFERS, describes three types of data formatting. During certain types of data transfer operations, you must specify the format of the data being transferred. FORTRAN provides three techniques for specifying the format of data: FORMAT-Statement, List-Directed, and NAMELIST-Statement formatting.

Chapter 13, FUNCTIONS AND SUBROUTINES, describes FORTRAN functions and subprograms. Functions and subprograms provide a technique for producing clear and concise FORTRAN programs. FORTRAN-10/20 provides both predefined functions and subprograms, and the statements for defining your own functions and subprograms.

Chapter 14, BLOCK DATA SUBPROGRAMS, describes the block-data subprogram. This type of subprogram enables you to define initial values for variables in COMMON.

1.2.3 FORTRAN Language Usage (Part III)

Parts I and II of the manual contain complete descriptions of FORTRAN elements and statements. Part III of the manual contains explanations of how you use FORTRAN-10/20. The following usage topics are covered in Part III:

Chapter 15, WRITING USER PROGRAMS, presents some general considerations that you should follow when you are creating FORTRAN source programs. In addition, this chapter contains a description of the FORTRAN optimizer.

Chapter 16, USING THE FORTRAN COMPILER, describes how to use the FORTRAN compiler and contains descriptions on how to compile, load, and execute a FORTRAN program. In addition, this chapter contains descriptions of how to read a compiler-generated program listing, and how to create a reentrant FORTRAN program. This Chapter also describes how to use FORTRAN-20 extended addressing.

Chapter 17, USING THE FORTRAN INTERACTIVE DEBUGGER (FORDDT), describes how to use the FORTRAN interactive debugging program (FORDDT) to test and debug a running program. This chapter also contains a brief explanation of how to debug a running FORTRAN program using DDT, the system debugger.

Chapter 18, USING THE FORTRAN OBJECT-TIME SYSTEM (FOROTS), describes the FORTRAN Object-Time System (FOROTS). This chapter also contains descriptions of how you can use the FOROTS software.

Chapter 19, USING THE FORTRAN REAL-TIME SOFTWARE (TOPS-10 ONLY), describes how to use the FORTRAN real-time software. This chapter is for TOPS-10 installations only.

1.2.4 APPENDIXES

The appendixes describe various useful information. The following topics are covered in the appendixes:

Appendix A, SUMMARY OF FORTRAN STATEMENTS, summarizes the forms of all FORTRAN statements and provides a section reference where each statement is described in detail.

Appendix B, ASCII-1968 CHARACTER CODE SET, lists the character code set defined in the X3.4-1968 version of the American National Standard Code for Information Interchange (ASCII).

Appendix C, COMPILER MESSAGES, describes the FORTRAN compiler messages.

Appendix D, FOROTS ERROR MESSAGES, describes the FOROTS error messages.

Appendix E, FORDDT ERROR MESSAGES, describes the FORDDT error messages.

Appendix F, FORTRAN-SUPPLIED PLOTTER SUBROUTINES, describes the FORTRAN-supplied plotter subroutines.

CHAPTER 2

CHARACTERS AND LINES

The basic elements of the FORTRAN source program are its characters and lines. Characters are used to form statements, expressions, and comments in FORTRAN source programs. Lines, and fields within lines, are used to define the context in which characters are interpreted by the FORTRAN compiler.

This chapter describes the relationships among source program characters, lines, and fields within source program lines.

2.1 CHARACTER SET

Table 2-1 lists the digits, letters, and symbols recognized by FORTRAN. The remainder of the ASCII-1968 character set is acceptable within character or Hollerith constants or comment text, but these characters cause fatal errors in other contexts.

NOTE

> The complete ASCII character set is defined in the X3.4-1968 version of the "American National Standard Code for Information Interchange". A summary of the standard ASCII set is also contained in Appendix B of this manual.

NOTE

> Lowercase alphabetic characters are treated as upper-case outside the context of Hollerith or character constants.

Table 2-1: FORTRAN Character Set

Letters	
Uppercase:	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
Lowercase:	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

Digits
0 1 2 3 4 5 6 7 8 9

Symbols			
!	Exclamation point	,	Comma
"	Quotation Mark	-	Hyphen (Minus)
#	Number Sign	.	Period (Decimal Point)
$	Dollar Sign	/	Slant (Slash)
&	Ampersand	:	Colon
'	Apostrophe	;	Semicolon
(	Left Parenthesis	<	Less Than
)	Right Parenthesis	=	Equal To
*	Asterisk	>	Greater Than
+	Plus	^	Circumflex

Line Termination Characters
Line Feed (LF), Form Feed (FF), Vertical Tab (VT)

Line Formatting Characters
Carriage Return (RET), Horizontal Tab (TAB), Blank

Note that horizontal tabs normally advance the character position pointer to the next position that is an even multiple of 8. An exception to this is the initial tab, which is defined as a tab that either includes or starts in character position 6. (Refer to Section 2.3.1 for a description of initial and continuation line types.) Tabs within character specifications count as one character, even though they may advance the character position as many as eight places.

2.2 STATEMENT DEFINITION AND FORMAT

Source program statements are divided into physical lines. A line is defined as a string of adjacent character positions, terminated by the first occurrence of a line termination character, regardless of context. As shown in Figure 2-1, each source program line is divided into four fields.

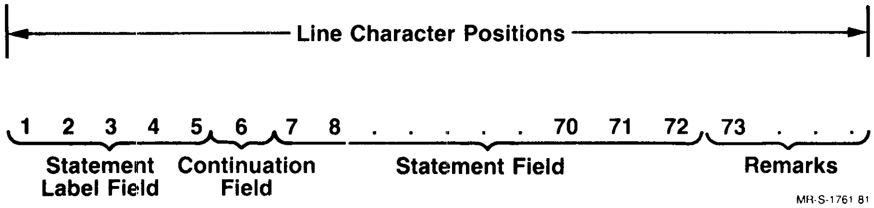


Figure 2-1: Fields Within a FORTRAN Line

2.2.1 Statement Label Field and Statement Numbers

You can place a number ranging from 1 to 99999 in the statement label field of an initial line to identify the statement. Any source program statement that is referenced by another statement must have a statement number. Leading zeros and all blanks in the label field are ignored; for example, the numbers 00105 and 105 are both accepted as statement number 105.

You can assign the statement numbers in a source program in any order; however, each statement number must be unique with respect to all other statements in the program or subprogram.

A main program and a subprogram can contain identical statement numbers. In this case, references to these numbers are understood to mean the numbers in the same program unit in which the reference is made. An example follows:

> Assume that main module MAINMD and subprogram SUB1 both contain statement number 105. A GO TO 105 statement, for instance, in MAINMD will refer to statement 105 in MAINMD, not to 105 in SUB1. A GO TO 105 in SUB1 will transfer control to 105 in SUB1.

An initial tab may be used to skip all or part of the label field. If an initial tab is encountered during compilation, FORTRAN-10/20 examines the character immediately following the tab to determine the type of line being entered. If this character is one of the digits 1 through 9, FORTRAN-10/20 considers the line as a continuation line and uses the second character after the tab as the first character of the statement field.

If the character following the tab is not one of the digits 1 through 9, FORTRAN-10/20 considers the line to be an initial line and the character following the tab is considered to be the first character of the statement field. The character following the initial tab is considered to be in character position 6 for a continuation line, and in character position 7 for an initial line.

2.2.2 Line Continuation Field

Any character of the FORTRAN character set (except a blank, a zero, or an exclamation point) placed in the line continuation field (position 6) identifies the line as a continuation line (see Section 2.3.1). Whenever you use an initial tab to skip all or part of the label field of a continuation line, the next character you enter must be one of the digits 1 through 9 to identify the line as a continuation line.

2.2.3 Statement Field

Any FORTRAN statement can appear in the statement field. Blanks (spaces) and tabs do not affect compilation of the statement. Blanks and tabs may be used freely in this field for appearance purposes, with the exception of textual data given within either a character or Hollerith specification, where blanks and tabs are significant characters.

2.2.4 Remark Field

In lines consisting of 73 or more character positions, only the first 72 character positions are interpreted by FORTRAN. Note that tabs generally occupy more than one character position, usually advancing the cursor to the next character position that is an even multiple of 8. The exception is the tab in a label field, which advances the cursor either to column 6 or 7, depending on the character following the tab (see Section 2.2.1).

All other characters after character position 72 are treated as remarks and do not affect compilation.

Note that remarks may also be added to a line in character positions 1 through 72, provided the text of the remark is preceded by the symbol "!" (see Section 2.3.3).

2.3 LINE TYPES

A line in a FORTRAN source program may be:

1. An initial line
2. A continuation line
3. A multi-statement line
4. A comment line
5. A debug line
6. A blank line

These lines are described in Sections 2.3.1 through 2.3.5.

2.3.1 Initial and Continuation Lines

A FORTRAN statement may occupy the statement fields of up to 20 consecutive lines. The first line in a multi-line statement is referred to as the initial line; the succeeding lines are referred to as continuation lines. Initial lines may be assigned a statement number and must have either a blank or a zero in character position 6.

An initial tab may be used to skip all or part of the label field. If you use an initial tab for this purpose, you must immediately follow it with a nonnumeric character; that is, the first character of the statement field must be nonnumeric.

You cannot assign a statement label to a continuation line. Instead, you identify a continuation line by placing a character from the FORTRAN character set (except blank, zero, or exclamation point) in character position 6 of that line. This position is the line continuation field. The label field of a continuation line must be blank.

Note that blank lines, comments, and debug lines that are treated like comments (that is, debug lines that are not compiled with the rest of the program) are legal continuation lines and do not terminate a continuation sequence (see Section 2.3.4).

The following is an example of a 3-line FORTRAN FORMAT statement with two continuation lines:

```
105   FORMAT (1X,'This example shows how continuation lines ',
     2 'are used to accommodate FORTRAN statements that do not ',
     3 'entirely fit on a single line.')
```

In this example the characters 2 and 3 in position 6 identify those lines as continuation lines.

2.3.2 Multi-Statement Lines

You may write more than one FORTRAN statement in the statement field of one line. The rules for structuring a multi-statement line are:

1. Successive statements must be separated by a semicolon (;).
2. Only the first statement in the series can have a statement number.
3. The last statement in a line is continued to the next line if that next line is made a continuation line.

An example of a multi-statement line is:

```
450     DIST=RATE * TIME; TIME=TIME+0.05; CALL PRIME(TIME,DIST)
```

NOTE

> If a statement sequence in a multi-statement line consists of a logical IF (see Section 9.2.2) followed by any other executable statement, then the statement following the IF will be executed in all cases, even if the IF expression evaluates as false.

2.3.3 Comment Lines and Remarks

Lines that contain descriptive text only are called comment lines. Comment lines commonly identify and introduce a source program, describe the purpose of a particular set of statements, and introduce subprograms.

To structure a comment line:

1. You must place one of the characters C (or c), *, $, /, or ! in character position 1 of the line to identify it as a comment line.

2. You place the text of the comment in the remainder of the line.

3. You may place comment lines anywhere in the source program, including preceding a continuation line.

4. You may write a large comment as a sequence of any number of lines; however, each line must carry the identifying character (C (or c), *, $, /, or !) in its first character position.

The following is an example of a comment that occupies more than one line:

```
C    SUBROUTINE - A12
C    This subroutine formats
c    and stores the results of
c    the HEAT-TEST program
```

Comment lines are printed on all listings, but are otherwise ignored by the compiler.

You may add a remark to any statement field, in character positions 7 through 72, provided the symbol ! precedes the text. For example, in the line

```
IF(N.EQ.0)STOP     ! Stop if card is blank
```

the text "Stop if card is blank" is identified as a remark by the preceding ! symbol. The compiler ignores all characters from the exclamation point to the end of the line. The characters following the exclamation point, however, appear in the source program listing. To be treated as a remark symbol, the exclamation point must not appear in a Hollerith or character constant.

Note that characters appearing in character positions 73 and beyond are automatically treated as remarks, so that you need not use the symbol ! (see Section 2.2.4).

2.3.4 Debug Lines

As an aid in program debugging, a D (or d) in character position 1 of any line causes the line to be interpreted as a comment line; that is, not compiled with the rest of the program unless the /INCLUDE switch is present in the compiler command string. (See Chapter 16 for a description of the compiler switches.)

When the /INCLUDE switch is present in the compiler command string, the D (or d) in character position 1 is treated as a blank so that the remainder of the line is compiled as an ordinary (noncomment) line. A debug line can have a label following the D (or d). Note that if the debug statement is an initial line, all of its continuation lines must contain a D (or d) in character position 1.

2.3.5 Blank Lines

You may insert lines consisting of only blanks, tabs, or no characters anywhere in a FORTRAN source program. Blank lines that contain remarks only, are considered as blank lines. Blank lines are used for formatting purposes only; they cause blank lines to appear in their corresponding positions in source program listings; otherwise, they are ignored by the compiler.

2.4 LINE-SEQUENCED SOURCE FILES

FORTRAN-10/20 accepts line-sequenced files as produced by line-oriented text editors (for example, SOS on TOPS-10 or EDIT on TOPS-20). These sequence numbers are used in place of the listing line numbers normally generated by FORTRAN. The listing line numbers are not the same as FORTRAN statement numbers.

CHAPTER 3

CONSTANTS

3.1 INTRODUCTION

Constants are quantities that do not change value during the execution of the object program. The data types you can use for constants in FORTRAN-10/20 source programs are:

1. Integer
2. Real
3. Double-precision
4. Complex
5. Character
6. Logical
7. Octal
8. Double-octal
9. Hollerith
10. Statement label

The use and format of each of these data types are discussed in Sections 3.2 through 3.10.

3.2 INTEGER CONSTANTS

An integer constant is a string of one to eleven digits that represents a whole decimal number (a number without a fractional part). Integer constants must be within the range of -(2**35-1) to (+2**35)-1 (-34359738367 to +34359738367). Positive integer constants may optionally be signed; negative integer constants must always be signed. You cannot use decimal points, commas, or other symbols in integer constants (except for a preceding sign, + or -).

Examples of valid integer constants are:

```
 345
+345
-345
```

Examples of invalid integer constants are:

+345.	(use of decimal point)
3,450	(use of comma)
34.5	(use of decimal point; not a whole number)

3.3 REAL CONSTANTS

A real constant can have any of the following forms:

1. A basic real constant: a string of decimal digits followed by a decimal point, followed optionally by a decimal fraction, for example, 1557.42.
2. A basic real constant followed by a decimal integer exponent written in E notation (exponential notation) form, for example, 1559.E2 or 1559.e2. The number following the E (or e) specifies a power of ten by which the basic real constant will be multiplied.
3. An integer constant (no decimal point) followed by a decimal integer exponent written in E notation, for example, 1559E2 or 1559e2.

Real constants may be of any size; however, each will be rounded to fit the precision of 27 bits (7 to 9 decimal digits).

Precision for real constants is maintained to approximately eight significant digits; the absolute precision depends upon the numbers involved.

The exponent field of a real constant written in E notation cannot be empty (blank); it must be either a zero or an integer constant. The range of magnitude permitted a real constant is from approximately 1.47 * 10**(-39) to 1.70 * 10**(+38).

The following are examples of valid real constants:

-98.765	
7.0E+0	(= 7.)
.7E-3	(= .0007)
5E+5	(= 500000.)
50115.	
50.E1	(= 500.)

The following are examples of invalid real constants:

72.6E512	(exponent is too large)
.375E	(exponent incorrectly written)
500	(no decimal point given)

3.4 DOUBLE-PRECISION CONSTANTS

Double-precision constants are similar to real constants written in E notation form; the differences between these two constants are:

1. Double-precision constants, depending on their magnitude, have precision from 16 to 18 places, rather than the 8-digit precision obtained for real constants.

2. Each double-precision constant occupies two storage locations.

3. The letter D (or d), instead of E, is used in double-precision constants to identify a decimal exponent.

On KL model B systems, there are two forms of double-precision number formats. If the /GFLOATING compiler switch is specified (see Chapter 16), the double-precision number format is called G-floating. If the /DFLOATING compiler switch (the default) is specified (see Chapter 16), the double-precision number format is called D-floating. See Section 3.4.1 for a comparison of the different double-precision number formats.

On KS systems, only the D-floating double-precision number format is provided.

You must use both the letter D and an exponent (including zero) in writing a double-precision constant. The range of magnitude permitted a double-precision constant is from approximately:

```
     1.47 * 10**(-39) to 1.70 * 10**(+38) for D-floating

                                       or

     2.78 * 10**(-309) to 8.99 * 10**(+307) for G-floating
```

The following are examples of valid double-precision constants:

```
     7.9D03      (= 7900.)
     7.9D+03     (= 7900.)
     7.9D-3      (= .0079)
     79D03       (= 79000.)
     79D0        (= 79.)
```

The following are examples of invalid double-precision constants:

```
     7.9D999     (exponent is too large)
     7.9E5       ("E" denotes single precision; "D" denotes double
                 precision)
```

3.4.1 Comparison of Real, D-floating, and G-floating

For KL model B systems, G-floating double-precision is provided as an alternative double-precision number format. You must specify the /GFLOATING compiler switch (see Chapter 16) to invoke the G-floating double-precision format. If you specify the /DFLOATING compiler switch (the default), the D-floating format is used. Table 3-1 summarizes the comparisons among real, D-floating, and G-floating.

Table 3-1: Comparison of Real, D-floating, and G-floating Numbers

	Bits of Exponent	Bits of Mantissa	Range	Digits of Precision
Real	8	27	1.47 * 10**(-39) to 1.70 * 10**(+38)	8.1
D-floating	8	62	1.47 * 10**(-39) to 1.70 * 10**(+38)	18.7
G-floating	11	59	2.78 * 10**(-309) to 8.99 * 10**(+307)	17.8

3.5 COMPLEX CONSTANTS

You can represent a complex constant by an ordered pair of integer, real, or octal constants written within parentheses and separated by a comma. For example, (.70712, -.70712) and (8.763E3, 2.297) are complex constants.

In a complex constant, the first (leftmost) constant of the pair represents the real part of the number; the second constant represents the imaginary part of the number. Both the real and imaginary parts of a complex constant can be signed.

The constants that represent the real and imaginary parts of a complex constant occupy two consecutive storage locations in the object program.

3.6 CHARACTER CONSTANTS

A character constant is a string of printable ASCII characters enclosed by apostrophes. Both delimiting apostrophes must be present, and the string must be at least one character in length. The compiler accepts control characters in character constants with the following exceptions:

Character	Octal Value
^@ - NUL	"0
^J - LF	"12
^K - VT	"13
^L - FF	"14
^M - CR	"15

NOTE

> The CHAR function (see Chapter 13) can be used to build variables that contain these control characters.

The value of a character constant is the string of characters between the delimiting apostrophes. The value does not include the delimiting apostrophes, but does include all spaces or tabs within the apostrophes.

Within a character constant, the apostrophe character is represented by two consecutive apostrophes (with no space or other character between them).

The length of the character constant is the number of characters between the apostrophes, except that two consecutive apostrophes count as a single apostrophe.

Each character in the string has a character position that is numbered consecutively starting at one. The number indicates the sequential position of a character in a string, from left to right. There is one character storage location for each character in the string.

If a character constant appears in a numeric context (for example, as the expression on the right side of an arithmetic assignment statement), it is considered a Hollerith constant (see Section 3.9).

Examples of valid character constants and their lengths are:

	Length	Value
'WHAT?'	5	WHAT?
'TODAYS''S DATE IS: '	18	TODAY'S DATE IS:
'He said, "hello"'	16	He said, "hello"
''''	1	'

Examples of invalid character constants are:

'HEADINGS	(no trailing apostrophe)
''	(a character constant must contain at least one character)
"Now or Never"	(quotation marks cannot be used in place of apostrophes)

3.7 OCTAL AND DOUBLE-OCTAL CONSTANTS

You may use octal numbers (radix 8) as constants in arithmetic expressions, logical expressions, and data statements. Octal numbers up to 12 digits in length are considered standard octal constants; they are stored right-justified in one storage location. When necessary, standard octal constants are padded with leading zeros to fill their storage location.

If you specify more than 12 digits in an octal number, it is considered a double-octal constant. Double-octal constants occupy two storage locations and may contain up to 24 right-justified octal digits; leading zeros are added to fill any unused digits.

If you assign a single-octal constant to a double-precision or complex variable, it is stored right-justified in the high-order word of the variable. The low-order portion of the variable is set to zero. If you assign a double-octal constant to a double-precision or complex variable, it is stored right-justified in the two words.

All octal constants must:

1. Be preceded by a double quote (") to identify the digits as octal, for example, "777

2. Be signed if negative, but optionally signed if positive

3. Contain one or more of the digits 0 through 7, but not 8 or 9

The following are examples of valid octal constants:

```
"123456700007
+"12345 (optional sign)
-"7777
"-7777
```

The following are examples of invalid octal constants:

```
"12368    (contains an 8)
7777      (no identifying double quote)
```

When you use an octal constant as an operand in an expression, its form (bit pattern) is not converted to accommodate it to the type of any other operand. For example, the subexpression (A + "202400000000) has as its result the sum of A with the floating point number 2.0; while the subexpression (I + "202400000000) has as its result the sum of I with a large integer.

You cannot use octal constants as stand-alone arguments for library functions that require non-octal arguments. MIN0, for instance, requires integer arguments and cannot accept octal arguments.

When you combine a double-octal constant in an expression with (or assign it to) either an integer or real variable, only the contents of the high order location (leftmost) are used.

3.8 LOGICAL CONSTANTS

The Boolean values of truth and falsehood are represented in FORTRAN source programs as the logical constants .TRUE. and .FALSE.. Always write logical constants enclosed by periods, as in the preceding sentence.

You may use logical quantities in arithmetic and logical statements. Only the sign of a numeric value used in a logical IF statement is tested to determine if it is true (negative) or false (nonnegative).

3.9 HOLLERITH CONSTANTS

A Hollerith constant is a string of alphanumeric and/or special characters preceded by nH (for example, nHstring). In the prefix nH, the letter n represents a number that specifies the exact number of characters (including blanks) that follow the letter H.

NOTE

> If a character constant appears in a numeric context it is considered a Hollerith constant (see Section 3.6).

The following are examples of Hollerith constants:

```
2HAB
14HLOAD TEST #124
6H#124-A
```

NOTE

> A tab in a Hollerith constant is counted as one character; for example, 3H   AB.

You may enter Hollerith constants into DATA statements as a string of:

1. Up to ten 7-bit ASCII characters for complex or double-precision type variables

2. Up to five 7-bit ASCII characters for all other type variables

The 7-bit ASCII characters that comprise a Hollerith constant are stored left-justified (starting in the first word of a double-precision constant (the high-order word) or the real part of a complex constant) with blanks placed in empty character positions. Hollerith constants that occupy more than one variable are stored as successive variables in the list. The following example illustrates how the string of characters is stored in a six-element array called A:

```
DIMENSION A(6)
DATA A/27HA string of many characters/

A(1) is set to 'A str'
A(2) is set to 'ing o'
A(3) is set to 'f man'
A(4) is set to 'y cha'
A(5) is set to 'racte'
A(6) is set to 'rs   '
```

3.10 STATEMENT LABEL CONSTANTS

Statement labels are numeric identifiers that represent program statement numbers.

You write statement label constants as strings of one to five decimal digits, that are preceded by either an asterisk (*), a dollar sign ($), or an ampersand (&). For example, *11992, $11992, and &11992 are all valid statement label constants. You use statement label constants only in the argument list of CALL statements to identify the number of the executable statement to return to in a multiple RETURN statement (see Chapter 13).

CHAPTER 4

SYMBOLIC NAMES, VARIABLES, AND ARRAYS

4.1 SYMBOLIC NAMES

Symbolic names consist of any alphanumeric combination of one to six characters, the first of which must be a letter. If you use more than six characters in a symbolic name, the compiler prints a warning message and ignores all but the first six characters. The compiler interprets lowercase letters in symbolic names as uppercase letters.

The following are examples of legal symbolic names:

```
A12345
IAMBIC
ABLE
C
```

The following are examples of illegal symbolic names:

```
.AMBIC     (first character is not a letter)
8AB        (first character is not a letter)
```

You use symbolic names to identify specific items of a FORTRAN source program; Table 4-1 lists these items, together with an example of a symbolic name and text reference for each.

Table 4-1: Use of Symbolic Names

Symbolic Names Can Identify	For Example	For a Detailed Description See Section
1. Variables	PI, CONST, LIMIT	4.2
2. Arrays	TAX	4.3
3. Array elements	TAX(3,5)	4.3.1
4. Substrings	FOO(1:N)	4.4
5. Functions	MYFUNC, VALFUN	13.2
6. Subroutines	CALCSB, SUB2, LOOKUP	13.4
7. Intrinsic functions	SIN, ATAN, COSH	13.1
8. PROGRAM Statement	TEST	6.4.1
9. PARAMETER Statement	V1,C2,K	7.8
10. COMMON block names	DATAR, COMDAT	7.4
11. NAMELIST list	DATA3	12.6

4.2 VARIABLES

A variable is a data storage location identified by a symbolic name; a variable is not a constant, an array, or an array element. Variables specify values that are assigned to them in such ways as assignment statements (Chapter 8), DATA statements (Chapter 7), or at run time through I/O data transfers (Chapter 10). Before you assign a value to a variable, its value is undefined; and you should not reference it except to assign a value to it.

The value you assign to a variable can be either a constant or the result of a calculation that is performed during the execution of the object program. For example, the statement IAB=5 assigns the constant 5 to the variable IAB. In the statement IAB=5+IB, however, the value assigned IAB depends on the value of variable IB at the time the statement is executed.

The type of a variable determines the interpretation of its contents. Variables can be:

1. Integer
2. Real
3. Logical
4. Double-precision
5. Complex
6. Character

The type of a variable is determined either implicitly, by the first letter of the variable name (described below), or explicitly, by declaring the variable type in a type declaration statement (see Chapter 7).

FORTRAN uses the following default conventions for variables whose types are not explicitly declared:

1. Variable names that begin with the letters I, J, K, L, M, or N are integer variables.
2. Variable names that begin with any letter other than I, J, K, L, M, or N are real variables.

NOTE

> These default conventions can be altered by use of the IMPLICIT statement, which is described in Section 7.3.

The following are examples of determining the type of a variable according to the preceding conventions:

Variable	Beginning Letter	Assumed Data Type
ITEMP	I	Integer
OTEMP	O	Real
KA123	K	Integer
AABLE	A	Real

4.3 ARRAYS

An array is an ordered set of data identified by an array name. Array names are symbolic names and must conform to the rules for writing symbolic names (see Section 4.1).

Arrays are made up of smaller units of data called array elements. As with variables, you may assign a value to an array element. Before you assign a value to an array element it has an undefined value. You should not reference an array element until you have assigned it a value.

An array element is referenced by using the array name together with some number of subscripts that describe the position of the element within the array.

4.3.1 Array Element Subscripts

The general form of an array element name is AN (S1, S2,...Sn), where AN is the array name and S1 through Sn represent 1 through n subscript expressions. You may use any number of subscript quantities in an element name; however, the number used must always equal the number of dimensions (see Section 4.3.2) specified for the array.

A subscript can be any constant or expression (see Chapter 5), for example:

1. Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, (A+B,C*5,D/2) and (A**3,(B/4+C)*E,3) are valid subscripts.

2. Arithmetic expressions (see Chapter 5) used in array subscripts may be of any type, but noninteger expressions (including complex) are converted to integer when the subscript is evaluated.

3. A subscript may contain function references (see Chapter 13). For example, TABLE (SIN(A)*B,2,3) is a valid array element identifier.

4. Subscripts may contain array element identifiers nested to any level as subscripts. For example, in the subscript (I(J(K(L))),A+B,C) the first subscript expression given is a nested 3-level array reference.

Some examples of valid array elements are:

1. IAB(1,5,3)

2. ABLE(A)

3. TABLE1(10/C+K**2,A,B)

4. MAT(A,AB(2*L),.3*TAB(A,M+1,D),55)

4.3.2 Dimensioning Arrays

You must declare the size (number of elements) of an array to enable FORTRAN to reserve the number of locations needed to store the array. Arrays are stored as a series of sequential storage locations. Arrays, however, are visualized and referenced as if they were single or multi-dimensional, rectilinear matrices dimensioned on a row, column, and plane basis. For example, Figure 4-1 represents a 3-row, 3-column, 2-plane array.

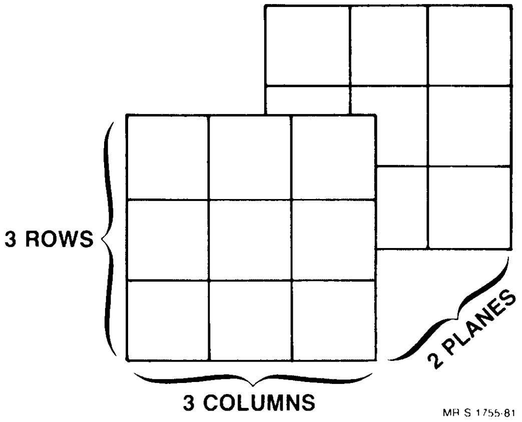


Figure 4-1: A 3 x 3 x 2 Array

You specify the size of an array by an array declarator written as a subscripted array name. In an array declarator each subscript quantity is a dimension of the array and must be either an integer expression, an integer variable, or an asterisk (*).

Only the upper bound in the last dimension declarator in a list of dimension declarators can be an asterisk. An asterisk marks the declarators as an assumed-size array declarator (see Section 7.1.2).

NOTE

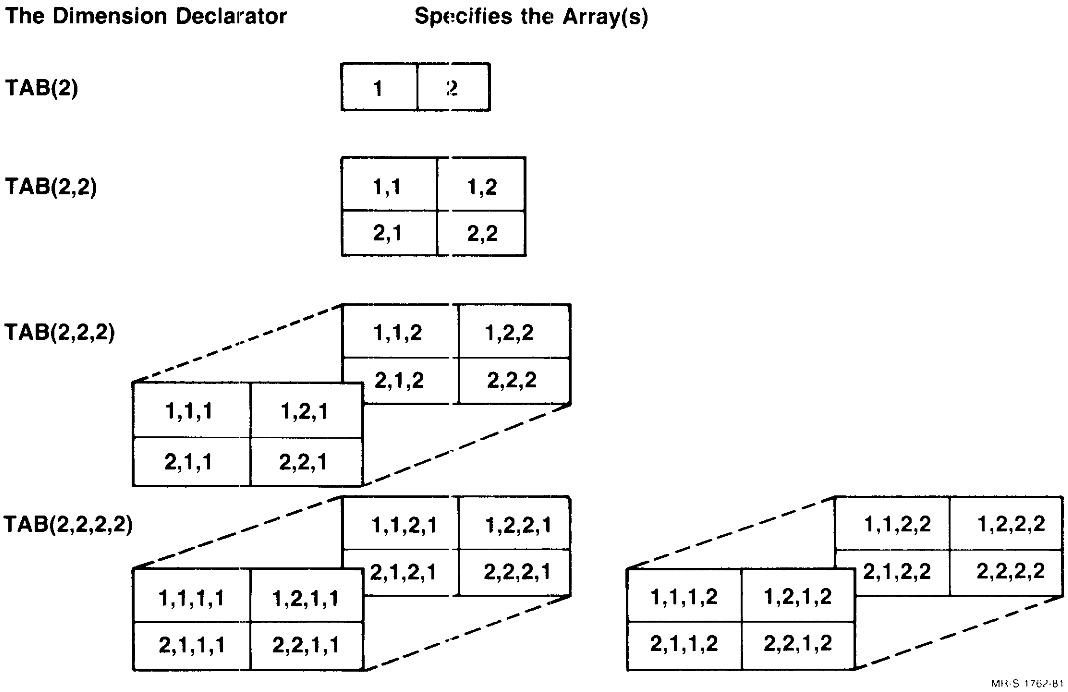
> Variable array dimensions are only allowed in subprograms. See adjustable dimension statements, Section 7.1.1.

For example, TABLE(I,J,K) and MATRIX(10,7,3,4) are valid array declarators.

The total number of elements that comprise an array is the product of the dimension quantities given in its array declarator. For example, the array IAB dimensioned as IAB(2,3,4) has 24 elements (2 * 3 * 4 = 24).

You dimension arrays only in the specification statements DIMENSION, COMMON, and type declaration (see Chapter 7). Subscripted array names appearing in any of the these statements are array declarators; subscripted array names appearing in any other statements are always array element identifiers.

In array declarators, the position of a given subscript quantity determines the particular dimension of the array (for example, row, column, or plane) that it represents. The first three subscript positions specify the number of rows, columns, and planes that comprise the named array; each following subscript given then specifies a set comprised of n-number (value of the subscript) of the previously defined sets. For example:



NOTE

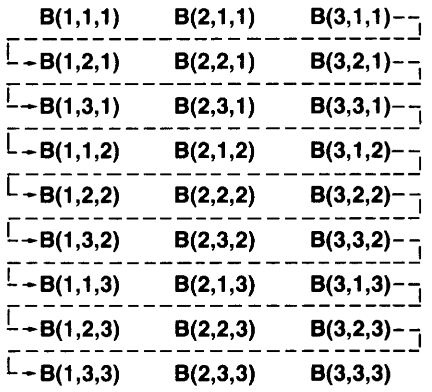
FORTRAN-10/20 permits up to 127 dimensions in an array declarator. (The FORTRAN-77 Standard allows a maximum of 7 dimensions.)

4.3.3 Order of Stored Array Elements

The elements of an array are stored in ascending order. The value of the first (leftmost) subscript varies between its minimum and maximum values most rapidly. The value of the last (rightmost) subscript increases to its maximum value least rapidly. For example, the elements of the array dimensioned as I(2,3) are stored in the following order:

```
I(1,1)  I(2,1)  I(1,2)  I(2,2)  I(1,3)  I(2,3)
```

In the following list, the elements of the three-dimensional array (B(3,3,3)) are stored row by row from left to right and from top to bottom.


B(1,1,1) B(2,1,1) B(3,1,1)
B(1,2,1) B(2,2,1) B(3,2,1)
B(1,3,1) B(2,3,1) B(3,3,1)
B(1,1,2) B(2,1,2) B(3,1,2)
B(1,2,2) B(2,2,2) B(3,2,2)
B(1,3,2) B(2,3,2) B(3,3,2)
B(1,1,3) B(2,1,3) B(3,1,3)
B(1,2,3) B(2,2,3) B(3,2,3)
B(1,3,3) B(2,3,3) B(3,3,3)

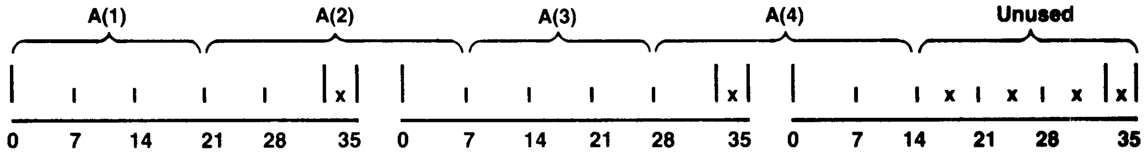
MR-S-1756-81

Thus B(3,1,1) is stored before B(1,2,1), and so forth.

Character array elements are stored in successive character positions, and do not necessarily start on a word boundary. Character array elements are stored five characters per word (seven bits per character), and the low order bit is never used, for example:

```
CHARACTER*3 A(4)
```

The array A will be stored in the following way:



MR-S-2528-83

where:

x   means bits are not used. The value in bit 35 is zero.

4.4 CHARACTER SUBSTRINGS

A character substring is a contiguous segment of a character variable or character array element. A character substring is identified by a substring name and can be assigned values and referenced.

A character substring reference has one of the following forms:

```
v([e1]:[e2])

    or

a(s[,s]...) ([e1]:[e2])
```

where:

v is a character variable name.

a is a character array name.

s is a subscript expression.

e1 is an optional numeric expression that specifies the leftmost character position of the substring.

e2 is an optional numeric expression that specifies the rightmost character position of the substring.

Character positions within a character variable or array element are numbered from left to right, beginning at 1. For example, LABEL(2:7) specifies the substring beginning with the second character position and ending with the seventh character position of the character variable LABEL.

If the value of the numeric expression e1 or e2 is not of type integer, FORTRAN converts it to an integer value by truncating any fractional part before use.

The values of the numeric expression e1 and e2 must meet the following conditions:

```
1 .LE.  e1 .LE.  e2 .LE.  len
```

where:

len is the length of the character variable or array element.

If e1 is omitted, FORTRAN assumes that e1 is 1. If e2 is omitted, FORTRAN assumes that e2 equals len.

For example, NAMES(1,3)(:7) specifies the substring starting with the first character position and ending with the seventh character position of the character array element NAMES(1,3).

CHAPTER 5

EXPRESSIONS

5.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is formed with arithmetic operands and arithmetic operators. The evaluation of such an expression produces a numeric value.

Arithmetic expressions may be either simple or compound. A simple arithmetic expression consists of an operand that can be:

1. A numeric constant

2. A numeric variable

3. A numeric array element

4. An arithmetic function reference (see Chapter 13)

5. An arithmetic or logical expression written within parentheses

Operands may be of integer, real, double-precision, complex, logical, octal, double-octal, or Hollerith type.

The following are examples of valid simple arithmetic expressions:

105	(integer constant)
IAB	(integer variable)
TABLE(3,4,5)	(array element)
SIN (X)	(function reference)
(A+B)	(a parenthetical expression)

A compound arithmetic expression consists of two or more operands combined by arithmetic operators. Table 5-1 lists the arithmetic operations permitted in FORTRAN and the operator recognized for each operation.

Table 5-1: Arithmetic Operations and Operators

Operation	Binary Operator	Example
Addition	+	A+B
Subtraction	-	A-B
Multiplication	*	A*B
Division	/	A/B
Exponentiation	** or ^	A**B or A^B
Operation	**Unary Operator**	**Example**
Identity	+	+A
Negation	-	-B

5.1.1 Rules for Writing Arithmetic Expressions

Observe the following rules in structuring arithmetic expressions:

1. The operands comprising an arithmetic expression can be of different types. Tables 5-2 and 5-3 illustrate all permitted combinations of data types and the type assigned to the result of each.

   NOTE

   All combinations of numeric data types except double-precision with complex are allowed in FORTRAN.

2. If you specify two adjacent operators, and the second is a minus or a plus, the second operator is considered a unary operator and acts only on the term immediately following it. Thus, in the example (A*X+B)*+C, the subexpression, *+C, is interpreted as the binary operator * and the unary +.

   You cannot, however, have two adjacent binary operators in an expression. For example, the expression A*/B is not permitted.

3. All operators must be included; no operation is implied. For example, the expression A(B) does not specify multiplication, although this is implied in standard algebraic notation. The expression A*(B) is required to specify a multiplication of the operands.

Table 5–2: Type of the Result Obtained from Mixed-Mode Operations

		Type of Argument 2							
Type of Argument 1	For operators +,-,*,/	Integer	Real	Double Precision	Complex	Logical	Octal	Double Octal	Literal
Integer	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. None 4. None	1. Real 2. Real 3. From Integer to Real 4. None	1. Double Precision 2. Double Precision 3. From Integer to Double Precision 4. None	1. Complex 2. Complex 3. From Integer to Complex. Value used as Real part 4. None	1. Integer 2. Integer 3. None 4. None	1. Integer 2. Integer 3. None 4. None	1. Integer 2. Integer 3. None 4. High order word is used directly; low order word is ignored	1. Integer 2. Integer 3. None 4. High order word is used directly; further words are ignored
Real	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Real 2. Real 3. None 4. From Integer to Real	1. Real 2. Real 3. None 4. None	1. Double Precision 2. Double Precision 3. Used directly as the high order word; low order word is zero. 4. None	1. Complex 2. Complex 3. Used directly as the Real part, imaginary part is zero. 4. None	1. Real 2. Real 3. None 4. None	1. Real 2. Real 3. None 4. None	1. Real 2. Real 3. None 4. High order word is used directly; low order word is ignored.	1. Real 2. Real 3. None 4. High order word is used directly, further words are ignored.
Double Precision	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Double Precision 2. Double Precision 3. None 4. From Integer to Double Precision	1. Double Precision 2. Double Precision 3. None 4. Used directly as the high order word; low order word is zero.	1. Double Precision 2. Double Precision 3. None 4. None		1. Double Precision 2. Double Precision 3. None 4. Used directly as the high order word; low order word is zero.	1. Double Precision 2. Double Precision 3. None	1. Double Precision 2. Double Precision 3. None 4. None	1. Double Precision 2. Double Precision 3. None 4. First two words are used directly; further words are ignored
Complex	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Complex 2. Complex 3. None 4. From Integer to Complex. Value used as Real part	1. Complex 2. Complex 3. None 4. Used directly as the Real part; imaginary part is zero.		1. Complex 2. Complex 3. None 4. None	1. Complex 2. Complex 3. None 4. Used directly as the Real part; imaginary part is zero	1. Complex 2. Complex 3. None 4. Used directly as the Real part; imaginary part is zero	1. Complex 2. Complex 3. None 4. None	1. Complex 2. Complex 3. None 4. First two words are used directly. Further words are ignored.
Logical	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. None 4. None	1. Real 2. Real 3. None 4. None	1. Double Precision 2. Double Precision 3. Used directly as the high order word; low order word is zero 4. None	1. Complex 2. Complex 3. Used directly as the Real part imaginary part is zero. 4. None	1. Integer 2. Octal 3. None 4. None	1. Integer 2. Octal 3. None 4. None	1. Integer 2. Octal 3. None 4. High order word is used directly; low order word is ignored	1. Integer 2. Octal 3. None 4. High order word is used directly; further words are ignored
Octal	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. None 4. None	1. Real 2. Real 3. None 4. None	1. Double Precision 2. Double Precision 3. Used directly as the high order word; low order word is zero. 4. None	1. Complex 2. Complex 3. Used directly as the Real part imaginary part is zero. 4. None	1. Integer 2. Octal 3. None 4. None	1. Integer 2. Octal 3. None 4. None	1. Integer 2. Octal 3. None 4. High order word is used directly. low order word is ignored	1. Integer 2. Octal 3. None 4. High order word is used directly. further words are ignored
Double Octal	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. High order word is used directly low order word is ignored 4. None	1. Real 2. Real 3. High order word is used directly low order word is ignored 4. None	1. Double Precision 2. Double Precision 3. None 4. None	1. Complex 2. Complex 3. None 4. None	1. Integer 2. Octal 3. High order word is used directly; low order word is ignored 4. None	1. Integer 2. Octal 3. High order word is used directly; low order word is ignored 4. None	1. Integer 2. Octal 3. High order word is used directly low order word is ignored 4. High order word is used directly, low order word is ignored	1. Integer 2. Octal 3. High order word is used directly low order words are ignored 4. High order word is used directly low order words are ignored
Literal	1. Type of operation used 2. Type associated with result 3. Conversion on Argument 1 4. Conversion on Argument 2	1. Integer 2. Integer 3. High order word is used directly further words are ignored 4. None	1. Real 2. Real 3. High order word is used directly further words are ignored 4. None	1. Double Precision 2. Double Precision 3. First two words are used directly further words are ignored 4. None	1. Complex 2. Complex 3. First two words are used directly, further words are ignored 4. None	1. Integer 2. Octal 3. High order word is used directly; further words are ignored 4. None	1. Integer 2. Octal 3. High order word is used directly; further words are ignored 4. None	1. Integer 2. Octal 3. High order word is used directly, further words are ignored 4. High order word is used directly; low order word is ignored	1. Integer 2. Octal 3. High order word is used directly further words are ignored 4. High order word is used directly further words are ignored

MR-S-1751-81

Table 5-3: Permitted Base/Exponent Type Combinations

Base Operand	Exponent Operand			
	Integer	Real	Double-Precision	Complex
Integer	Integer	Real	Double-Precision	Complex
Real	Real	Real	Double-Precision	Complex
Double-Precision	Double-Precision	Double-Precision	Double-Precision	(Illegal)
Complex	Complex	Complex	(Illegal)	Complex

5.1.2 Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each operand is one of the following:

1. A numeric constant

2. A symbolic name of a numeric constant

3. An arithmetic constant expression enclosed in parentheses

4. A call to the function ICHAR (see Chapter 13) where the argument is a character constant expression

The exponentiation operator is not permitted unless the exponent is of type integer. Note that variables, array elements, and function references are not allowed.

Example:

```
5+6*(ICHAR('Z')-ICHAR('A')+1)*4.1**3
```

5.1.2.1 Integer Constant Expression - An integer constant expression is an arithmetic constant expression in which each constant or symbolic name of a constant is of type integer.

Example:

```
3+4**6+2
```

5.2 CHARACTER EXPRESSIONS

Character expressions consist of character operands and character operators. The evaluation of a character expression yields a single value of character data type.

A character operand can be any one of the following:

1. A character constant
2. A symbolic name of a character constant
3. A character variable
4. A character array element
5. A character substring
6. A character expression, optionally enclosed in parentheses
7. A character function reference

The only character operator is the concatenation operator (//).

A character expression has the form:

```
character operand [//character operand]...
```

The value of a character expression is a character string formed by successive left-to-right concatenations of the value of the elements of the character expression. The length of a character expression is the sum of the lengths of the character elements. For example, the value of the character expression 'AB'//'CDE' is 'ABCDE', which has a length of 5.

Note that the expression:

```
A=A//B
```

has no effect on A, since the concatenation result is truncated to the length of A.

Parentheses do not affect the value of a character expression. For example, the following character expressions are equivalent:

```
('ABC'//'DE')//'F'
'ABC'//('DE'//'F')
'ABC'//'DE'//'F'
```

Each of these character expressions has the value 'ABCDEF'.

If a character element in a character expression contains spaces, the spaces are included in the value of the character expression. For example, 'ABC '//'D E'//'F' has a value of 'ABC D EF'.

5.2.1 Character Constant Expression

A character constant expression is a character expression in which each operand is one of the following:

1. A character constant

2. The symbolic name of a character constant

3. A character constant expression enclosed in parentheses

4. A call to the function CHAR (see Chapter 13) where the argument is an integer constant expression

Variables, array elements, substrings, and function references are not allowed.

Example:

```
'HELLO'//CHAR(13)//CHAR(10)//'GOODBYE'
```

5.3 LOGICAL EXPRESSIONS

Logical expressions can be either simple or compound. Simple logical expressions consist of a logical operand, which can be one of the following:

1. A constant

2. A variable

3. An array element

4. A function reference (see Chapter 13)

5. An expression written within parentheses

Compound logical expressions consist of two or more logical or numeric operands combined by logical operators. The evaluation of a logical expression produces a truth value (type logical, true or false) as determined by the resulting bit pattern.

Table 5-4 gives the logical operators permitted by FORTRAN and a description of the operation each provides.

Table 5-4: Logical Operators

Operator	Description
.AND.	AND operator. Both of the logical operands combined by this operator must be true to produce a true result.
.OR.	Inclusive OR operator. If either or both of the logical operands combined by .OR. are true, the result will be true.
.NEQV.	Exclusive OR operator (also .XOR.). If either but not both of the logical operands combined by .NEQV. is true, the result will be true.
.EQV.	Equivalence operator. If the logical operands being combined by .EQV. are both the same (both are true or both are false), the result will be true.
.NOT.	Complement operator. This operator specifies complementation (inversion) of the item (operand or expression) that it modifies. The original item, if true by itself, becomes false, and vice versa.

Logical expressions are of the general form P .op. Q, where P and Q are logical operands and .op. is any logical operator except ".NOT.". The .NOT. operator complements the value of an operand; it must appear immediately before the operand that it modifies, for example, .NOT.P.

Table 5-5 is a truth table illustrating all possible logical combinations of two logical operands (P and Q) and the result of each combination.

Table 5-5: Logical Operations Truth Table

When P is	And Q is:	Then the Expression:	Is:
True	-----	.NOT. P	False
False	-----	.NOT. P	True
True	True	P .AND. Q	True
True	False	P .AND. Q	False
False	True	P .AND. Q	False
False	False	P .AND. Q	False
True	True	P .OR. Q	True
True	False	P .OR. Q	True
False	True	P .OR. Q	True
False	False	P .OR. Q	False
True	True	P .NEQV. Q	False
True	False	P .NEQV. Q	True
False	True	P .NEQV. Q	True
False	False	P .NEQV. Q	False
True	True	P .EQV. Q	True
True	False	P .EQV. Q	False
False	True	P. EQV. Q	False
False	False	P .EQV. Q	True

For example, consider the following variables:

Variables	Type
PHETT, RUN	Real
I,J,K	Integer
DP,D	Double-Precision
L,A,B	Logical
CPX,C	Complex

Examples of valid logical expressions consisting of the preceding variables are:

```
L.AND.B
(PHETT*I).NEQV.(DP+K)
L.AND.A.OR..NOT.(I-K)
```

Logical operations are performed on the full 36-bit binary representation of the operands involved. However, when an operand of a logical expression is double-precision or complex, only the first word of a double-precision operand (the high-order word) or the real part of the complex operand is used in the specified logical operation.

The result of a logical operation is found by performing the specified operation simultaneously for each of the corresponding bits in each operand. For example, consider the expression A=C.OR.D, where C="456 and D="201. The operation performed by the processor and the result is:

Word Bits	0	1 → 24	25	26	27	28	29	30	31	32	33	34	35
Operand C	0	0 → 0	0	0	1	0	0	1	0	1	1	1	0
Operand D	0	0 → 0	0	0	0	1	0	0	0	0	0	0	1
Result A	0	0 → 0	0	0	1	1	0	1	0	1	1	1	1

Table 5-5 also illustrates all possible logical combinations of two one-bit binary operands (P and Q) and gives the result of each combination. Simply read 1 for true and 0 for false.

If a logical expression is used as an operand in an arithmetic expression, its value is not converted to accommodate it to the type of any other operand.

5.3.1 Logical Constant Expression

A logical constant expression is a logical expression in which each operand is one of the following:

1. A logical constant
2. The symbolic name of a logical constant
3. A relational expression in which each operand is a constant expression
4. A logical constant expression enclosed in parentheses

Variables, array elements, and function references are not allowed.

Example:

```
.NOT.(PARML1.NE.PARML2)
```

where PARML1 and PARML2 are specified in a PARAMETER statement (see Section 7.8).

5.4 RELATIONAL EXPRESSIONS

Relational expressions consist of two arithmetic expressions or two character expressions combined by a relational operator. The relational operator allows you to test the relationship between two arithmetic or two character expressions.

The result of a relational expression is always a logically true or false value.

You can write relational operators either as a 2-letter mnemonic enclosed within periods (for example, .GT.) or use the symbolic equivalent, for example, >, instead of .GT.

Table 5-6 lists the mnemonic and symbolic forms of the FORTRAN-10/20 relational operators and specifies the type of test performed by each.

Table 5-6: Relational Operators and Operations

Operators		Relation Tested
Mnemonic	Symbolic	
.GT.	>	Greater than
.GE.	>=	Greater than or equal to
.LT.	<	Less than
.LE.	<=	Less than or equal to
.EQ.	==	Equal to
.NE.	#	Not equal to

Relational expressions are of the general form A .op. B, where A and B represent arithmetic or character operands, and .op. is a relational operator.

You can mix arithmetic operands of type integer, real, and double-precision in relational expressions.

A relational expression cannot be used to compare the value of an arithmetic expression with the value of a character expression. However, you can compare a numeric expression to a character constant. In this case, the character constant is considered to be a Hollerith (see Section 3.9).

You can compare complex operands using only the operators .EQ. (==) and .NE. (#). Complex quantities are equal if the corresponding parts of both words are equal.

For example, assume the following variables:

Variables	Type
PHETT, RON	Real
I,J,K	Integer
DP,D	Double-Precision
L,A,B	Logical
CPX,C	Complex
CHR,RA	Character

Examples of valid relational expressions consisting of the above variables are:

```
(PHETT).GT.10
I == 5
C.EQ.CPX
CHR.LT.RA
```

Examples of invalid relational expressions consisting of the above variables are:

```
(PHETT).GT 10   (closing period missing from operator)

C.GT.CPX        (complex operands can only be compared by .EQ.
                and .NE.  operators)

RA.EQ.RON       (you cannot compare arithmetic operands and
                character operands)
```

Examples of valid expressions that use both logical and relational operators to combine the preceding variables are:

```
(I.GT. 10).AND.(J.LE.K)
((I*RON).EQ.(I/J)).OR.L
(I.AND.K)#((PHETT).OR.(RON))
C#CPX.OR.RON
```

If a logical expression is used as an operand in an arithmetic expression, its value is not converted to accommodate it to the type of any other operand.

In character relational expressions "less than" means "precedes in the ASCII collating sequence," and "greater than" means "follows in the ASCII collating sequence", for example:

```
'AB'//'ZZZ' .LT.  'CCCCC'
```

This expression tests whether 'ABZZZ' is less than 'CCCCC'. Since that relationship does exist, the value of the expression is true. If the relationship stated does not exist, the value of the expression is false.

If the two character expressions in a relational expression are not the same length, the comparison is performed as if the shorter one is padded on the right with spaces until the lengths are equal, for example:

```
'ABC' .EQ. 'ABC   '

'AB' .LT. 'C'
```

The first relational expression has a value of true even though the lengths of the expressions are not equal, and the second has a value of true even though 'AB' is longer than 'C'.

NOTE

> The rule that character relationals extend the shorter operand with spaces to match the length of the longer operand has an interesting effect when the longer string ends with characters in the range CHAR(0) to CHAR(31) (ASCII control characters such as 'bell' and line feed).
>
> Since space is CHAR(32), the trailing spaces supplied as filler by FORTRAN compare being greater than trailing control characters. Thus, the string 'FOO' is .GT. 'FOO^G' (FOO followed by a bell).

5.5 EVALUATION OF EXPRESSIONS

The following considerations determine the order of computation of a FORTRAN expression:

1. The use of parentheses

2. An established hierarchy for the execution of arithmetic relational, and logical operations

3. The location of operators within an expression

5.5.1 Parenthetical Subexpressions

In an expression, all subexpressions enclosed within parentheses are evaluated first. When parenthetical subexpressions are nested (one contained within another), the most deeply nested subexpression is evaluated first; the next most deeply nested subexpression is evaluated second; and so on, until the value of the final parenthetical expression is computed.

When more than one operator is contained in a parenthetical subexpression, the required computations are performed according to the hierarchy assigned to operators by FORTRAN (see Section 5.5.2).

For example, the separate computations performed in evaluating the expression A+B/((A/B)+C)-C are:

1. R1=A/B

2. R2=R1+C

3. R3=B/R2

4. R4=A+R3

5. R5=R4-C

where:

R1 through R5 represent the interim and final results of the computations performed.

5.5.2 Hierarchy of Operators

The following hierarchy (order of execution) is assigned to the classes of FORTRAN operators:

first, arithmetic operators
second, relational operators
third, logical operators

Table 5-7 specifies the precedence assigned to the individual operators of the above classes.

With the exception of exponentiation and integer division, all operations on expressions or subexpressions involving operators of equal precedence are computed in any order that is algebraically correct.

A subexpression of a given expression may be computed in any order. For example, in the expression (F(X) + A*B), the function reference may be computed either before or after A*B.

Table 5-7: Hierarchy of FORTRAN Operators

Class	Level	Symbol or Mnemonic
EXPONENTIAL	First	** or ^
ARITHMETIC	Second Third Fourth	-(negation) and + (identity) *,/ +,-
RELATIONAL	Fifth	.GT.,.GE.,.LT.,.LE.,.EQ.,.NE. or >,>=,<,<=,==,#
LOGICAL	Sixth Seventh Eighth Ninth	.NOT. .AND. .OR. .EQV.,.NEQV.

Operations specifying integer division are evaluated from left to right. For example, the expression I/J*K is evaluated as if it had been written as (I/J)*K), but this left-to-right evaluation process can be overridden by parentheses. I/J*K (evaluated as(I/J) *K) does not equal I/(J*K).

When a series of exponentiation operations occurs in an expression, it is evaluated in order from right to left. For example, the expression A**2**B is evaluated in the following order:

first Rl = 2**B (intermediate result)
second R2 = A**Rl (final result).

As with other expressions, parentheses alter the evaluation of the above expression. The expression (A**2)**B is evaluated in these two steps:

first Rl = A**2 (intermediate result)
second R2 = Rl**B (final result)

5.5.3 Mixed-Mode Expressions

Mixed-mode expressions are evaluated on a basis of subexpression-by-subexpression, with the type of the results obtained converted and combined with other results or terms according to the conversion procedures described in Table 5-2.

For example, assume the following variables and data types:

Variables	Type
D	Double-Precision
X	Real
I,J	Integer

The mixed-mode expression D+X*(I/J) is evaluated in the following manner:

1. Rl = I/J Rl is integer
2. R2 = X*Rl Rl is converted to type real and is multiplied by X to produce R2
3. R3 = D+R2 R2 is converted to type double-precision and is added to D to produce R3

where:

Rl and R2, and R3 represent the interim and final results respectively of the computations performed.

5.5.4 Use of Logical Operands in Mixed-Mode Expressions

When you use logical operands in mixed-mode expressions, the value of the logical operand is not converted in any way to accommodate it to the type of the other operands in the expression. For example, in L*R, where L is type logical and R is type real, the expression is evaluated without converting L to type real.

5.6 CONSTANT EXPRESSIONS

A constant expression is an arithmetic constant expression (see Section 5.1.2), a character constant expression (see Section 5.2.1), or a logical constant expression (see Section 5.3.1).

CHAPTER 6

EXECUTABLE AND NONEXECUTABLE STATEMENTS

Each statement is classified as executable or nonexecutable. Executable statements specify actions and form an execution sequence in a program. Nonexecutable statements do the following:

1. Specify characteristics, arrangement, and initial values of data

2. Contain editing information

3. Specify statement functions

4. Classify program units

5. Specify entry points within subprograms

Nonexecutable statements are not part of the execution sequence. Nonexecutable statements may be labeled, but such statement labels must not be used to control the execution sequence.

6.1 EXECUTABLE STATEMENTS

The following statements are classified as executable:

1. Arithmetic, logical, statement label (ASSIGN), and character assignment statements

2. Unconditional GO TO, assigned GO TO, and computed GO TO statements

3. Arithmetic IF, logical IF statements, and two-branch logical IF statements, IF THEN, ELSE, and ELSE IF THEN statements

4. CONTINUE statement

5. STOP and PAUSE statements

6. DO and DO WHILE statements

7. READ, REREAD, WRITE, and PRINT statements

8. OPEN and CLOSE statements

9. REWIND, BACKSPACE, ENDFILE, BACKFILE, SKIPRECORD, SKIPFILE, FIND and UNLOAD statements

10. CALL and RETURN statements

11. END, END IF, and END DO statements

12. DECODE and ENCODE statements

13. ACCEPT, PUNCH, and TYPE statements

14. INQUIRE statement

6.2 NONEXECUTABLE STATEMENTS

The following statements are classified as nonexecutable:

1. PROGRAM, FUNCTION, SUBROUTINE, ENTRY, and BLOCK DATA statements

2. DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, PARAMETER, INTRINSIC, EXTERNAL, and SAVE statements

3. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER type-specification statements

4. DATA statement

5. FORMAT statement

6. Statement function statement

7. INCLUDE statement

8. NAMELIST statement

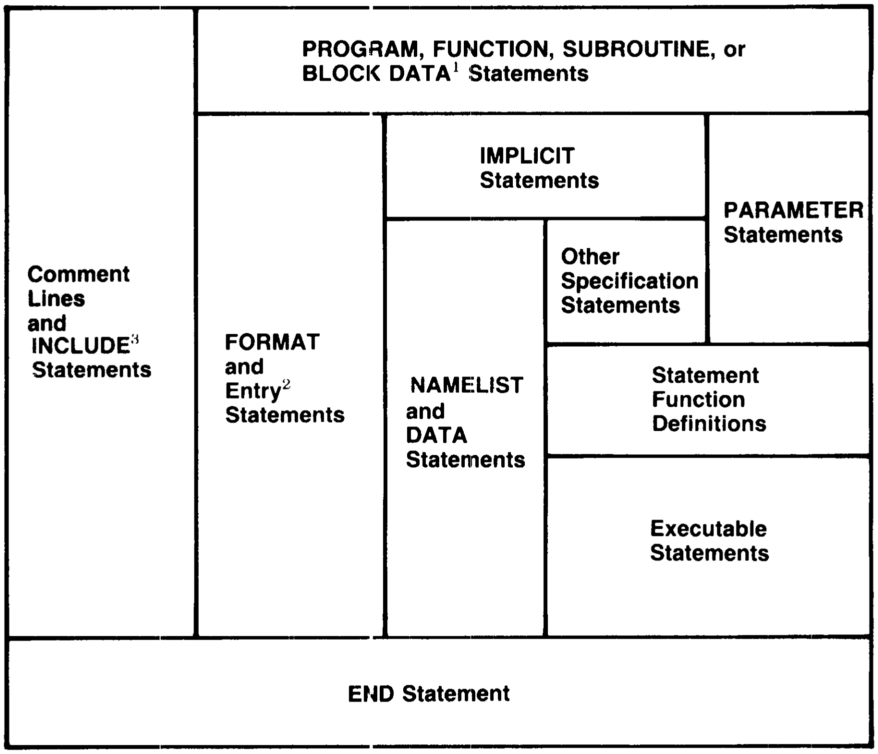
6.3 ORDERING OF FORTRAN STATEMENTS

The order in which you place FORTRAN statements in a program unit is important. Certain types of statements must be processed before others to guarantee that compilation takes place as you expect.

Figure 6-1 shows the required order of statements and comment lines within a program unit. Horizontal lines indicate (from the top of the diagram to the bottom) the order in which statements and comment lines must appear in a program. For example, a PROGRAM statement must occur before FORMAT statements. FORMAT statements, in turn, must occur before an END statement.

Vertical lines in the diagram indicate how comment lines and statements may be interspersed in the program. For example, PARAMETER statements must be placed after all PROGRAM, FUNCTION, or SUBROUTINE statements, and before all statement function and executable statements. PARAMETER statements can be placed before, after, or between all IMPLICIT and other specification statements. Comment lines may be interspersed anywhere in a program.

Generally if FORTRAN encounters statements that are out of place, it prints warning messages and continues compilation. In some cases, however, out-of-place statements cause the compiler to terminate compilation or generate unexpected results.


PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA[1] Statements
Comment Lines and INCLUDE[3] Statements
FORMAT and Entry[2] Statements
IMPLICIT Statements
PARAMETER Statements
NAMELIST and DATA Statements
Other Specification Statements
Statement Function Definitions
Executable Statements
END Statement

[1] BLOCK DATA subroutines cannot contain any executable statements, statement functions, FORMAT statements, EXTERNAL statements, INTRINSIC statements, or NAMELIST statements (See Section 14.1).

[2] The ENTRY statement is allowed only in functions or subroutines. All executable statements which reference any dummy parameters must physically follow the ENTRY statement unless the references appear in the FUNCTION statement, the SUBROUTINE statement, or in a preceding ENTRY statement.

[3] The placement of an INCLUDE statement is dictated by the types of statements to be included.

MR-S-3822-85

Figure 6-1: Ordering of FORTRAN Statements

NOTE

In FORTRAN-10/20, a DATA statement can precede a PARAMETER statement or another specification statement.

6.4 COMPILATION CONTROL STATEMENTS

You use compilation control statements to identify FORTRAN programs and to specify their termination. Statements of this type do not affect either the operations performed by the object program, or the manner in which the object program is executed. The three compilation control statements are:

1. PROGRAM statement
2. INCLUDE statement
3. END statement

The PROGRAM statement and the INCLUDE statement are described in the following sections. The END statement is described in Section 9.8.

6.4.1 PROGRAM Statement

This statement allows you to give the main program a name other than the compiler-assumed name "MAIN." The general form of a PROGRAM statement is:

```
PROGRAM name
```

where:

name — is a symbolic name that begins with an alphabetic character and contains a maximum of six characters. (See Section 4.1 for a description of symbolic names.)

The PROGRAM statement must be the first statement in a program unit. (see Section 6.3 for a discussion of the ordering of FORTRAN statements.)

6.4.2 INCLUDE Statement

This statement allows you to include code segments or external declarations in a program unit without having them in the same file as the primary program unit. When an INCLUDE statement is encountered during compilation, the compiler replaces the INCLUDE statement with the contents of the specified file. The general form of the INCLUDE statement is:

```
INCLUDE 'filespec [/switch]'
```

where:

filespec — is a TOPS-10 or TOPS-20 file specification (refer to the _TOPS-10 or TOPS-20 Operating System Commands manual_). The only restriction is that _under TOPS-10_ you cannot specify subdirectory information.

switch — is one of the following optional switches:

/CREF — indicates the included statements are to be used to augment the cross-reference listing (default).

/LIST	indicates that the statement in the specified file is to be listed in the compilation source listing. A number indicating the depth of nesting of include files precedes each statement listed (default).
/NOLIST	indicates that the included statements are not to be printed in the compilation listing.
/NOCREF	indicates that the included statements are not to be used to augment the cross-reference listing.

The following rules govern the use of the INCLUDE statement:

1. The INCLUDEd file can contain any legal FORTRAN statement except a statement that terminates the current program unit, such as the END, PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statements.

   The INCLUDEd file can contain other INCLUDE statements. This is called nesting INCLUDE statements. The number of nested levels is restricted to 12.

2. The proper placement of the INCLUDE statement within a program unit depends upon the types of statements to be INCLUDEd. (See Section 6.3 for information on the ordering of FORTRAN statements.)

3. The file to be INCLUDEd must be on disk.

Note that an asterisk (*) is appended to the line numbers of the INCLUDEd statements on the compilation listing. The level of nesting is indicated following the asterisk.

CHAPTER 7

SPECIFICATION AND DATA STATEMENTS

Specification statements are used to specify type characteristics, storage allocation, and data arrangement. There are ten types of specification statements:

1. DIMENSION

2. Statements that explicitly specify type, including INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER

3. IMPLICIT

4. COMMON

5. EQUIVALENCE

6. EXTERNAL

7. INTRINSIC

8. PARAMETER

9. DATA

10. SAVE

Specification statements are nonexecutable and must conform to the ordering guidelines described in Section 6.3.

7.1 DIMENSION STATEMENT

The DIMENSION statement provides FORTRAN with information needed to identify and allocate the space required for arrays. You may specify any number of subscripted array names as array declarators in a DIMENSION statement. The general form of a DIMENSION statement is:

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

where:

each a(d) is an array declarator. An array declarator provides the name and dimension(s) of an array. An array declarator is written in the following form:

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

where:

a is the symbolic name of the array.

d is the array dimension declarator. The form of a dimension declarator is as follows:

```
[ dl: ] d2
```

where:

dl is an optional integer expression or integer variable specifying the lower dimension bound. The lower dimension bound is the first element in that dimension of the array. If dl is not specified the default is 1.

d2 is an integer expression or integer variable that specifies the upper dimension bound. The upper dimension bound is the greatest element in that dimension of the array. You must have at least one d2 specification in each array declaration.

If both dl and d2 are specified, dl cannot have a value greater than d2. The values dl and d2 can, however, be the same.

An asterisk (*) can also occur as an upper bound, but only as the last dimension. An asterisk marks the declarator as an assumed-size array declarator (see Section 7.1.2).

NOTE

A dimension bound expression must not contain a function or array element reference.

If the array is a dummy argument to a subprogram, then dl and d2 can be integer dummy arguments and d2 can be an asterisk; otherwise, they must be constants.

If the symbolic name of a constant or variable that appears in a dimension bound expression is not of implicit default integer type (see Section 4.2), it must be specified integer by an IMPLICIT statement or a type-statement.

Examples:

```
DIMENSION EDGE (-1:1,4:8), NET (5,10,4), TABLE (567)
DIMENSION TABLE (IAB:J,K,M,10:20)
```

where:

IAB, J, K, and M are of type integer.

7.1.1 Adjustable Dimensions

When used within a subprogram, a declarator for an array that is a dummy argument can use integer dummy arguments as dimension bounds. The following rules govern the use of adjustable dimensions in a subprogram:

1. The array name must be a dummy argument. Each variable that is used as a dimension bound must be either a dummy argument or in COMMON (see Section 7.4).

2. For multiple entry subprograms, if any variables that specify dimension bounds are dummy arguments which do not occur in the formal argument list of the entry point used, the value of the variables as passed for a previous call are used. However, this is only permitted if the subprogram has not changed those dummy arguments. Futhermore, when overlays are used, a SAVE statement that preserves the local variables of the subprogram is needed.

3. If the value of an array dimension variable is altered within the program, the dimensionality of the array is not affected.

4. The size of an array within a subprogram cannot exceed the size of the original array, as defined in the calling program.

Example 1:

```
      SUBROUTINE SBR (ARRAY,M1,M2,M3,M4)
      DIMENSION ARRAY(M1:M2,M3:M4)
      DO 27 L=M3,M4
      DO 27 K=M1,M2
      ARRAY (K,L)=VALUE
27    CONTINUE
      END
```

In the example above, the dimensions of ARRAY will be re-specified each time the subroutine SBR is entered.

Example 2:

```
      SUBROUTINE SB1 (ARR1,M,N)
      DIMENSION ARR1(M,N)
      ARR1(M,N)=VALUE
      ENTRY SB2(ARR1,M)
      ENTRY SB3(ARR1,N)
      ENTRY SB4(ARR1)
          .
          .
          .
      END
```

In the example above, the first call made to the subroutine must be made to SB1 so that all of the dimension bounds have defined values. If a call is made to SB1 with the values M=11 and N=13, a succeeding call to SB2 will use the value N=13, but will give M a new value. If a succeeding call is made to SB4, the last values passed through entries SB1, SB2, or SB3 are used for M and N.

7.1.2 Assumed-size Arrays

An assumed-size array is a dummy array for which the upper bound of the last dimension is specified as an asterisk(*), for example:

```
     SUBROUTINE SUB(A,N)
     DIMENSION A(1:N,1:*)
```

Since storage for array A is allocated in the calling routine, the upper bound of the rightmost dimension of A does not affect the subscript calculations or storage allocation for A.

Therefore, subroutine SUB can be written to handle arguments with any rightmost dimension (the last subscript is never range checked for being too large, even when the /DEBUG:BOUNDS compiler switch is specified). Such a subroutine can declare assumed-size arrays.

The size of an assumed-size array, and the number of elements that can be referenced, are determined as follows:

1. If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual-argument array.

2. If the actual argument corresponding to the dummy argument is a noncharacter array element name, with a subscript value of s in an array of size a, the size of the dummy array is: a+1-s.

3. If the actual argument is a character-array name, character-array element name, or character-array element substring name, and begins at character storage unit b of an array with n character storage units, the size of the dummy array is INT((n+1-b)/y). Where y is the length of an element of the dummy array.

Because the actual size of an assumed-size array is not known, an assumed-size array name cannot be used as:

1. An array name in the list of an I/O statement

2. A unit identifier for an internal file in an I/O statement

3. A format specifier in an I/O statement

4. A NAMELIST statement element

7.2 TYPE SPECIFICATION STATEMENTS

Type specification statements explicitly declare the data type of variables, arrays, or function names. You can give an array name in a type specification statement, either alone (unsubscripted) to declare the type of all its elements, or with dimension bounds, to specify both its type and dimensions.

There are two forms of type specification statements: numeric type specification (see Section 7.2.1) and character type specification (see Section 7.2.2).

7.2.1 Numeric Type Specification Statements

The general form of numeric type specification statements is:

```
type v[,v...]
```

where:

type can be any one of the following declarators:

1. INTEGER
2. REAL
3. DOUBLE PRECISION
4. COMPLEX
5. LOGICAL

v is a variable, array, or function name to be declared the specified type. The names listed must be separated by commas and can appear in only one type statement within a program unit.

Examples:

```
INTEGER A, B, TABLE, FUNC
REAL R, M, ARRAY(5:10,10:20,5)
```

If a name that is the same as an intrinsic FORTRAN function name appears in a conflicting type statement, it is assumed that the name refers to a user-defined routine, variable, or array of the given type. If you place a generic FORTRAN function name in an explicit type statement, it loses its generic properties.

NOTE

The data type size modifier, *n, is accepted by FORTRAN-10/20 to be compatible with the type statements used by other manufacturers. You may append this size modifier to the declarators, causing some to elicit messages warning users of the form of the variable specified by FORTRAN-10/20:

Declarator	Form of Variable Specified
INTEGER*2	Full word integer with warning message
INTEGER*4	Full word integer
LOGICAL*1	Full word logical with warning message
LOGICAL*2	Full word logical with warning message
LOGICAL*4	Full word logical
REAL*4	Full word real
REAL*8	Double-precision real
COMPLEX*8	Complex
COMPLEX*16	Complex with warning message
REAL*16	Double-precision real with warning message
COMPLEX*32	Complex with warning message

In addition, you can append the data type size modifier to individual variables, arrays, or function names. Its effect is to override, for the particular element, the size modifier (explicit or implicit) of the primary type. For example,

```
REAL*4 A, B*8, C*8(10), D
```

A and D are single-precision (one word) real, and B and C are double-precision (two words for each element) real.

7.2.2 Character Type Specification Statements

The form of the character type specification statement is:

```
CHARACTER [*len[,]] v[*len] [,v[*len]]...
```

where:

v is one of the following:

- The name of a symbolic constant, variable, array, or function subprogram
- An array declarator

len is the length of the character data item and is one of the following:

- An unsigned, nonzero integer constant
- An integer constant expression enclosed in parentheses and with a positive value
- An asterisk enclosed in parentheses

If you specify CHARACTER*len, len is the default length specification for that list. If an item in that list does not have a length specification, the item's length is len. But if an item does have a length specification, it overrides the default length specified in CHARACTER*len.

A length specification of asterisk (for example, CHARACTER*(*)) specifies that a dummy argument or function name assumes the length specification of the corresponding actual argument or function reference (see Chapter 13). A length specification of asterisk for the symbolic name of a constant specifies that the symbolic constant assumes the actual length of the constant that it represents.

If you do not specify a length, a length of one is assumed. Note that a length specification of zero is invalid. You can use a character type declaration statement to define arrays by including array declarators (see Section 4.3.2) in the list. If you specify both an array declarator and a length, the array bounds precede the length, the form is:

```
a[(d)][*len]
```

where:

a   is an array name, and a(d) is an array declarator.

Examples of character type specification statements follow:

```
CHARACTER*32 SOCSEC(100)*9, NAMES(100)
```

The above statement specifies an array SOCSEC comprising one hundred 9-character elements, and an array NAMES comprising one hundred 32-character elements.

```
PARAMETER (LENGTH=4)
CHARACTER*(4+LENGTH) LAST, FIRST
```

The above statements specify two 8-character variables, LAST and FIRST. (The PARAMETER statement is described in Section 7.8.)

```
SUBROUTINE S1(BUBBLE)
CHARACTER LETTER(26), BUBBLE*(*)
```

The above statements specify an array LETTER comprising twenty-six 1-character elements and a dummy argument, BUBBLE, which has a length defined by the calling program.

```
CHARACTER*16 QUEST*(5*INT(A))
```

The above statement is invalid. The length specifier for QUEST is not an integer constant expression.

7.3 IMPLICIT STATEMENTS

IMPLICIT statements declare the data type of variables and functions according to the first letter of each symbolic name. The IMPLICIT statement has two forms:

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

IMPLICIT NONE
```

As shown in the statement above, an IMPLICIT statement consists of one or more type declarators separated by commas. Each type declarator has the form:

```
type (a[,a]...)
```

where:

type   can be any one of the following declarators:

1. INTEGER
2. REAL
3. DOUBLE PRECISION
4. COMPLEX
5. LOGICAL
6. CHARACTER[*len]

a is an alphabetic specification in either of the general forms: c or cl-c2, where c, cl, or c2 is an alphabetic character. The latter form specifies a range of letters, from cl through c2, which must occur in alphabetical order.

When you specify type as CHARACTER*len, len specifies the length for character data type. Len is an unsigned, nonzero integer constant or an integer constant expression enclosed in parentheses and with a positive value. If you do not specify a length, a length of one is assumed.

Each letter in a type declarator list specifies that each symbolic name (not declared in an explicit type specification statement) starting with that letter is assigned the data type named in the declarator. For example, the statement:

```
     IMPLICIT REAL (R,M,N,O)
```

declares that all names that begin with the letters R, M, N, or O are type REAL names, unless declared otherwise in an explicit type statement.

NOTE

> Type declarations given in an explicit type specification override those also given in an IMPLICIT statement. IMPLICIT declarations do not affect intrinsic functions. The length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement (see Chapter 13).

You may specify a range of letters within the alphabet by writing the first and last letters of the desired range separated by a dash, for example, A-E for A,B,C,D,E.

Thus, the statement:

```
     IMPLICIT INTEGER (I,L-P)
```

declares that all symbolic names that begin with the letters I,L,M,N,O, and P are of type INTEGER.

You may use more than one IMPLICIT statement, but they must appear before any other declaration statement in the program unit. (See Section 6.3 for a discussion on ordering FORTRAN 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.

You can use an IMPLICIT NONE statement to provide warning messages for variables not explicitly declared, including variables implicitly declared by other IMPLICIT statements. If you specify IMPLICIT NONE, no other IMPLICIT statement should be included in the program unit.

7.4 COMMON STATEMENT

The COMMON statement enables you to establish storage that may be shared by two or more programs and/or subprograms, and to name the variables and arrays that are to occupy the common storage. The use of common storage conserves storage and provides a means to reference the same data in different subprograms without passing arguments.

COMMON statements have the following form:

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

where:

cb — is an optional common block name. (See Section 4.1 for the rules for symbolic names.)

nlist — is a list of variable names, array names, and array declarators that are to occupy the common area. The items specified for a common area (block) are ordered within storage as they are listed in the COMMON statement.

The items in nlist must be either all numeric data type or all character data type. A common block cannot contain both numeric and character data.

A symbolic name can be used to identify each block. However, you can omit the symbolic name for one block in a program unit. This unlabeled block is called the blank common block.

The elements of a named common block can be assigned initial values by DATA statements appearing in the BLOCK DATA subprograms (see Chapter 14). In standard-conforming programs, the elements of the blank common block may not be assigned initial values. However, FORTRAN-10/20 allows any common block elements to be defined in a DATA statement in any program unit.

A given common block name may appear more than once in the same COMMON statement, and in more than one COMMON statement within the same program or subprogram.

When extended addressing is in effect, COMMON blocks reside in the large data area by default. However, the /EXTEND:COMMON or /EXTEND:NOCOMMON switches can be used to explicitly allocate COMMON blocks in the large data area or small data area (see Section 16.5).

During compilation of a source program, FORTRAN strings together all items listed for each common block in the order in which they appear in the source program. For example:

```
     COMMON X,Y,Z/ST1/A,B

     COMMON/ST1/TST(3,4)/ST2/TAB(2,2)

     COMMON/ST2/C,D,E//P,Q

     COMMON W
```

has the same effect as the single statement:

```
     COMMON X,Y,Z,P,Q,W/ST1/A,B,TST(3,4)/ST2/TAB(2,2),C,D,E
```

All elements specified for a common block are placed into one area. Common block names must be unique with respect to all subroutine, function, and entry point names.

NOTE

> If you use overlays, you can use the SAVE statement to retain the value of variables in a named common across overlays (see Section 7.10). (Blank common is always saved.)

For example:

```
         Main Program                  Subprogram

     COMMON DELTA,LENGTH             SUBROUTINE CALC
     COMMON /COM1/KILOS,PRICE        COMMON/COM1/N,A
       .                             COMMON Z,KOUNT
       .                               .
       .                               .
     CALL CALC                         .
       .
       .
       .
```

The COMMON statements in the main program put DELTA and LENGTH into the blank common block, and put KILOS and PRICE into a common block named COM1.

The COMMON statements in the subroutine make Z correspond to DELTA in the main program, KOUNT correspond to LENGTH, N correspond to KILOS, and A correspond to PRICE.

To prevent conflict with other COMMON blocks, LINK requires that the largest definition for each common block be loaded first.

7.4.1 Dimensioning Arrays in COMMON Statements

Array names with dimension bounds can be given in COMMON statements. However, variables cannot be used as dimension bounds in a declarator appearing in a COMMON statement; adjustable dimensioning is not permitted in COMMON.

Each array name given in a COMMON statement must be dimensioned either by the COMMON statement or by another dimensioning statement within the program or subprogram that contains the COMMON statement, but not both.

For examples,

```
     COMMON /A/B(100), C(10,10)
     COMMON X(5,15),Y(5)
```

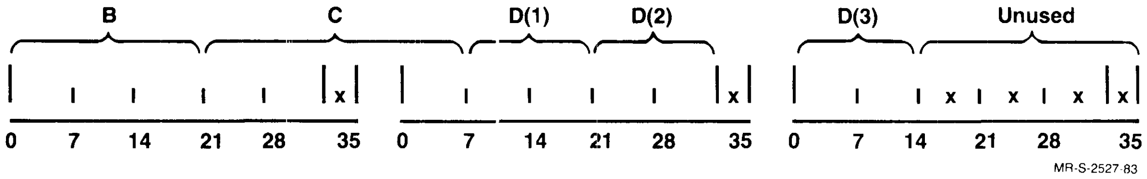
7.4.2 Character Data in COMMON

Each character variable in a COMMON block is allocated to start at the first available character position.

For example,

```
     CHARACTER B*3,C*3,D(3)*2
     COMMON B,C,D
```

The COMMON block will be allocated in the following way:



where x means the bits are not used.

7.5 EQUIVALENCE STATEMENT

The EQUIVALENCE statement associates two or more variables with the same storage location.

The format of the EQUIVALENCE statement is:

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

where:

nlist is a list of variable names, array elements, array names, and character substring references separated by commas and enclosed in parentheses. You must specify two or more of these items in each list.

In an EQUIVALENCE statement, each expression in a subscript or a substring reference must be an integer constant expression.

The EQUIVALENCE statement allocates all of the items in one parenthesized list beginning at the same storage location. For example, the statements:

```
EQUIVALENCE (A,B,C)
EQUIVALENCE (LOC,SHARE(3))
```

specify that the variables A, B, and C are to share the same storage location, and that the variable LOC and the array element SHARE(3) are to share the same location.

The relationship of equivalence is transitive. For example, the following statements have the same effect:

```
EQUIVALENCE (A,B),(B,C)
EQUIVALENCE (A,B,C)
```

The following EQUIVALENCE statement makes the first character of the character variables KEY and STAR share the same storage location. The character variable STAR is equivalent to the substring KEY (1:10):

```
CHARACTER KEY*16, STAR*10
EQUIVALENCE (KEY,STAR)
```

You can equivalence variables of different numeric data types. Character variables must not be equivalenced to numeric variables. For example, you can equivalence a real variable equivalent to a complex variable. In this case, since each complex variable occupies two words of storage, and each single-precision variable occupies one word of storage, if you equivalence a real and a complex variable, the real variable shares storage with the real part of the complex variable. Figure 7-1 depicts the shared storage when a complex variable is equivalenced with a real variable.

Source Program Statements:

```
COMPLEX A
REAL B
EQUIVALENCE (A,B)
```

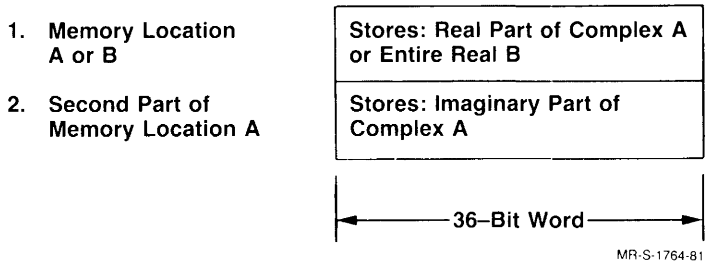


Figure 7-1: Shared Storage using EQUIVALENCE Statement

The EQUIVALENCE statement does not imply (or perform) any type conversions. If you equivalence a real variable and an integer variable, the data within the equivalenced location will be treated as a real or integer number, depending on whether it is referenced by the real or integer variable.

If you equivalence a real variable with a double-precision variable, the data in the high-order word of the double-precision variable will be used by the real variable. For positive D-floating double-precision numbers (see Section 3.4), the high-order word is in the same format as a single-precision number.

For G-floating double-precision numbers (KL model B only - see Section 3.4), the high-order word is not in the format of a single-precision number. Thus, equivalencing a real variable and a G-floating double-precision variable will produce incorrect results.

Equivalencing a negative D-floating number and a real variable may not produce correct results either, for example the number:

577000000000 000000000001 (the negative of 200777777777 777777777777, almost 1.0)

does not have a valid single-precision number in its high-order word.

If you equivalence an array and a variable, the array does not assume any of the properties of the variable, and the variable does not assume any of the properties of the array.

When you use an array element in EQUIVALENCE statements, it must have either as many subscripts as dimensions of the array, or only one subscript. In either case, the subscripts must be integer constants. Note that the single subscript case treats the array as a one-dimensional array of the given type.

The following example shows the effect of equivalencing a 1-dimensional and a 2-dimensional array:

```
     DIMENSION A(3,2),B(6)
     EQUIVALENCE (A(1,1),B(1))
                   or
     EQUIVALENCE (A(1),B(1))
```

In this example, each array element of array A shares the same storage area with an element of array B:

A(1,1)	B(1)
A(2,1)	B(2)
A(3,1)	B(3)
A(1,2)	B(4)
A(2,2)	B(5)
A(3,2)	B(6)

Specifying an array name without subscripts in an EQUIVALENCE statement is the same as specifying the first element of the array.

When you make one character substring equivalent to another character substring, the EQUIVALENCE statement also sets equivalences between the other corresponding characters in the character strings, for example,

```
     CHARACTER NAME*16, ID*9
     EQUIVALENCE (NAME(10:13), ID(2:5))
```

As a result of these statements, the character variables NAME and ID share space as illustrated in Figure 7-2.

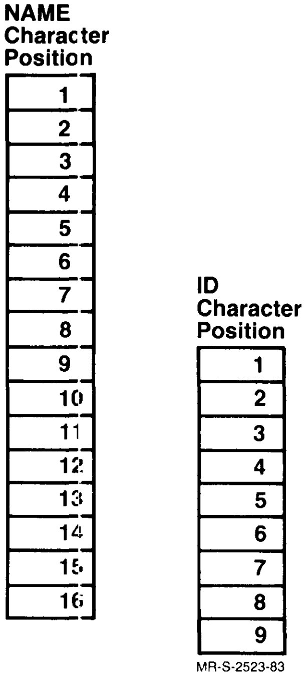


Figure 7-2: Equivalence of Substrings

The following statement also aligns the variables as shown in Figure 7-2:

```
     EQUIVALENCE (NAME(9:9),ID(1:1))
```

If the character substring references are array elements, the EQUIVALENCE statement sets equivalences between the other corresponding characters in the complete arrays.

Character elements of arrays can overlap at any character position, for example:

```
     CHARACTER FIELDS(6)*4, STAR(5)*5
     EQUIVALENCE (FIELDS(1) (2:4), STAR(2) (3:5))
```

As a result of these statements, the character arrays FIELDS and STAR share storage space as shown in Figure 7-3.

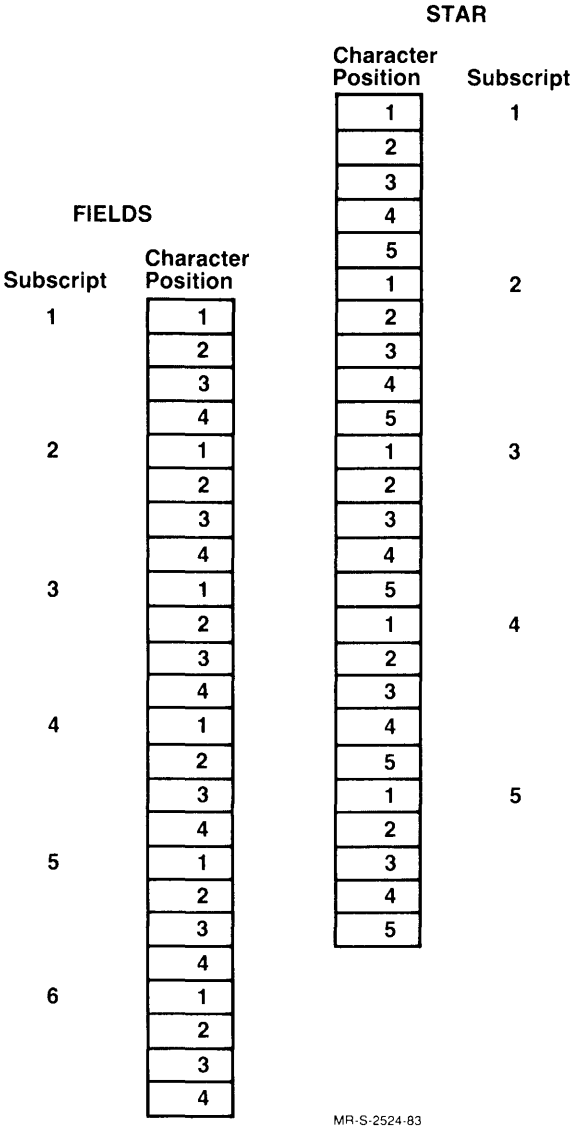


Figure 7-3: Equivalence of Character Arrays

General EQUIVALENCE Restrictions:

1. You cannot cause two different elements of an array to become equivalenced to each other. Thus, the following statement sequence is prohibited because it specifies the same storage location (B) for A(1) and A(2):

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

2. An EQUIVALENCE statement must not specify that two consecutive locations are nonconsecutive. For example, the following statement sequence is prohibited because B(1) takes two storage locations, the second of which would make A(2) nonconsecutive to A(1):

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

3. An EQUIVALENCE statement in a SUBROUTINE or FUNCTION subprogram must not refer to an argument of the subprogram. For example, the following statement sequence is prohibited:

   ```
   SUBROUTINE A(B,C)
   EQUIVALENCE (B,X)
   ```

4. You cannot cause two different substrings of the same character variables or arrays to become equivalenced to each other. For example, the following statement sequence is prohibited because it specifies the same substring B(1:3) for A(1:3) and A(2:4):

   ```
   CHARACTER A(3)*4,B*4
   EQUIVALENCE (A(1)(1:3),B(1:3)), (A(1)(2:4),B(1:3))
   ```

5. You also cannot use the EQUIVALENCE statement to assign memory locations in a way that is inconsistent with the normal linear storage of character variables and arrays. For example, the following statement sequence is prohibited because it would require the character variable B(2:2) to be equivalent to both A(1)(2:2) and A(1)(1:1):

   ```
   CHARACTER A(2)*10,B*10
   EQUIVALENCE (A(1)(1:3),B(1:3)), (A(1)(4:6),B(5:7))
   ```

Restrictions on EQUIVALENCE and COMMON:

1. You cannot use the EQUIVALENCE statement to equivalence two elements in different common blocks. Thus, the following statement sequence is prohibited:

   ```
   COMMON /BLOCK1/A,B,F/BLOCK2/C,D,E
   EQUIVALENCE (A,C)
   ```

2. You cannot set two quantities declared in a COMMON block to be equivalent to one another. Thus, the following statement sequence is prohibited:

   ```
   COMMON A,B,C
   EQUIVALENCE (A,C)
   ```

3. Quantities placed in a common area by means of an EQUIVALENCE statement are permitted to extend the end of the common area forward. For example, the statements:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (A,Y)
```

cause the common block R to extend from Z to A(4) arranged as shown in Figure 7-4.

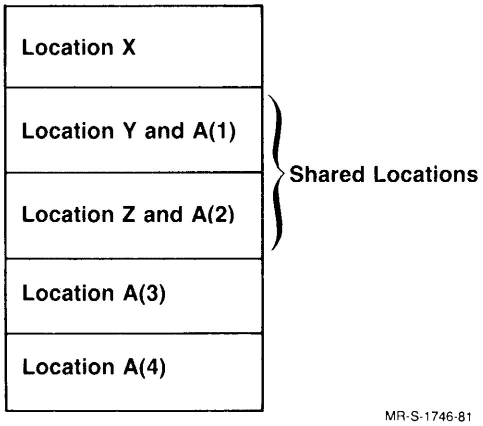


Figure 7-4: Valid Equivalencing

4. You cannot use EQUIVALENCE statements that cause the start of a common block to be extended backwards. For example, the invalid sequence:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

would require A(1) and A(2) to extend the starting location of block R in a backwards direction as illustrated in Figure 7-5.

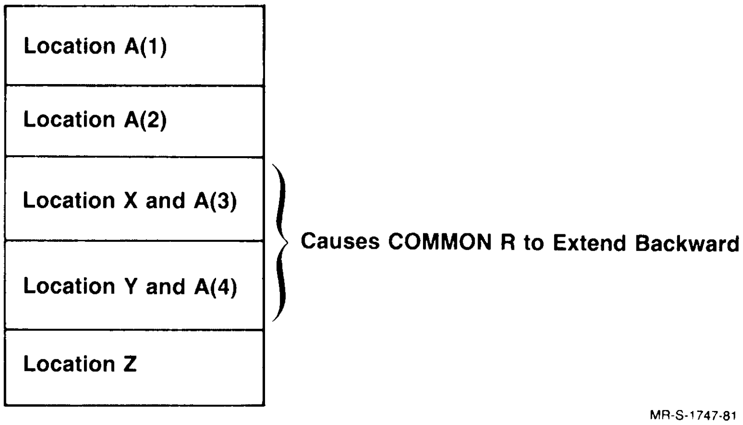


Figure 7-5: Invalid Equivalencing

7.5.1 EQUIVALENCE and Extended Addressing

When extended addressing is in effect, and an EQUIVALENCE statement causes a variable to be in COMMON, that variable resides in the same psect as the rest of the COMMON block.

For variables not in COMMON, if you equivalence a large variable (default 10,000 or more words) with other variables (including scalars), all these variables will reside in the large data psect. For example,

```
     REAL A(20000),X
     EQUIVALENCE (A(1),X)
```

causes variable X to be placed in the large data psect, since it is equivalenced with a large array.

Conversely, if each equivalence variable is small (default less than 10,000 words), they will reside in the small data psect, even if the total size of the equivalence class is over the small variable limit. For example,

```
     REAL A(8000),B(8000)
     EQUIVALENCE (A(8000), B(1))
```

The arrays will reside in the small data psect, because each one is smaller than 10,000 words.

See Sections 15.4.11 and 16.5 for more information on extended addressing.

7.6 EXTERNAL STATEMENT

Any user subprogram name to be used as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. The EXTERNAL statement declares names to be subprogram names to distinguish them from other variable or array names.

The subprograms mentioned in the EXTERNAL statement cannot be FORTRAN intrinsic functions; they can be only user-supplied functions, subroutines, or block data subprograms. (The INTRINSIC statement discussed in Section 7.7 allows intrinsic function names to be used as arguments.) The EXTERNAL statement has the following form:

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

where:

proc — is the symbolic name of a user-supplied subprogram, the name of a dummy argument associated with the name of a subprogram, or a block data subprogram.

The EXTERNAL statement declares each symbolic name included in it to be the name of an external procedure, even if a name is the same as that of an intrinsic function. For example, if SIN is specified in an EXTERNAL statement (EXTERNAL SIN), all subsequent references to SIN are to a user-supplied function name SIN, not to the intrinsic function of the same name.

The name specified in an EXTERNAL statement can be used as an actual argument to a subprogram, which can then use the corresponding dummy argument in a function reference or a CALL statement.

NOTE

> Note that a complete function reference used as an argument, for instance, FUNC(B) in CALL SUBR (A(FUNC(B),C)), represents a value, not a subprogram. A complete function reference is not, therefore, defined in an EXTERNAL statement.

The interpretation of the EXTERNAL statement described above is different from that of earlier versions of FORTRAN-10/20. If the /NOF77 compiler switch is specified (see Sections 16.1.3 and 16.2.3), the subprogram names can be intrinsic functions.

For compatibility with previous versions of FORTRAN-10/20, the names of external subprograms can be preceded by an asterisk (*) or an ampersand (&) within an EXTERNAL statement. For example,

```
     EXTERNAL *SIN, &COS
```

declares SIN and COS to be user subprograms. (If a prefixed name is not an intrinsic function, then the prefix is ignored.)

Note that specifying an intrinsic function in an EXTERNAL statement without a prefix (that is, EXTERNAL SIN) has no effect upon the usage of the function name outside of actual argument lists. If the name has generic properties, they are retained outside the actual argument list. (The name has no generic properties within an argument list.)

The names declared in a program EXTERNAL statement are reserved throughout the compilation of the program, and cannot be used in any declarator statement other than a type statement.

7.7 INTRINSIC STATEMENT

The INTRINSIC statement allows you to use intrinsic function names as arguments to subprograms. See Section 13.1 for further information on intrinsic functions.

The INTRINSIC statement has the form:

```
     INTRINSIC fun[,fun]...
```

where:

fun is the symbolic name of an intrinsic function.

The INTRINSIC statement declares each symbolic name included in it to be the name of an intrinsic procedure. This name can then be used as an actual argument to a subprogram, which can use the corresponding dummy argument in a function reference or a CALL statement.

The appearance of a generic function name in an INTRINSIC statement does not cause that name to lose its generic property.

NOTE

> You cannot use a generic-only name in an INTRINSIC statement. The generic name must be the same as an instrinic function name. For example,
>
> ```
> INTRINSIC LOG
> ```
>
> is illegal because there is no function named 'LOG'. LOG is the generic name that selects between functions such as ALOG, DLOG, or CLOG.

Only one appearance of a symbolic name is permitted in all of the INTRINSIC statements of a program unit. Also, a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

An example of the use of the EXTERNAL and INTRINSIC statements follows:

```
Main Program

    .
    .
    .
EXTERNAL CTN
INTRINSIC SIN, COS
    .
    .
    .
CALL TRIG(ANGLE,SIN,SINE)
    .
    .
    .
CALL TRIG(ANGLE,COS,COSINE)
    .
    .
    .
CALL TRIG(ANGLE,CTN,COTANT)
    .
    .
    .

Subprograms


SUBROUTINE TRIG(X,F,Y)
Y = F(X)
RETURN
END

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

In this example, when TRIG is called with a second argument of SIN or COS, the function reference F(X) references the math library functions SIN and COS; but when TRIG is called with a second argument of CTN, F(X) references the user function CTN.

7.8 PARAMETER STATEMENT

The PARAMETER statement allows you to define constants symbolically during compilation.

The general form of the PARAMETER Statement is:

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

where:

p   is a symbolic name.

c   is a constant expression (except for expressions involving multiplication, division, or exponentiation of complex numbers). (See Chapter 3 for a description of FORTRAN constants.)

The constant acquires the same data type as the symbolic name. If the symbolic name is of type integer, real, double precision, or complex, the corresponding expression (c) must be an arithmetic constant expression (see Section 5.1.2). If the symbolic name is of type character or logical, the corresponding expression (c) must be a character constant expression (see Section 5.2.1) or a logical constant expression (see Section 5.3.1), respectively.

The data type of a symbolic name defined to be a constant is specified by a type-statement or IMPLICIT statement preceding the defining PARAMETER statement. Also, if the length specified for the symbolic name of a character constant is not the default length of one, its length must be specified by a type-statement or IMPLICIT statement preceding the symbolic name of the constant.

NOTE

> The form and the interpretation to the PARAMETER statement described above are different from those of the PARAMETER statement provided in earlier versions of FORTRAN-10/20. The earlier version is described below. This form and interpretation can still be used, however a warning message will be issued. This form of the PARAMETER statement is:
>
> ```
>      PARAMETER p=c[,p=c...]
> ```
>
> The symbolic name acquires the same data type as the constant.

During compilation, the symbolic names are replaced by their associated constants, provided the following rules are observed:

1. Symbolic names may appear only where FORTRAN constants are acceptable.
2. Symbolic name references must appear after the PARAMETER statement definition.
3. Symbolic names must be unique with respect to all other names in the program unit.

4. Symbolic names must not be redefined in subsequent PARAMETER statements.

5. Symbolic names must not be used as part of another constant, such as within a character constant or as the count for a Hollerith constant.

6. Symbolic names must not be used as part of a format specification.

7.9 DATA STATEMENT

DATA statements are used to supply the initial values of variables, arrays, array elements, substrings, and COMMON areas.

The form of the data statement is:

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

where:

nlist identifies a set of items to be initialized.

clist contains the set of values to be assigned the items in nlist.

For example, the statement:

```
DATA IA/5/,IB/10/,IC/15/
```

initializes variable IA to the value 5, variable IB to the value 10, and variable IC to the value 15. The number of storage locations you specify in the list of variables must be equal to the number of storage locations you specify in its associated list of values. If not, a warning message is output.

When the value list specifies more storage locations than the variable list, the excess values are ignored. When the value list specifies fewer storage locations than the variable list, the excess variables are not initialized.

The nlist portion of each nlist/clist/ set can contain the names of one or more variables, array names, array elements, character substring names, or labeled COMMON variables. You may specify an entire array (unsubscripted array name) or a portion of an array in a DATA statement nlist as an implied DO loop construct. (See Section 10.4.9.2 for a description of implied DO loops.)

The form of an implied-DO list in a DATA statement is:

```
(dlist,i=n1,n2[,n3])
```

where:

dlist is a list of array element names, character substring names, or implied-DO lists.

i is the name of an integer variable, called the loop index variable.

n1,n2,n3 are integer expressions that can contain integer constants and loop index variables.

For example, the statement:

```
DATA (NARY(I),I=1,5)/1,2,3,4,5/
```

initializes the first five elements of array NARY as NARY(1)=1, NARY(2)=2, NARY(3)=3, NARY(4)=4, and NARY(5)=5.

When you use an implied DO loop in a DATA statement, the loop index variable must be of type INTEGER, and the Initial, Terminal, and Increment parameters of the loop must be of type INTEGER.

In a DATA statement, references to an array element or substring must be integer expressions in which all terms are either integer constants or indices of embracing implied DO loops. These types of integer expressions can include the exponentiation operator.

The clist portion of each nlist/clist/ set can contain one or more numeric, logical, Hollerith, octal, hexadecimal, or character constants. You may specify literal data as either a Hollerith specification, for example, 5HABCDE, or a string enclosed in single quotes, for example, 'Abcde'. Each ASCII data item is stored left-justified and is padded with blanks if necessary.

When you assign the same value to more than one item in nlist, a repeat specification may be used. The repeat specification has the form:

```
n*d
```

where:

n is an integer that specifies how many times the value d is to be used. For example, a clist specification of /3*20/ specifies that the value 20 is to be assigned to the first three items named in the preceding list. The statement:

```
DATA M,N,L/3*20/
```

assigns the value 20 to the variables M, N, and L.

When the specified data type is not the same as that of the variable to which it is assigned, FORTRAN converts the data item to the type of the variable. The type conversion is performed using the rules given for type conversion in arithmetic assignments. (See Table 8-1.) Octal, logical, Hollerith, hexadecimal, and character constants are not converted.

Sample Statement	Result
DATA PRINT,I,O/'TEST',30,"77/,(TAB(J),J=1,30)/30*5/	The first 30 elements of array TAB are initialized to 5.0.
DATA((A(I,J),I=1,5),J=1,6)/30*1.0/	No conversion required.
DATA((A(I,J),I=5,10),J=6,15)/60*2.0/	No conversion required.

When character variables are initialized, length conversions are made. The conversion is based on the following rules:

1. If the constant contains fewer characters than the length of the element in nlist, the rightmost character positions of the element are initialized with spaces.

2. If the constant contains more characters than the length of the element in nlist, the character constant is truncated to the right.

Character constants and Hollerith constants can also be used to initialize numeric variables. The character string is stored left justified in the numeric variable. When the character string specified is longer than one numeric variable can hold, the string is stored left justified across as many variables as are needed to hold it. If necessary, the last variable used is padded with blanks up to its right boundary.

For character variables, each variable or array element must have exactly one character constant in the data list.

For example, assuming that X, Y, and Z are single-precision, the statement:

```
DATA X,Y,Z/'abcdefghijkl'/
```

causes:

```
X to be initialized to 'abcde'
Y to be initialized to 'fghij'
Z to be initialized to 'klbbb'
```

When a character string is to be stored in double-precision and/or complex variables, and the specified string is only one word long, the second word of the variable is padded with blanks.

For example, assuming that the variable C is complex, the statement:

```
DATA C/'ABCDE','FGHIJ'/
```

causes the first word of C to be initialized to 'ABCDE' and its second word to be initialized to 'bbbbb'. The string 'FGHIJ' is ignored. The first word contains the real part of the complex variable; the second word contains the imaginary part.

In addition, the following two forms of bit data constants are allowed in DATA statements:

```
O'di...dn'

Z'hi...hn'
```

where di are octal digits and hi are hexadecimal digits with A-F representing the decimal equivalent of 10-15. These constants are right-justified. Note that you can also use the double quote (") form of octal constants as described in Section 3.7.

7.10 SAVE STATEMENT

The SAVE statement retains the values stored in a variable, array, or common block after execution of a RETURN or END statement in a subprogram.

The SAVE statement has the following form:

```
     SAVE [a[,a]...]
```

where:

a is a named common block name (preceded and followed by a slash), a variable name, or an array name.

NOTE

> Ordinarily, the values of all variables are retained after execution of a RETURN or END statement. However, when overlays are used, the SAVE statement must be used to ensure retention of values.

An entity specified by a SAVE statement within a program unit does not become undefined upon execution of a RETURN or END statement in that unit. If the entity is in a common block, however, it may be redefined in another program unit that references that common.

Procedure names, the names of variables and arrays in a common block, and dummy argument names cannot be used in a SAVE statement.

A SAVE statement that does not explicitly contain a list is treated as though it contained a list of all allowable items in the program unit that contains the SAVE statement.

If a particular common block name is specified by a SAVE statement in a subprogram of an executable program, it must be specified by a SAVE statement in every subprogram in which that common block appears.

NOTE

> It is not necessary to use the SAVE statement to retain the value of a blank common block; the definition status of blank common is automatically retained after a RETURN or END statement.
>
> Also, when the SAVE statement is used, it is not necessary to specify the LINK switch /OVERLAY:WRITABLE when loading a program.

CHAPTER 8

ASSIGNMENT STATEMENTS

Assignment statements assign values to variables, array elements, or character substrings. There are four kinds of assignment statements:

1. Arithmetic assignment statements (see Section 8.1)

2. Logical assignment statements (see Section 8.2)

3. Statement Label assignment (ASSIGN) statements (see Section 8.3)

4. Character assignment statements (see Section 8.4)

8.1 ARITHMETIC ASSIGNMENT STATEMENT

You use statements of this type to assign numeric values to numeric variables or array elements. Arithmetic assignment statements have the form:

```
     v=e
```

where:

v is the name of the numeric variable or array element that is to receive the specified value.

e is an arithmetic expression.

In assignment statements, the equal symbol (=) does not imply equality as it would in algebraic expressions; it implies replacement. For example, the expression v=e is interpreted as "the contents of the location identified as v are to be replaced by the value of expression e; the previous contents of v are lost."

When the type of the specified variable or array element name differs from that of its assigned value, FORTRAN converts the value to the type of its assigned variable or array element. Table 8-1 describes the type conversion operations performed by FORTRAN for each possible combination of variable and value types.

Table 8-1: Rules for Conversion in Mixed-Mode Assignments

Expression Type (e)	Variable Type (v)					
	REAL	INTEGER	COMPLEX	DOUBLE-PRECISION	LOGICAL	CHARACTER
Real	D	C	R,I	H,L	D	X
Integer	F	D	R,F,I	M	D	X
Complex	R	C,R	D	R,L	R	X
Double-precision	O	N	H,I	D	H	X
Logical	D	D	R,I	H,L	D	X
Octal	D	D	R,I	H,L	D	X
Hollerith	D%	D%	D&	D&	D%	X
Character	X	X	X	X	X	D
Double-Octal*	H	H	D#	D	H	X

Legend

D = Direct replacement
C = Conversion from real to integer with truncation, overflow is possible
F = Conversion from integer to real with rounding
R = Real part only
I = Set imaginary part to 0
H = High-order only
L = Set low-order part to 0
M = Convert with no truncation and no rounding
N = Convert with rounding; truncation can occur and overflow is possible
O = Round to one word of precision, overflow is possible
X = Not allowed

Notes

* Octal numbers with 13 to 24 digits are termed double-octal. Double-octals require two storage locations. They are stored right-justified and are padded with zeros to fill the locations.

& Use the first two words of the Hollerith constant. If the Hollerith constant is only one word long, the second word is padded with blanks.

% Use the first word of the Hollerith constant.

# To convert double-octal numbers to complex, the low-order octal digits are assumed to be the imaginary part, and the high-order digits are assumed to be the real part of the complex value.

8.2 LOGICAL ASSIGNMENT STATEMENTS

Statements of this type are used to assign values to variables and array elements of type logical. Logical assignment statements have the following form:

```
v=e
```

where:

v is the name of a variable or array element

e is a logical expression

For example, assuming that the variables L, F, M, and G are of type logical, the following statements are valid:

Sample Statement	Results
L=.TRUE.	The contents of L are replaced by logical truth.
F=.NOT.G	The contents of F are replaced by the complement of the contents of G.

M=A.GT.T
or
M=A>T
	If A is greater than T, the contents of M are replaced by logical truth; if A is less than or equal to T, the contents of M are replaced by logical false. This can also be read: If A is greater than T, then M is true, otherwise, M is false.

L=((I.GT.H).AND.(J<=K))	The contents of L are replaced by either the true or false resultant of the expression.

8.3 ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT

The ASSIGN statement is used to assign a statement label constant (a 1 to 5 digit statement number) to a variable name. The form of the ASSIGN statement is:

```
ASSIGN s TO i
```

where:

s is a statement number in the current program unit.

i is a variable name.

For example, the statement:

```
     ASSIGN 2000 TO LABEL
```

specifies that the variable LABEL references the statement number 2000.

With the exception of complex, double-precision, or character, you can use any type of variable in an ASSIGN statement.

Use the ASSIGN statement in conjunction with assigned GO TO control statements (see Chapter 9), or as a format identifier in an I/O statement (See Chapter 10). The ASSIGN statement sets up statement label variables that are then referenced in subsequent GO TO control statements, or in format specifiers in I/O statements. The following sequence illustrates the use of the ASSIGN statement:

```
     555 TAX=(A+B+C)*.05
         .
         .
         .
         ASSIGN 555 TO LABEL
         .
         .
         .
         GO TO LABEL
```

8.4 CHARACTER ASSIGNMENT STATEMENT

The character assignment statement assigns the value of the character expression on the right of the equal sign to the character variable, array element, or substring on the left of the equal sign.

The form of the character assignment statement is:

```
     v=e
```

where:

v is a character variable, array element, or substring.

e is a character expression.

If the length of the expression on the right side of the assignment is greater than the length of the variable on the left side, the character expression is truncated on the right.

If the length of the expression on the right side of the assignment is less than the length of the variable on the left side, the character expression is filled on the right with blanks.

FORTRAN-10/20 allows overlap between the character expression and the character variable, array element, or substring. (That is, the character positions defined in the character variable, array element, or substring can be referenced in the character expression.) For example, the following assignments are allowed:

```
     CHARACTER *4 A,B
     DATA A/'abcd'/,B/'efgh'/

     A(1:3) = A(2:4)
     B(2:4) = B(1:3)
```

After the above assignment statements, A is 'bcdd', and B is 'eefg'.

The expression must be of character data type. You cannot assign a numeric value to a character variable, array element, or substring.

Note that assigning a value to a character substring does not affect character positions in the character variable or array element not included in the substring. If a character position outside of the substring has a value previously assigned, it remains unchanged. If the character position is undefined, it remains undefined.

Examples of valid and invalid character assignment statements follow. All variables and arrays in the examples are assumed to be of character data type.

Valid

```
     FILE = 'PROG2'

     REVOL(1) = 'MAR'//'CIA'

     LOCA(3:8) = 'PLANT5'

     TEXT(I,J+1)(2:N-1) = NAME//X
```

Invalid

```
     'ABC' = CHARS        (the left side must be a character variable,
                          array element, or substring reference)

     CHARS = 25           (expression on the right must be of character
                          data type)
```

CHAPTER 9

CONTROL STATEMENTS

FORTRAN object programs normally execute statement by statement in the order in which they were presented to the compiler. The following control statements, however, enable you to alter the normal sequence of statement execution:

1. CALL (Section 13.4.2.2)
2. CONTINUE (Section 9.5)
3. DO (Section 9.3)
4. DO WHILE (Section 9.3.2)
5. ELSE (Section 9.2.4)
6. ELSE IF THEN (Section 9.2.4)
7. END (Section 9.8)
8. END DO (Section 9.4)
9. END IF (Section 9.2.4)
10. GO TO (Section 9.1)
11. IF (Section 9.2)
12. IF THEN (Section 9.2.4)
13. STOP (Section 9.6)
14. PAUSE (Section 9.7)
15. RETURN (Section 13.4.4)

The CALL and RETURN statements are described in Sections 13.4.2.2 and 13.4.4, respectively. The remaining statements are described in this chapter.

9.1 GO TO STATEMENTS

A GO TO statement causes the statement that it identifies to be executed next, regardless of its position within the program.

There are three kinds of GO TO statements: Unconditional (see Section 9.1.1), Computed (see Section 9.1.2), and Assigned (see Section 9.1.3).

9.1.1 Unconditional GO TO Statements

An unconditional GO TO statement transfers program control to the specified statement label.

The form of the unconditional GO TO statement is:

```
GO TO s
```

where:

s   is a statement label of an executable statement.

For example:

```
GO TO 300
```

You can position an unconditional GO TO statement anywhere in the source program, except as the terminating statement of a DO loop.

9.1.2 Computed GO TO Statements

The form of a computed GO TO statement is:

```
GO TO (s [,s]...)[,] e
```

where:

(s[,s]...)   is a list of statement labels.

e   is an integer expression.

You may include any number of statement labels in the list of a computed GO TO statement. However, each statement label must appear within the program unit containing the GO TO statement. The same statement label can appear more than once in the list.

The value of the expression must be an integer value (it will be truncated if necessary) that is greater than 0 and less than or equal to the number of statement labels given in the list. If the value of the expression is not within this range, the next sequential statement is executed.

When a computed GO TO statement is executed, the value of the expression is computed first. The value of the expression specifies the position of the label (within the given list of statement labels) that identifies the statement to be executed next. For example, in the statement sequence:

```
GO TO (20, 10, 5)K
CALL XRANGE(K)
```

the variable K acts as a switch, causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3. The subprogram XRANGE is called if K is less than 1 or greater than 3.

9.1.3 Assigned GO TO Statements

The form of an assigned GO TO statement is:

```
GO TO i [[,] (s [,s]...)]
```

where:

i is a variable name and the optional parenthesized list is a list of statement labels. The statement labels specified must appear within the program unit containing the GO TO statement.

Assigned GO TO statements must be logically preceded by an ASSIGN statement (see Section 8.3) that assigns a statement label value to the variable i. The assigned GO TO statement transfers program control to the label that has been ASSIGNed.

The statement label value assigned must appear within the same program unit as the GO TO statement that uses that value. In statements with a specified list, if i is not assigned one of the statement label values given in the list, the next sequential statement is executed.

Examples:

```
ASSIGN 300 TO STAT1
GO TO STAT1
GO TO STAT1,(177,300,777)
```

9.2 IF STATEMENTS

There are four kinds of IF statements: arithmetic (see Section 9.2.1), logical (see Section 9.2.2), logical two-branch (see Section 9.2.3), and block IF (see Section 9.2.4).

9.2.1 Arithmetic IF Statements

The form of the arithmetic IF statement is:

```
IF (e) s1, s2, s3
```

where:

e is an expression enclosed within parentheses and s1, s2, and s3 are statement labels of three executable statements appearing within the program unit containing the IF statement. The expression e must not be of type complex. The same statement label can appear more than once in the IF statement.

This type of IF statement transfers control of the program to one of the given statements according to the computed value of the given expression. If the value of the expression is:

1. Less than 0, control is transferred to the statement identified by label s1.

2. Equal to 0, control is transferred to the statement identified by label s2.

3. Greater than 0, control is transferred to the statement identified by label s3.

Examples:

`IF(ETA)4, 7, 12`

Transfers control to statement 4 if ETA is negative, to statement 7 if ETA is 0, and to statement 12 if ETA is greater than 0.

`IF(KAPPA-L(10))20, 14, 14`

Transfers control to statement 20 if KAPPA is less than the 10th element of array L and to statement 14 if KAPPA is greater than or equal to the 10th element of array L.

9.2.2 Logical IF Statements

The form of the logical IF statement is:

```
IF (e) st
```

where:

e is any expression. The expression must not be of type complex.

st is an executable statement.

If the value of the expression is true (negative), control is transferred to the executable statement within the IF statement. If the value of the expression is false (nonnegative), control is transferred to the next sequential executable statement. The statement you give in a logical IF statement may be any executable statement except a DO statement, an END statement, or another logical IF statement.

Examples:

`IF(T.OR.S) X=Y+1`

Performs an arithmetic assignment operation if the result of the IF is true.

`IF(Z.GT.X(K)) CALL SWITCH(S,Y)`

Performs a subroutine call if the result of the IF is true.

`IF(K.EQ.INDEX) GO TO 15`

Performs an unconditional transfer if the result of the IF is true.

9.2.3 Logical Two-Branch IF Statements

The format of a logical two-branch IF statement is:

```
IF (e) s1, s2
```

where:

e is any expression, and s1 and s2 are statement labels appearing within the program unit containing the IF statement. The expression must not be of type complex.

Logical two-branch IF statements transfer control to either statement s1 or s2, depending on the computed value of the given expression. If the value of the given logical expression is true (negative), control is transferred to statement s1. If the value of the expression is false (nonnegative), control is transferred to statement s2.

Examples:

IF (LOG1) 10,20	Transfers control to statement 10 if LOG1 is true (negative); otherwise transfers control to statement 20.
IF (A.LT.B.AND.A.LT.C) 31,32	Transfers control to statement 31 if A is less than both B and C; transfers control to statement 32 if A is greater than or equal to either B or C.

9.2.4 Block IF Statements

Block IF statements conditionally execute blocks (or groups) of statements. The four block IF statements are:

- IF THEN
- ELSE IF THEN
- ELSE
- END IF

These statements are used in block IF constructs. The block IF construct has the following form, where the ELSE IF THEN and ELSE statements are optional:

```
IF (e) THEN
    block

ELSE IF (e) THEN
    block
      .
      .
      .

ELSE
    block

END IF
```

where:

e is a logical expression.

block is a sequence of zero or more complete FORTRAN statements. This sequence is called a statement block.

Each block IF statement, except the END IF statement, has an associated statement block. The statement block consists of all the statements following the block IF statement up to (but not including) the next block IF statement in the block IF construct. The statement block is conditionally executed based on the values of logical expressions in the preceding block IF statements. A statement block can be empty.

The IF THEN statement begins a block IF construct. The block following it is executed if the value of the logical expression in the IF THEN statement is true. The first statement of the block cannot directly follow the THEN on the same line. For example, the following is illegal:

```
     IF (T.LT.X) THEN T = X
```

The correct form is:

```
     IF (T.LT.X) THEN
        T = X
```

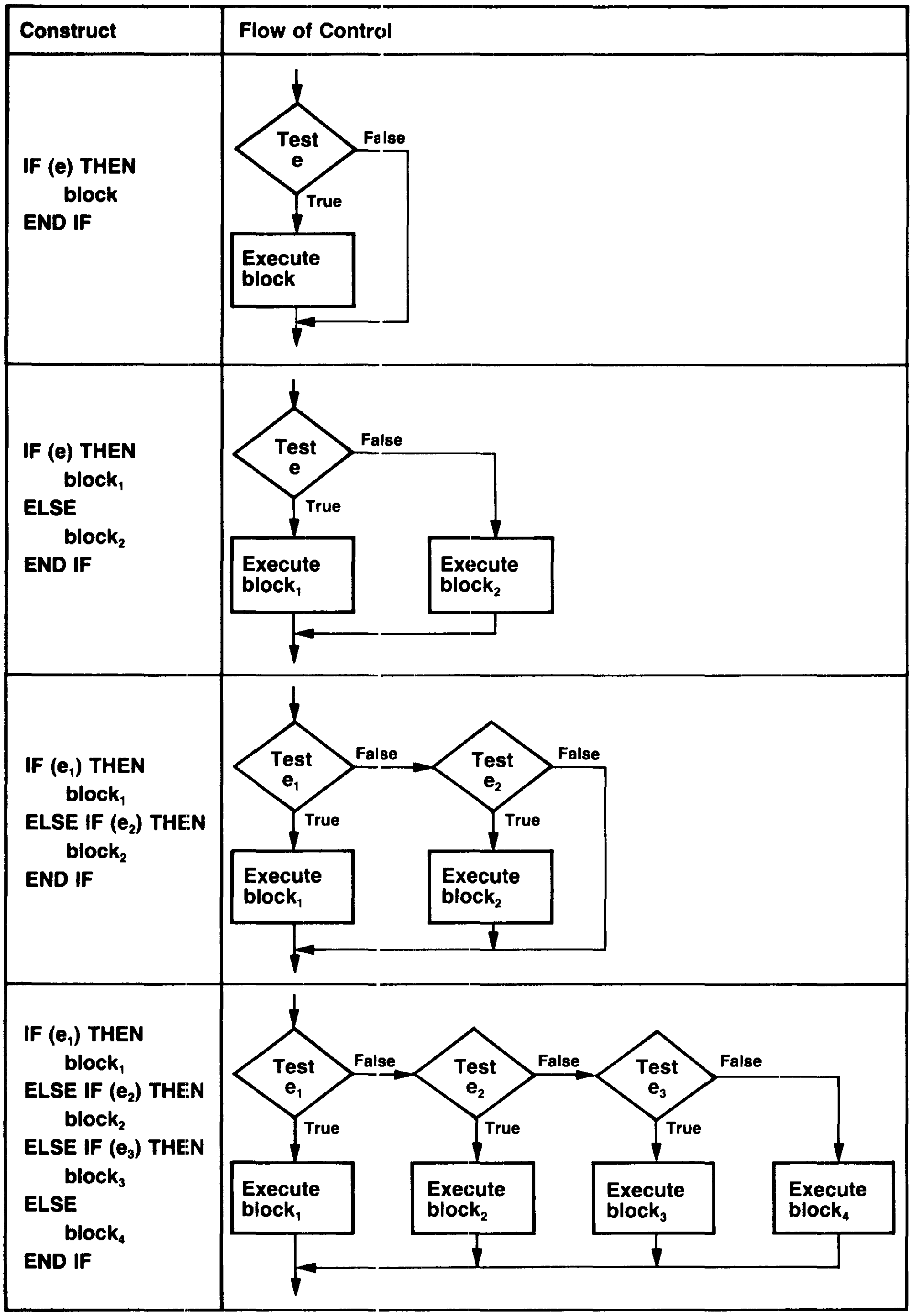
The ELSE statement specifies a statement block to be executed if no preceding statement block in the block IF construct was executed. The ELSE statement is optional.

The ELSE IF THEN statement is similar to the ELSE statement, except it requires an additional condition for execution. The ELSE IF THEN statement specifies a statement block to be executed if both the value of the logical expression in the ELSE IF THEN statement is true, and no preceding statement block in the block IF construct was executed. A block IF construct can contain any number of ELSE IF THEN statements. The ELSE IF THEN statement is optional.

The END IF statement terminates the block IF construct.

Figure 9-1 describes the flow of control for four examples of block IF constructs.

CONTROL STATEMENTS



MR-S-2525-83

Figure 9-1: Examples of Block IF Constructs

After the last statement in a statement block is executed, control passes to the next executable statement following the END IF statement. Consequently, at most one statement block in a block IF construct is executed each time the IF THEN statement is executed.

ELSE IF THEN and ELSE statements can have statement labels, but these labels cannot be referenced. The END IF statement can have a statement label to which control can be transferred, but only from within the block IF construct.

Section 9.2.4.1 describes restrictions on statements in a statement block. Section 9.2.4.2 describes examples of block IF constructs. Section 9.2.4.3 describes nested block IF constructs.

9.2.4.1 Statement Blocks - A statement block can contain any executable FORTRAN statement except an END statement (see Section 9.8). You can transfer control out of a statement block, but you cannot transfer control back into the block. Note that you cannot transfer control from one statement block into another.

DO loops cannot overlap statement blocks. When a statement block contains a DO statement (see Section 9.3), it must also contain the DO loop's terminal statement or END DO statement. Conversely, if a block IF construct appears within the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

9.2.4.2 Block IF Examples - The simplest block IF construct consists of the IF THEN and END IF statements; this construct conditionally executes one statement block.

```
     Form            Example

     IF (e) THEN     IF (LOWER.LE.UPPER) THEN
        block          MIDDLE=(LOWER+UPPER)/2
     END IF          END IF
```

The statement block consists of all the statements between the IF THEN and END IF statements.

The IF THEN statement first evaluates the logical expression (e), (LOWER.LE.UPPER). If the value of e is true, the statement block is executed. If the value of e is false, control transfers to the next executable statement after the END IF statement; the block is not executed.

The following example contains a block IF construct with an ELSE IF THEN statement:

```
     Form                 Example

     IF (el) THEN         IF (ITEM.LT.A(MIDDLE)) THEN
        blockl               UPPER=MIDDLE-1
     ELSE IF (e2) THEN    ELSE IF (ITEM.GT.A(MIDDLE)) THEN
        block2               LOWER=MIDDLE+1
     END IF               END IF
```

Blockl consists of all statements between the IF THEN and the ELSE IF THEN statements; block2 consists of all the statements between the ELSE IF THEN and the END IF statements.

If ITEM is less than A(MIDDLE), blockl is executed.

If ITEM is not less than A(MIDDLE), but ITEM is greater than A(MIDDLE), block2 is executed.

If ITEM is not less than A(MIDDLE) and ITEM is not greater than A(MIDDLE), neither block1 nor block2 is executed; control transfers directly to the next executable statement after the END IF statement.

The following example contains a block IF construct with an ELSE statement:

```
Form                Example

IF (e) THEN         IF (ITEM.GT.A(MIDDLE)) THEN
   block1              LOWER=MIDDLE+1
ELSE                ELSE
   block2              SEARCH=MIDDLE
                       RETURN
END IF              END IF
```

Block1 consists of all the statements between the IF THEN and the ELSE statements; block2 consists of all the statements between the ELSE and the END IF statements.

If ITEM is greater than A(MIDDLE), block1 is executed.

If ITEM is not greater than A(MIDDLE), block2 is executed.

9.2.4.3 Nested Block IF Constructs - A block IF construct can be included in a statement block of another block IF construct. But the nested block IF construct must be completely contained within a statement block; it must not overlap statement blocks.

The following example contains a nested block IF construct.

```
Form                     Example

                            FUNCTION SEARCH(A,N,ITEM)
                            CHARACTER*(*) A(N),ITEM
                            INTEGER SEARCH,N,LOWER,MIDDLE,UPPER

                            LOWER=1
                            UPPER=N

IF (e) THEN                 IF(LOWER.LE.UPPER) THEN
                               MIDDLE=(LOWER+UPPER)/2

         IF (e) THEN     10    IF (ITEM.LT.A(MIDDLE)) THEN
             blocka               UPPER=MIDDLE-1
block1   ELSE IF (e) THEN      ELSE IF (ITEM.GT.A(MIDDLE)) THEN
             blockb               LOWER=MIDDLE+1
         ELSE                  ELSE
             blockc               SEARCH=MIDDLE
                                  RETURN
         END IF                END IF

                               GOTO10

END IF                      END IF
                         20 SEARCH=0
                            RETURN

                            END
```

If LOWER is less than or equal to UPPER, block1 is executed. Block1 contains a nested block IF construct. If ITEM is less than A(MIDDLE), blocka is executed. If ITEM is greater than A(MIDDLE) blockb is executed. If ITEM is equal to A(MIDDLE), blockc is executed.

If LOWER is greater than UPPER, control is transferred to the first executable statement after the last END IF statement. The nested IF construct is not executed.

9.3 DO STATEMENT

The two types of DO statements are:

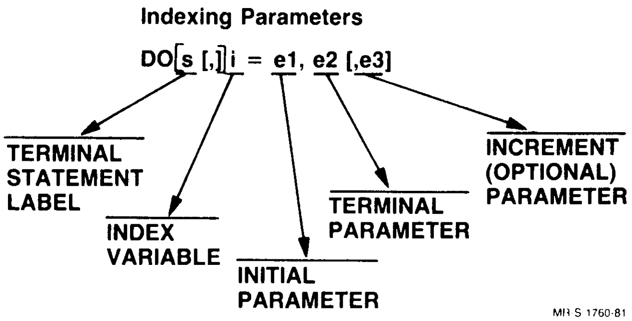
1. Indexed DO (DO statement)
2. Pretested indefinite DO (DO WHILE statement)

The indexed DO statement is described in Section 9.3.1, and the DO WHILE statement is described in Section 9.3.2.

9.3.1 Indexed DO Statement

DO statements simplify the coding of iterative procedures; that is, the statements in the DO statement range are executed repeatedly a specified number of times.

The form of an indexed DO statement is:



where:

s Terminal statement label s identifies the last statement of the DO statement range. The statement must follow the DO statement in the same program unit. If s is omitted, then the loop must be terminated by an END DO statement (see Section 9.4).

The terminal statement can be any executable statement other than one of the following:

- Unconditional or assigned GO TO statement
- Arithmetic IF or logical two-branch IF statement
- Block IF, ELSE IF, ELSE, or END IF statement

- RETURN statement
- STOP statement
- END statement
- DO statement

If the terminal statement is a logical IF, it can contain any executable statement except one of the following:

- DO statement
- Block IF, ELSE IF, ELSE, or END IF statement
- END statement
- Another logical IF statement

i Index variable i is an unsubscripted numeric variable whose value is defined at the start of the DO statement operations. The index variable must not be of type complex.

The index variable is available for use throughout each execution of the range of the DO statement, but altering its value within the DO loop does not change the number of times the DO loop will execute. The DO loop index variable is also available for use in the program when:

a. Control is transferred outside the range of the DO loop by a GO TO, IF, or RETURN statement located within the DO range

b. Control is transferred outside the range of the DO loop by an I/O statement with either or both the options END= or ERR= (see Chapter 10)

c. A subprogram is executed from within the DO statement range having the index variable as an argument or in COMMON

e1 Initial parameter e1 assigns the index variable i its initial value. This parameter can be any expression, but cannot be of type complex.

e2 Terminal parameter e2 provides the value used to determine how many repetitions of the DO statement range are performed. This parameter can be any expression, but cannot be of type complex.

e3 Increment parameter e3 specifies the value to be added to the initial parameter (e1) on completion of each cycle of the DO loop. The increment parameter is optional. If e3 and its preceding comma are omitted, e3 is assumed to be equal to 1. This parameter can be any expression, but cannot be of type complex.

9.3.1.1 **Executing an Indexed DO Statement** - The indexing parameters el, e2 or e3 can be any expressions. Their values are calculated only once, at the start of each DO loop operation, to determine the values for the initial, terminal, and increment parameters. If necessary, the initial, terminal, and increment parameters are converted, before use, to the data type of the index variable.

The number of times that a DO loop will execute, called the iteration count, is specified by the formula:

```
MAX(INT((e2-el+e3)/e3),0)
```

If the iteration count is less than or equal to zero, the body of the loop is not executed. The index variable retains its assigned value (el).

NOTE

> The interpretation of the iteration count described above is different from that of earlier versions of FORTRAN-10/20. If the /NOF77 compiler switch is specified (see Sections 16.1.3 or 16.2.3), and the iteration count is less than or equal to zero, the body of the loop is executed once.

Since the iteration count is computed at the start of a DO loop operation, changing the value of the loop index variable within the loop cannot affect the number of times that the loop is executed.

At the start of a DO loop operation, the index value is set to the value of the initial parameter (el); and the iteration count is established.

9.3.1.2 **DO Iteration Control** - At the end of each DO loop cycle, the following steps are executed:

1. The value of the increment parameter (e3) is added to the index variable.
2. The iteration count is decremented.
3. If the iteration count is greater than zero, control transfers to the first executable statement after the DO statement for another iteration of the loop.
4. If the iteration count is less than or equal to zero, execution of the DO loop terminates.

Exit from a DO loop upon completion of the number of iterations specified by the loop count is referred to as a normal exit. If no other DO loop shares the terminal statement, or if this DO loop statement is outermost, control passes to the first executable statement after the terminal statement of the DO loop.

The final value of the index variable is the value determined by step 1.

NOTE

> The interpretation of the index variable described above is different from that of earlier versions of FORTRAN-10/20. If the /NOF77 compiler switch is specified (see Sections 16.1.3 or 16.2.3), the final value of the index variable of the DO statement is undefined after a normal loop exit.

Exit from a DO loop may also be accomplished by a transfer of control by a statement within the DO loop range to a statement outside the range of the DO statement. This is called an extended range DO loop (see Section 9.3.5).

When execution of a DO loop terminates, and other DO loops share its terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (see Section 9.3.4).

Examples of DO Iteration Control:

```
          DO 100 I = 1,10
     100  J=I
```

After execution of these statements, I=11 and J=10. (If the /NOF77 switch is specified, I is undefined and J=10).

```
          L=0
          DO 200 K = 5,1
     200  L=K
```

After execution of these statements, K=5 and L=0. (If the /NOF77 switch is specified, K is undefined and L=5).

9.3.2 DO WHILE Statement

The DO WHILE statement is similar to the DO statement described in Section 9.3.1. Instead of executing a fixed number of iterations, the DO WHILE statement executes for as long as a logical expression contained in the statement continues to be true.

The form of the DO WHILE statement is:

```
     DO [s[,]] WHILE (e)
```

where:

s   is the label of an executable statement that must physically follow in the same program unit.

e   is a logical expression.

The DO WHILE statement tests the logical expression at the beginning of each execution of the loop, including the first. If the value of the expression is true, the statements in the body of the loop are executed; if the expression is false, control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement (see Section 9.4).

The following example demonstrates the use of the DO WHILE statement:

```
     CHARACTER*132 LINE
     I=1
     LINE(132:) = 'X'
     DO WHILE (LINE(I:I) .EQ. ' ')
          I = I + 1
     END DO
```

9.3.3 The Range of a DO Statement

The range of a DO statement is defined as the series of statements that follows the DO statement, up to and including the specified terminal statement or END DO statement.

If another DO statement appears within the range of a DO statement, the range of that statement must be entirely contained within the range of the first DO statement. More than one DO statement may have the same labeled terminal statement but not unlabeled END DO statement. (See Section 9.3.4, Nested DO Statements.)

If a DO statement appears within an IF block, ELSE IF block, or ELSE block (see Section 9.2.4), the range of the DO statement must be contained entirely within that block.

If a block IF statement appears within the range of a DO statement, the corresponding END IF statement must also appear within the range of the DO statement.

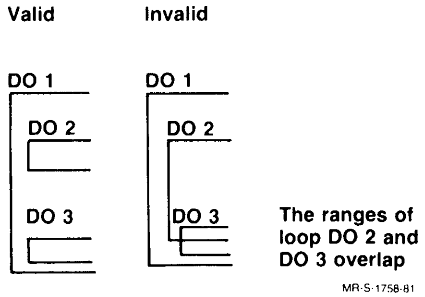
9.3.4 Nested DO Statements

One or more DO statements can be contained within the range of another DO statement. This is called nesting. The following rules govern the nesting of DO statements:

1. The number of nested levels (DO loop within DO loop) is restricted to 79 DO loops.

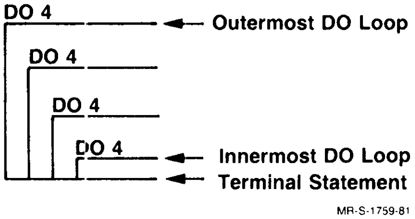
2. The range of each nested DO statement must be entirely within the range of the containing DO statement (such as, they cannot overlap).

   For example:



3. More than one DO loop within a nest of labeled DO loops can end on the same statement. When this occurs, the terminal statement is considered to belong to the innermost DO statement that ends on that statement. Only a statement that occurs within the range of the innermost DO statement can use the statement label of the shared terminal statement for transfer of control.

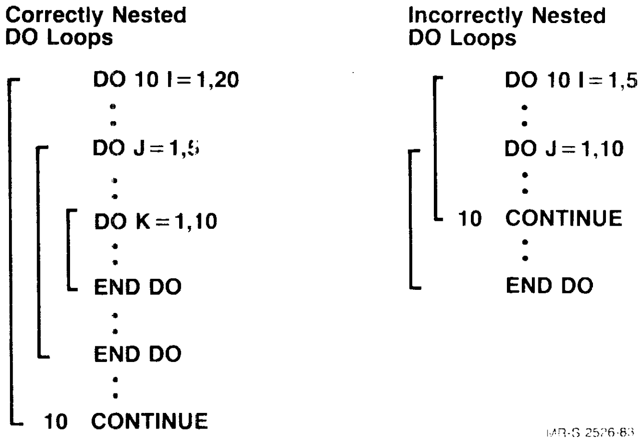
   For example:



   Although all four DO loops share the same terminal statement, the terminal statement "belongs" to the innermost DO loop.

4. Nested loops cannot share an unlabeled END DO statement. Each unlabeled END DO terminates exactly one DO loop.

   For example:



Correctly Nested DO Loops

```
        DO 10 I=1,20
          .
        DO J=1,5
          .
        DO K=1,10
          .
        END DO
          .
        END DO
          .
     10 CONTINUE
```

Incorrectly Nested DO Loops

```
        DO 10 I=1,5
          .
        DO J=1,10
          .
     10 CONTINUE
          .
        END DO
```

9.3.5 Extended Range

By following certain rules, it is possible to transfer out of a DO loop, perform a series of statements elsewhere in the program, and then transfer back into the DO loop. The statements that are executed after a transfer out of a DO loop and before a transfer back into the same DO loop are collectively known as the "extended range." A DO loop that permits transfer in and out of its range is called an extended range DO loop.

NOTE

> This feature makes the flow of a program difficult to follow, does not conform to the FORTRAN-77 standard, and is therefore discouraged.

The following rules govern the use of extended range DO loops:

1. The statement that causes the transfer out of the DO loop must be contained within the most deeply nested DO (innermost loop having the same terminal statement). This loop must also contain the statement to which the extended range returns.

2. A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

3. The extended range of a DO statement must not contain another DO statement.

4. The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.

5. You can call a subprogram within an extended range.

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

```
        DIMENSION TABLE(10,5), VALUE(10)
        LOGICAL LOGARR(10)
        DO 1000 I= 1, 10               ! An extended range DO loop
        IF (LOGARR(I)) GOTO 500        ! Test logical array item
        I=K
        CALL SUBROT(K)                 ! Invoke subroutine using
                                       ! current index value
          DO 200 J= 1, 5               ! Nonextended range loop
          TABLE(I,J) = 0
200       CONTINUE

        GOTO 2000                      ! Extended range invocation
500     VALUE(I) = GETVAL(K)           ! Invoke function GETVAL with
                                       ! current index
1000    CONTINUE                       ! Terminal statement for outer
                                       ! loop
        STOP
2000    TYPE 2100, I                   ! Extended range starts
2100    FORMAT(' I = ',I2)
        LOGARR(I) = .TRUE.
        GOTO 500                       ! Extended range ends and
                                       ! returns
        END
```

9.3.6 Permitted Transfer Operations

The following rules govern the transfer of program control from within a DO statement range or the ranges of nested DO statements:

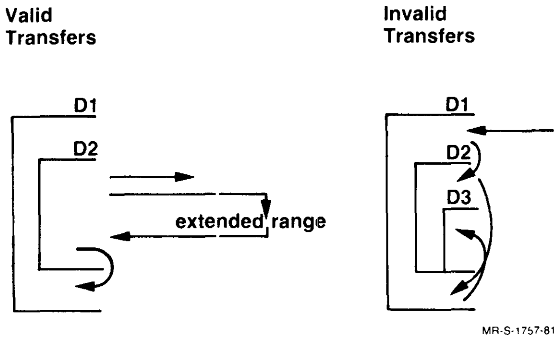
1. A transfer out of the range of any DO loop is permitted at any time. When such a transfer occurs, the value of the controlling DO loop's index variable is defined as the current value.

2. A transfer into the range of a DO statement is permitted if it is made from the extended range of the DO statement.

3. You can call a subprogram from within the range of any:

   a. DO loop
   b. nested DO loop
   c. extended range loop (in which you leave the loop through a GO TO, execute statements in the extended range, and return to the original loop)

The following examples illustrate the transfer operations permitted from within the ranges of nested DO statements:



9.4 END DO STATEMENT

The END DO statement terminates the range of a DO or DO WHILE statement. The END DO statement must be used to terminate a DO block if the DO or DO WHILE statement defining the block does not contain a terminal-statement label. The END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement does contain a terminal-statement label.

The form of the END DO statement is:

```
END DO
```

9.5 CONTINUE STATEMENT

The form of the CONTINUE statement is:

```
CONTINUE
```

Execution of the CONTINUE statement has no effect. It may be used as the terminating statement of a DO loop.

In the following example, the labeled CONTINUE statement provides a legal termination for the range of the DO loop.

```
      DIMENSION STOCK(100)
      DO 20 I=1,100
      STOCK(I)=0
      CALL UPDATE (STOCK(I))
      IF(STOCK (I).EQ. 0) GO TO 30
   20 CONTINUE
      STOP
   30 TYPE 35
   35 FORMAT (' UPDATE ERROR')
      END
```

9.6 STOP STATEMENT

Execution of the STOP statement causes program execution to be terminated. A descriptive message may optionally be included in the STOP statement to be output to your terminal immediately before program execution is terminated.

The form of the STOP statement is:

```
STOP [n]
```

where:

n   is an optional decimal integer constant of up to 6 digits, or a character constant. The constant is printed at the terminal when the STOP statement is executed.

You can have any number of characters in the character constant. You can use continuation lines to accommodate large character strings. The constant is printed without leading zeroes, unless they are specified in the statement.

NOTE

> The word STOP is not printed when the STOP statement is executed unless the word STOP is included in the statement as a character constant.

The following examples show the results of executing STOP statements that contain a 6-digit decimal string and a character constant.

```
        PROGRAM TEST
10      STOP 123456
        END

EXECUTE STOP1.FOR
FORTRAN: STOP1
TEST
LINK:   Loading
[LNKXCT TEST execution]
123456
CPU time 0.1   Elapsed time 0.3
```

```
        PROGRAM TEST
10      STOP 'The program has stopped'
        END

EXECUTE STOP2.FOR
FORTRAN: STOP2
TEST
LINK:   Loading
[LNKXCT TEST execution]
The program has stopped
CPU time 0.1   Elapsed time 0.3
```

9.7 PAUSE STATEMENT

Execution of a PAUSE statement suspends the execution of the object program and gives you the option of continuing execution of the program, exiting from the program, or beginning a TRACE operation.

The form of the PAUSE statement is:

```
PAUSE [n]
```

where:

n   is an optional integer constant of up to 6 digits, or a character constant. The constant is printed at the terminal when the PAUSE statement is executed.

    You can have any number of characters in the character constant. You can use continuation lines to accommodate large character strings. The constant is printed without leading zeros, unless they are specified in the statement.

If execution of the program is resumed after a PAUSE, program control continues as if a CONTINUE had been executed. Execution of the PAUSE statement causes the word PAUSE, the optionally specified constant, and the following prompt to be printed at the terminal:

```
Type G to Continue, X to Exit, T to Trace
```

The responses to this prompt are:

G   continues program execution at the statement immediately following the PAUSE statement.

X   causes program termination.

T   produces a trace back list at the terminal. This list consists of invoked routine names and locations, plus the location and module names of the callers of those routines. Using this information you can track the active path of execution from the main program to the PAUSE trace routine. (See Section 13.4.1.32 for a detailed description of this feature.)

```
PROGRAM PTEST
PAUSE
PAUSE 234
PAUSE 'Character String'
END

EXECUTE PTEST.FOR
FORTRAN: PTEST
PTEST
LINK: Loading
[LNKXCT PTEST execution]
PAUSE
Type G to Continue, X to Exit, T to Trace.
G
PAUSE
234
Type G to Continue, X to Exit, T to Trace.
G
PAUSE
Character String
Type G to Continue, X to Exit, T to Trace.
X

CPU time 0.3   Elapsed time 18.8
```

9.8 END STATEMENT

This statement signals FORTRAN that the physical end of a program unit has been reached. END is an executable statement. The general form of an END statement is:

```
END
```

The following rules govern the use of the END statement:

1. This statement must be the last physical statement of a source program unit (main program or subprogram).

2. When executed in a main program, the END statement has the effect of a STOP statement; in a subprogram, END has the effect of a RETURN statement.

3. An END statement may be labeled, but it must not be continued (that is, it must appear only on an initial line).

CHAPTER 10

DATA TRANSFER STATEMENTS

FORTRAN I/O statements are divided into three categories by function, as follows:

1. Data Transfer Statements - transfer data between memory and files. The "files" can be devices such as TTY: or MTA:. Internal files and ENCODE/DECODE statements are used for memory-to-memory data transfers.

2. File Control Statements - associate and disassociate files and FORTRAN logical unit numbers, and can specify characteristics of such an association.

3. Device Control Statements - position files. For example, using the device control statements you can position magnetic tape to a particular file or record.

This chapter describes data transfer statements. Chapter 11 describes file-control and device-control statements.

Table 10-1 lists the three categories of I/O statements, the statements within each category, and the sections in which each I/O statement is further described.

Table 10–1: FORTRAN I/O Statement Categories

Categories	Statements	Sections
Data Transfer	READ	10.5
	WRITE	10.6
	REREAD	10.7
	ACCEPT	10.8
	TYPE	10.9
	PRINT	10.10
	PUNCH	10.11
	ENCODE	10.12
	DECODE	10.12
	Internal READ	10.12
	Internal WRITE	10.12
File Control	OPEN	11.2
	CLOSE	11.4
	INQUIRE	11.7
Device Control	FIND	11.8.1
	REWIND	11.8.2
	UNLOAD	11.8.3
	BACKSPACE	11.8.4
	ENDFILE	11.8.5
	SKIPRECORD	11.8.6
	SKIPFILE	11.8.7
	BACKFILE	11.8.8

Table 10-2, on the tab-divider, summarizes all the data transfer statement forms.

Table 10–2: Summary of Data Transfer Statement Forms

Data Access	Statement Construct	Section
Sequential Formatted (FORMAT Statement)	READ(UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un, f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ f[,iolist] READ(UNIT=*,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	10.5.1.1
	WRITE(UNIT=un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, f[,ERR=s][,IOSTAT=ios])[iolist] WRITE f[,iolist] WRITE(UNIT=*,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]	10.6.1.1
	REREAD(FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] REREAD f[,iolist]	10.7.1
	ACCEPT(FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT f[,iolist]	10.8.1
	TYPE(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] TYPE f[,iolist]	10.9.1
	PRINT(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] PRINT f[,iolist]	10.10.1
	PUNCH(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] PUNCH f[,iolist]	10.11.1
	ENCODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist] DECODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist]	10.12
Sequential Formatted (List Directed)	READ(UNIT=un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un, *[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ *[,iolist] READ(UNIT=*,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	10.5.1.3
	WRITE(UNIT=un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, *[,ERR=s][,IOSTAT=ios])[iolist] WRITE *[,iolist] WRITE(UNIT=*,FMT=*[,ERR=s][,IOSTAT=ios])[iolist]	10.6.1.3
	REREAD(FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] REREAD *[,iolist]	10.7.2
	ACCEPT(FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT *[,iolist]	10.8.2
	TYPE(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] TYPE *[,iolist]	10.9.2
	PRINT(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] PRINT *[,iolist]	10.10.2
	PUNCH(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] PUNCH *[,iolist]	10.11.2

Table 10–2: Summary of Data Transfer Statement Forms (Cont.)

Data Access	Statement Construct	Section

Sequential Formatted (NAMELIST Statement)	READ(UNIT=un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(UNIT=un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un, name[,END=s][,ERR=s][,IOSTAT=ios])
		10.5.1.4

	WRITE(UNIT=un,FMT=name[,ERR=s][,IOSTAT=ios])
	WRITE(UNIT=un,NML=name[,ERR=s][,IOSTAT=ios])
	WRITE(un,FMT=name[,ERR=s][,IOSTAT=ios])
	WRITE(un,NML=name[,ERR=s][,IOSTAT=ios])
	WRITE(un, name[,ERR=s][,IOSTAT=ios])
		10.6.1.4

Sequential Unformatted	READ(UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	10.5.2.1
	WRITE(UNIT=un[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un[,ERR=s][,IOSTAT=ios])[iolist]	10.6.2.1
Direct Formatted	READ(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un, f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn,FMT=f [,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn, f [,ERR=s][,IOSTAT=ios])[iolist]	10.5.1.2
	WRITE(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn,FMT=f [,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn, f [,ERR=s][,IOSTAT=ios])[iolist]	10.6.1.2
Direct Unformatted	READ(UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn [,ERR=s][,IOSTAT=ios])[iolist]	10.5.2.2
	WRITE(UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn [,ERR=s][,IOSTAT=ios])[iolist]	10.6.2.2

Key:

UNIT=un is a FORTRAN logical unit number or internal file specifier (Section 10.4.3).

UNIT=* is a default unit specification used with the READ Statement to read from CDR:, and with the WRITE Statement to write to LPT: (see Section 10.4.3).

REC=rn is a direct-access record number (Section 10.4.4).

un'rn is an alternate way of specifying Logical Unit Number and record number of a direct-access transfer (Section 10.4.4).

FMT=f is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).

FMT=* is list-directed formatting; iolist is optional (Section 10.4.5.2).

FMT=name is NAMELIST-statement formatting; iolist is prohibited (Section 10.4.5.3).

NML= is the alternative form of the NAMELIST statement format specifier (Section 10.4.5.3).

END=s is an optional end-of-file transfer specifier (Section 10.4.6).

ERR=s is an optional error transfer specifier (Section 10.4.7).

IOSTAT=ios is an optional I/O status specifier (Section 10.4.8).

iolist is a data transfer I/O list (Section 10.4.9).

10.1 DATA TRANSFER OPERATIONS

Data transfer statements are used to transfer data between memory and files or between memory and memory. Data can be transferred sequentially (sequential access) or randomly (direct access). The areas in memory from which data is to be taken during output (write) operations, and into which data is stored during input (read) operations are specified by:

1. A list in the data transfer statement

2. A list defined by a NAMELIST statement

3. FORMAT specifications referenced in the data transfer statement

The appearance and arrangement of transferred data can be specified by:

1. Format specifications located in either a FORMAT statement or an array (FORMAT-statement I/O)

2. The contents of an I/O list (list-directed I/O)

3. An I/O list defined in a NAMELIST statement (NAMELIST I/O)

These three methods are known collectively as formatted I/O.

In contrast to formatted I/O transfers, FORTRAN has several methods for transferring data without regard for the type and arrangement of the data being transferred. These methods are known collectively as unformatted I/O. Unformatted I/O transfers are particularly useful when you want the internal (memory) representation of the data being transferred to be the same as the external (file) representation of the data.

In addition, unformatted data transfers are generally faster than formatted transfers. This is because unformatted data transfers do not convert the data to or from its ASCII representation during the transfer.

The following sections describe the types of access available, the types of data transfers available, and the statements used for I/O transfer operations.

10.2 DATA ACCESS

There are two forms of access available - sequential and direct. These forms are described in the following sections.

10.2.1 Sequential Access

If the data access is sequential, the data records are transferred in a serial fashion to or from the external data file. Each sequential-access input statement transfers the next record(s) from the accessed data file, such that data records are transferred in the same order that they appear in the file.

10.2.2 Direct Access

If the data access is direct, the data records are transferred to or from a file in any desired order, as specified by a record number in the data transfer statement. (Section 10.4.4 describes specifying records in data transfer statements.)

Direct-access transfers, however, can be made only to files residing on disk that have been previously set up (using an OPEN statement) for direct access. Direct-access files must contain identically sized records that are accessed by a record number.

You must use the OPEN statement to establish direct access (see Section 11.2). Execution of the OPEN statement must precede the first data transfer statement for the specified logical unit.

10.3 FORMATTED AND UNFORMATTED DATA TRANSFERS

The term "formatted data transfer" describes an intermediate step that occurs during a data transfer. This intermediate step, which does not occur in an unformatted data transfer, converts the data from its internal (memory) representation to a different external (file) representation. (Formatted data transfers are described in Section 10.3.1.)

An unformatted data transfer refers to the transfer of data with no change to the data during the transfer. In an unformatted data transfer, the internal (memory) representation of the data and the external (file) representation of the data are the same. (Unformatted data transfers are described in Section 10.3.2.)

10.3.1 Formatted Data Transfers

In a formatted data transfer, the internal and external format of the data is controlled during the data transfer in one of three ways:

1. FORMAT-Statement Formatting - The data transfer statement contains a statement number, a numeric array name, a character expression, or an integer, real, or logical variable as a format identifier.

   The statement number references a line that contains a FORMAT statement. The array name references an array that contains a format specification. The value of the character expression is a format specification. The integer, real, or logical variable references a FORMAT statement number that was assigned with an ASSIGN statement.

   In the following example, the data transfer statement contains a statement number of a FORMAT statement. The FORMAT statement, in turn, contains edit descriptors that control the formatting of the data during the transfer:

   ```
          WRITE (22,101)X,J,Z
   101    FORMAT (1X,F10.5,I5,F6.4)
   ```

   See Section 10.4.5.1 for more information on FORMAT-statement formatting.

2. List-Directed Formatting - The data transfer statement contains an asterisk as the format identifier. The asterisk signifies that the transfer is controlled by the data type of the variables in the data transfer statement I/O list.

   In the following example, the data transfer is controlled by the I/O list items X, J, and Z:

   ```
   WRITE (22,*)X,J,Z
   ```

   In this example, unless the data types of X, J, and Z have been set explicitly to a type other than the default data type, the transferred values of X and Z appear in floating-point form, and the transferred value of J appears in integer form.

   See Section 10.4.5.2 for more information on list-directed formatting.

3. NAMELIST-Statement Formatting - The data transfer statement contains a NAMELIST name as the format identifier. This NAMELIST name associates the data transfer statement with a NAMELIST I/O list defined in the NAMELIST statement elsewhere in the same program unit. Elements in the NAMELIST I/O list, in turn, dictate the formatting of the data during the data transfer.

   In the following example, the data transfer is controlled by the NAMELIST.

   ```
   PROGRAM NAMLST
   NAMELIST/VAR/X,Y,Z
   READ(22,VAR)
   WRITE(5,VAR)
   END
   ```

See Section 10.4.5.3 for more information on NAMELIST-statement formatting.

10.3.1.1 Internal Files - Internal files provide the capability to perform formatted data transfers between character variables and the elements of an I/O list. Their use with formatted sequential READ and WRITE statements reduces the need to use the ENCODE and DECODE statements for internal I/O (see Section 10.12).

An internal file consists of a character variable, a character array element, a character array, or a character substring; a record in an internal file consists of any of the above except a character array.

If an internal file is a character variable, array element, or substring, that file comprises a single record whose length is the same as the length of the variable, array element, or substring.

If an internal file is a character array, that file comprises a sequence of records, with each record consisting of a single array element. The sequence of records in an internal file is determined by the order of subscript progression (see Section 4.3.2). Every record of the file has the same length, which is the length of an array element in the array.

The character variable, array element, or substring that is the record of the internal file becomes defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is left-justified and filled with blanks.

A record in an internal file can be read only if the character variable, array element, or substring comprising the record has been defined (that is, a value has been assigned to the record). Prior to data transfer, an internal file is always positioned at the beginning of the first record.

10.3.2 Unformatted Data Transfers

Unformatted data is transferred in two forms on TOPS-20 (BINARY or IMAGE), and three forms on TOPS-10 (BINARY, IMAGE, or DUMP). In an explicit OPEN statement (Section 11.2.1), you can specify one of these forms as an argument to the MODE specifier. (Section 11.3.20 describes the MODE specifier and its arguments.)

On disk devices and CORE-DUMP tapes, numeric data items are transferred directly as 36-bit words. Character data items are transferred as 7-bit bytes. Numeric and character items can be interpersed in the same I/O list. Numeric data items and, for BINARY files, record markers (LSCWs) are always word-aligned (see Section 18.4.2). On INDUSTRY tapes, numeric data items should not be used. Character data items are transferred one character per frame (see Section 11.3.30).

10.3.3 Unformatted Data Transfer to ASCII Devices

Unformatted data transfer can be done to and from ASCII devices (such as line printer, plotter, or terminal). Character data is transferred exactly as it appears in the input/output list, with no formatting or carriage control.

The method for transferring numeric data items depends on the device. For non-terminal devices (such as, line printer or plotter), numeric data is treated as if it were packed (Hollerith) data, left-justifited, five characters per word. For the terminal, the data is treated as if it were right-justified, one character per word.

10.4 DATA TRANSFER STATEMENT FORMS

Table 10-2, on the tab divider, summarizes the forms of all the FORTRAN data transfer statements. Figure 10-1 shows the three major components of data transfer statements.

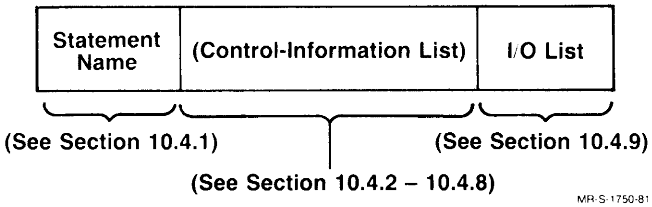


Figure 10-1: Components of Data Transfer Statements

10.4.1 Data Transfer Statement Names

In a data transfer statement, the statement name indicates whether the operation is an input (read) or output (write) operation.

The FORTRAN data transfer statements described in this chapter are:

1. READ (See Section 10.5)
2. WRITE (See Section 10.6)
3. REREAD (See Section 10.7)
4. ACCEPT (See Section 10.8)
5. TYPE (See Section 10.9)
6. PRINT (See Section 10.10)
7. PUNCH (See Section 10.11)
8. ENCODE (See Section 10.12)
9. DECODE (See Section 10.12)
10. Internal READ (See Section 10.12)
11. Internal WRITE (See Section 10.12)

10.4.2 Data Transfer Control-Information List

A control-information list is included in every data transfer statement. Each control-information list (including those having an implicit definition of device) can contain:

1. One unit specifier (see Section 10.4.3)
2. One format specifier (see Section 10.4.5)
3. One record specifier (see Section 10.4.4)
4. One I/O status specifier (see Section 10.4.8)
5. One error specifier (see Section 10.4.7)
6. One end-of-file specifier (see Section 10.4.6)

The following rules govern the placement and inclusion of items in a control-information list:

1. If the keyword UNIT= is omitted from the unit specifier, the unit specifier must be the first item in the control-information list.

2. If the control-information list contains a format specifier (FMT= or NML=), the statement is a formatted data transfer statement. Otherwise, it is an unformatted data transfer statement. The NML= keyword is used for NAMELIST formatting only, although you can also use the FMT= keyword for NAMELIST formatting.

   If the keywords FMT= or NML= is omitted from the format specifier, the format specifier must be the second item in the control-information list, and the first item must be the unit specifier without the keyword UNIT=.

3. If the control-information list contains a record specifier (REC=), the statement is a direct-access data transfer statement. Otherwise, it is a sequential-access data transfer statement.

   If the keyword REC= is omitted from the record specifier, the unit specifier (without the keyword UNIT=) must appear first in the control-information list, followed by a single quote ('), and then the record specifier.

4. A control-information list cannot contain both a record specifier and an end-of-file specifier.

5. If the format specifier is an asterisk or a NAMELIST name, a record specifier must not be included in the control-information list.

6. A control-information list in an internal file or ENCODE or DECODE statement must contain a format specifier other than an asterisk or NAMELIST name, and must not contain a record specifier.

10.4.3 Unit References in Data Transfer Statements

The unit specifier is used to refer to a file or device. The form of a unit specifier is:

```
UNIT = un
```

where:

un is a logical unit identifier or an internal file identifier.

A logical unit identifier (see Section 10.4.3.1) is used to refer to an external file. An internal file identifier (see Section 10.4.3.2) is used to refer to an internal file.

The keyword UNIT= is optional if the unit specifier is the first item in the control-information list.

10.4.3.1 FORTRAN Logical Unit Identifier - The FORTRAN logical unit identifier is associated with the file to or from which data is being transferred. This identifier is an integer expression whose value is in the range of 0 to 99, or an asterisk.

For example, the following WRITE statement contains the reference to logical unit number 22 as the first item in the control-information list:

```
     WRITE (22,101)
```

Table 10-3 lists the default logical unit number assignments. Note that logical unit number 22 identifies the file as DSK:FOR22.DAT. Thus, the sample WRITE statement references a disk. The unit identifier asterisk corresponds to the card reader for the READ statement, and to the line printer for the WRITE statement.

The compiler automatically assigns default logical unit numbers for the REREAD, READ, ACCEPT, PRINT, PUNCH, TYPE, and WRITE statements. Default unit numbers are negative integers that cannot be accessed. For example:

1. OPEN(UNIT=n) or READ/WRITE(UNIT=n) where n is a negative integer is illegal.

2. Assigning a negative decimal number to a device at command level is illegal.

   You can, however, from monitor command level, assign a default device to another device. For example, using the TOPS-20 DEFINE command (or TOPS-10 ASSIGN command), you can assign LPT: (line printer) to DSK: (disk). If you do this, then any I/O statements that reference the line printer actually reference the disk.

You can optionally make the logical device assignments at runtime, or you can use the default assignments contained by the FORTRAN Object Time System (FOROTS). Table 10-3 lists the default logical device assignments. You should specify the device explicitly in an OPEN statement (see Section 11.2) if you wish to override the default assignment.

Table 10-3: FORTRAN Logical Device Assignments

Default Devices (inaccessible to the user)

Device	Default Filename	Logical Unit Number	Use
PLT	FORPLT.DAT	-7	For use by FORPLT
Device last read	File last read	-6	REREAD statement
CDR	FORCDR.DAT	-5	READ statement
TTY	FORTTY.DAT	-4	ACCEPT statement
LPT	FORLPT.DAT	-3	PRINT statement
PTP	FORPTP.DAT	-2	PUNCH statement
TTY	FORTTY.DAT	-1	TYPE statement

Standard Devices*

Device	Default Filename	Logical Unit Number	Use
DSK	FOR00.DAT	00	Disk
DSK	FOR01.DAT	01	Disk
CDR	.	02	Card Reader
LPT	.	03	Line Printer
CTY	.	04	Console Teletype
TTY	.	05	User's Teletype
PTR	.	06	Paper Tape Reader
PTP	.	07	Paper Tape Punch
DIS	.	08	Display
DTA1	.	09	DECtape
DTA2	.	10	.
DTA3	.	11	.
DTA4	.	12	.
DTA5	.	13	.
DTA6	.	14	.
DTA7	.	15	.
MTA0	.	16	Magnetic Tape
MTA1	.	17	.
MTA2	.	18	.
FORTR	.	19	Assignable Device
DSK	.	20	Disk
DSK	.	21	.
DSK	.	22	.
DSK	.	23	.
DSK	.	24	.
DEV1	FOR25.DAT	25	Assignable Devices
DEV2	.	26	.
DEV3	.	27	.
DEV4	.	28	.
DEV5	.	29	.
DSK	.	30	Disk
DSK	.	31	.
.	.	.	.
			.
DSK	FOR99.DAT	99	Disk

*The device table can be altered when FOROTS is installed or by the system administrator. The supplied options are either values in the default table pictured above, or all positive logical unit numbers default to disk. Check to see which device table is being used at your installation.

10.4.3.2 Internal File Identifier - The internal file identifier specifies the internal file to be used. This identifier is the name of a character variable, character array, character array element, or character substring.

Example:

```
     CHARACTER*132 LINE
     WRITE(UNIT=LINE,FMT='(F)')3.14159
```

10.4.4 Record Number References In Data Transfer Statements

All direct-access data transfer statements must contain a record specifier, which is used in the transfer to identify the number of the record to be accessed.

The form of the record specifier in the control-information list is:

```
     REC=rn
```

where:

rn — is a positive integer expression that indicates the record number.

When you use the REC=rn form to specify the record number, you can place the record specifier anywhere in the control-information list.

An alternative way for including the record specifier is:

```
     un'rn
```

where:

un — is a positive integer constant, variable, or array element that represents the logical unit number of the device to or from which the data transfer is being made. When you use this form for specifying the logical unit number, you cannot use the UNIT= keyword.

' — is an apostrophe delimiting the logical unit number from the record number.

rn — is a positive integer constant, variable, or array element that represents the record number.

When you use the alternative form for specifying the record number, you cannot use the keyword REC=.

10.4.5 Format References in Data Transfer Statements

All formatted data transfer statements must contain a format specifier in the control-information list. The general form of the format specifier is:

```
     FMT=f

     or

     NML= a NAMELIST name
```

where:

FMT= is the keyword used in the keyword form of the format specifier. Using the keyword form of the format specifier makes it positionally independent in the control-information list.

NML= is the keyword that can be used instead of FMT= for NAMELIST formatting. Either FMT= or NML= can be used for NAMELIST formatting.

f is a format identifier. Depending on the type of formatting chosen, f can be one of the following:

1. A statement number
2. A numeric array name
3. A character expression
4. An integer, real, or logical variable
5. An asterisk
6. A NAMELIST name

If you do not use the keyword form of the format specifier, you must place the format specifier as the second item of the control-information list (immediately following the logical unit number specifier) (see Section 10.4.2).

Sections 10.4.5.1 through 10.4.5.3 describe all forms of the format specifier.

10.4.5.1 FORMAT-Statement Formatting - The FORMAT-statement format specifier has the following form:

```
FMT=f
```

where:

FMT= is the optional keyword in the format specifier.

f is one of the following:

1. The statement number of a FORMAT statement appearing in the same program unit as the data transfer statement
2. The name of a numeric array
3. A character expression
4. An integer, real, or logical variable that has been assigned a FORMAT statement number with an ASSIGN statement (see Section 8.3)

(See Section 12.1 for more information on FORMAT-statement formatting.)

The following examples show all forms of the FORMAT-statement format specifier. In the first example, the format specifier (FMT=101) references the FORMAT statement 101 in the same program unit.

```
     PROGRAM TEST
     I=67
     P=90.8
     WRITE (UNIT=22,FMT=101) I,P
101  FORMAT (1X,'FIRST VALUE IS: ',I,' SECOND VALUE IS: ',F)
     END
```

In the second example, the same format list used in the first example is stored in an 10-element array. Note that the word "FORMAT" is not included in the array.

```
PROGRAM TESTB
DIMENSION MYARAY(10)

MYARAY(1)='(1X,'''
MYARAY(2)='FIRST'
MYARAY(3)=' VALU'
MYARAY(4)='E IS:'
MYARAY(5)=' '',I,'
MYARAY(6)=''' SEC'
MYARAY(7)='OND V'
MYARAY(8)='ALUE '
MYARAY(9)='IS: '''
MYARAY(10)=',F)'

I=67
P=90.8

WRITE (UNIT=22,FMT=MYARAY)I,P
END
```

In the third example, the same format list used in the first two examples is stored in a character expression.

```
PROGRAM TESTC
INTEGER I
REAL P
CHARACTER WORD1*5,WORD2*6

I=67
P=90.8
WORD1='FIRST'
WORD2='SECOND'
WRITE(UNIT=22,FMT='(1X,'''//WORD1//' VALUE IS: '',I,'' '//
1 WORD2//' VALUE IS: '',F)') I,P
END
```

In the fourth example, the format specifier (FMT=IFORMT) references a variable that has been assigned a statement number.

```
     PROGRAM TESTD
     ASSIGN 101 TO IFORMT
     I=67
     P=90.8
     WRITE (UNIT=22,FMT=IFORMT) I,P
101  FORMAT (1X,'FIRST VALUE IS: ',I, 'SECOND VALUE IS: ',F)
     END
```

For more information on FORMAT-statement formatting, see Section 12.1.

10.4.5.2 List-Directed Formatting - In list-directed formatting, the variables in the I/O list of the data transfer statement dictate the formatting of the data during the transfer.

The form of the list-directed format specifier is:

```
FMT=*
```

where:

FMT= is the optional keyword part of the format specifier. Including this keyword in the format specification makes the specification positionally independent in the control-information list. If you omit the FMT= keyword, the format specifier must be the second specifier (the unit specifier must be first).

* is an asterisk that indicates that the formatting is list-directed.

In the following example, the variables I and P are formatted by list-directed formatting.

```
PROGRAM TESTLD
I=67
P=90.8
WRITE (UNIT=22,FMT=*) I,P
END
```

List-directed formatting is further described in Section 12.5.

10.4.5.3 NAMELIST-Statement Formatting - If the formatting is NAMELIST, the format specifier in the control-information list contains a reference to a NAMELIST name defined in a NAMELIST statement in the same program unit. Since the NAMELIST name definition contains an I/O list, a data transfer statement that contains a NAMELIST name in the format specifier cannot also contain an I/O list.

The form of the NAMELIST format specifier is:

```
FMT=name

or

NML=name
```

where:

FMT= is the optional keyword part of the format specifier. Including the keyword in the format specification makes it positionally independent in the control-information list. If you do not include the keyword part of the format specifier, you must place the format specifier second (after the logical unit number specifier) in the control-information list.

NML= is an alternative keyword that can be used in place of FMT.

name is the NAMELIST name. The NAMELIST name is defined in a NAMELIST statement in the same program unit.

In the following example, the data transfer statement uses a NAMELIST name in its format reference:

```
     PROGRAM TESTNL
     NAMELIST/MYIOLT/I,P
     READ (UNIT=5,NML=MYIOLT)
     WRITE (UNIT=5,FMT=MYIOLT)
     END
```

The execution of this sample program is as follows:

```
     EXECUTE TEST.FOR
     LINK:   Loading
     [LNKXCT TESTNL  execution]
      $MYIOLT I=675,P=34.71$

     $MYIOLT
     I= 675, P= 34.71000
     $END

     CPU time 0.2   Elapsed time 32.0
```

For further information on the NAMELIST statement, see Section 12.7.

10.4.6 Optional End-of-File Transfer of Control (END=)

The optional end-of-file transfer specifier (END=) specifies a statement number to which control passes if this statement attempts to read past the last data record of a file.

If you include an ERR= specifier (Section 10.4.7) and no END= specifier, control passes to the statement indicated in the ERR= specifier whenever an end-of-file condition occurs. Note that an END= specifier on any output statement and on an input statement of a direct-access file is ignored.

If no END= specifier, IOSTAT= specifier, or ERR= specifier is included in the data transfer statement, and an end-of-file condition is encountered, an error message is displayed on the controlling terminal, and program execution is terminated.

The form of the END specifier is:

```
     END=s
```

where:

END= is the keyword part of the END= specifier. The END= portion of the END= specifier is required.

s is the statement number of an executable statement in the current program unit.

In the following example, the end-of-file specifier causes a transfer of control to statement 50 after the data transfer statement encounters an end-of-file on unit 22.

```
          PROGRAM TESTEN
          READ (UNIT=22,FMT=30,END=50) A,B,C
     30   FORMAT (F/F/F)
          GO TO 100
     50   WRITE (UNIT=5,FMT=75)
     75   FORMAT (1X,'END-OF-FILE HAS BEEN ENCOUNTERED')
     100  WRITE (UNIT=5,FMT=105)
     105  FORMAT (1X,'EXECUTION HAS ENDED')
          END
```

The following shows the sample program being executed and the end-of-file branch being taken. In this example, the READ statement reads from the default filename, FOR22.DAT. To demonstrate the end-of-file branch, FOR22.DAT is an empty file. Thus, when the READ statement attempts to read records from FOR22.DAT, an immediate end-of-file condition is detected.

```
     EXECUTE TEST.FOR
     FORTRAN:TESTEN
     TESTEN
     LINK:   Loading
     [LNKXCT TESTEN execution]

     END-OF-FILE HAS BEEN ENCOUNTERED
     EXECUTION HAS ENDED
     CPU time 0.2   Elapsed time 0.5
```

10.4.7 Optional Data Transfer Error Control (ERR=)

The optional error specifier (ERR=) enables you to specify a statement to which control passes if an error occurs during the data transfer. If an error occurs other than for end-of-file, the file is positioned after the record containing the error.

NOTE

> If the program attempts to read from the same unit after an ERR= branch occurs, the record following the record containing the error will be read. To read a record containing the error, the program must execute either a REREAD statement (Section 10.7) or a BACKSPACE (Section 11.8.4) followed by a READ statement.

If no ERR= specifier or IOSTAT= specifier is present and an error occurs during the data transfer, the program is aborted.

The form of the error specifier is:

```
     ERR=s
```

where:

ERR= is the keyword portion of the error specifier.

s is the statement number of an executable statement in the same program unit.

The following example shows the error specifier being used to pass control to the statement at line 85 if an error occurs during the data transfer.

```
          PROGRAM TESTEN
          READ (UNIT=22,FMT=30,END=50,ERR=85) A,B,C
     30   FORMAT (F/F/F)
          GO TO 100
     50   WRITE (UNIT=5,FMT=75)
     75   FORMAT (1X,'END-OF-FILE HAS BEEN ENCOUNTERED')
          GO TO 100
     85   WRITE (UNIT=5,FMT=86)
     86   FORMAT (1X,'THE TRANSFER ENCOUNTERED AN ERROR')
     100  WRITE (5,105)
     105  FORMAT (' EXECUTION HAS ENDED')
          END
```

```
     TYPE FOR22.DAT
     100.
     200.
     AAAA BBBB CCCC DDDD

     EXECUTE TESTEN.FOR
     FORTRAN: TESTEN
     TESTEN
     LINK:   Loading
     [LNKXCT TESTEN execution]

     THE TRANSFER ENCOUNTERED AN ERROR
     EXECUTION HAS ENDED
     CPU time 0.2   Elapsed time 2.8
```

In this example, the error branch is taken when the input routine detects a nonnumeric data item while attempting to read a floating-point number into variable C. If the file FOR22.DAT contains more than three records, the next READ accesses record 4 in the file.

10.4.8 Optional Error Variable For Error Reporting (IOSTAT=)

The optional I/O status specifier enables you to designate an integer variable which receives a value indicating the success or failure of the data transfer.

When the data transfer statement is successfully executed, the variable is assigned a value of zero. If an error occurs during the data transfer, the variable is assigned a positive value indicating which error occured (see Appendix D). In this case, if there is no ERR= specifier, the program proceeds to the statement after the data transfer statement.

If an end-of-file occurs during the data transfer, the variable is set to -1. In this case, if there is no END= or ERR= specifier, the program proceeds to the statement after the data transfer statement.

The form of the error variable specifier is:

```
     IOSTAT=ios
```

where:

ios   is an integer variable that is the I/O status specifier.

The following example shows the I/O status specifier being used to report the number of the error on default unit 5 if the error branch is taken.

```
     PROGRAM TESTEN
10   READ (UNIT=22,FMT=30,END=50,ERR=85,IOSTAT=J)A,B,C
30   FORMAT (F4.1/F4.1/F4.1)
     WRITE (UNIT=5,FMT=40)A,B,C
40   FORMAT (1X,'THE VALUES ARE: ',3F6.1)
     GO TO 100
50   WRITE (UNIT=5,FMT=75)
75   FORMAT (1X,'END-OF-FILE HAS BEEN ENCOUNTERED')
     GO TO 100
85   WRITE (UNIT=5,FMT=86)J
86   FORMAT (1X,'THE TRANSFER ENCOUNTERED AN ERROR; STATUS: ',I5)
     IF(J.GT.0) GO TO 10
100  WRITE (5,105)
105  FORMAT (' EXECUTION HAS ENDED')
     END
```

```
TYPE FOR22.DAT
100.
200.
AAAA BBBB CCCC DDDD
80.
90.
95.
```

```
EXECUTE TESTEN.FOR
FORTRAN: TESTEN
TESTEN
LINK:   Loading
[LNKXCT TESTEN execution]
THE TRANSFER ENCOUNTERED AN ERROR; STATUS:   307
THE VALUES ARE:   80.0  90.0  95.0
EXECUTION HAS ENDED

CPU time 0.2   Elapsed time 1.5
```

In this example, the IOSTAT variable J is set when the first READ detects a nonnumeric data item while trying to input the data for variable C. In this case, the value of IOSTAT represents the processor specific error number (the second value listed in the FOROTS error messages in Section D.1), and indicates that an illegal character has been detected in the data. After the error status has been printed, the second READ successfully executes using records 4, 5, and 6 from the file.

10.4.9 Data Transfer Statement Input/Output Lists

The I/O list in an input or output statement contains the names of variables, arrays, array elements, or character substrings. The I/O list in an output statement can also contain expressions, function references, or constants.

An I/O list has the following form:

```
e[,e]...
```

The variable i and the parameters e1, e2, and e3 have the same forms and the same functions that they have in the DO statement (see Section 9.3). The list immediately preceding the DO loop control variable is the range of the implied DO loop. Elements in that list can reference the index, but they must not alter it. Some examples are:

```
     WRITE (3,200) (A,B,C, I=1,3)
```

The statement in this example functions as though you had written:

```
     WRITE (3,200) A,B,C,A,B,C,A,B,C
```

The following two statements are the same:

```
     WRITE (3,200) (X(I),I=1,3)

     WRITE (3,200) X(1),X(2),X(3)
```

Another example is:

```
     WRITE (6) (I,(J,P(I),Q(I,J),J=1,L),I=1,M)
```

The I/O list in this example consists of an implied DO list containing another implied DO list nested with it. The implied DO lists together write a total of (1+3*L) *M fields, varying values of J for each value of I.

In a series of nested implied DO lists, the parentheses indicate the nesting (see Section 9.3.4). Execution of the innermost list is repeated most often. For example:

```
          WRITE (6,150) ((FORM(K,L), L=1,10), K=1,10,2)
     150  FORMAT (F10.2)
```

Because the inner DO loop is executed 10 times for each iteration of the outer loop, the second subscript, L, advances from 1 through 10 for each increment of the first subscript. This is the reverse of the order of subscript progression. In addition, K is incremented by 2, so only the odd-numbered rows of the array are output.

The entire list of an implied DO list is transmitted before the control variable is incremented, for example:

```
     READ (5,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

In this example, P(1), Q(1,1), Q(1,2),...,Q(1,10) are read before I is incremented to 2.

When processing multidimensional arrays, you can use a combination of fixed subscripts and subscripts that vary according to an implied DO list, for example:

```
     READ (3,5555) (BOX(1,J), J=1,10)
```

This statement assigns input values to BOX(1,1) through BOX(1,10) and then terminates without affecting any other element of the array.

The value of the control variable can also be output directly. For example:

```
     WRITE (6,1111) (I, I=1,20)
```

This statement simply outputs the integers 1 through 20.

10.5 READ STATEMENT

The READ statement transfers data from a file into memory. There are two categories of READ statements: formatted (see Section 10.5.1) and unformatted (see Section 10.5.2).

Table 10-4 summarizes the various forms of the READ statement.

Table 10–4: Summary of READ Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	READ(UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un, f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
Sequential Formatted (List Directed)	READ(UNIT=un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un, *[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

Sequential Formatted
(NAMELIST Statement)
	READ(UNIT=un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(UNIT=un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])
	READ(un, name[,END=s][,ERR=s][,IOSTAT=ios])

Sequential Formatted (Default Unit)	READ f[,iolist] READ *[,iolist] READ(UNIT=*,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(UNIT=*,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
Sequential Unformatted	READ(UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
Direct Formatted	READ(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un, f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn,FMT=f [,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn, f [,ERR=s][,IOSTAT=ios])[iolist]
Direct Unformatted	READ(UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] READ(un'rn [,ERR=s][,IOSTAT=ios])[iolist]

Key:

UNIT=un	is a FORTRAN logical unit number (Section 10.4.3).
UNIT=*	is a default unit specification (Section 10.4.3).
REC=rn	is a direct-access record number (Section 10.4.4).
un'rn	is an alternate way of specifying Logical Unit Number and record number for a direct-access transfer (Section 10.4.4).
FMT=f	is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).
FMT=*	is list-directed formatting; iolist is optional (Section 10.4.5.2).
FMT=name	is NAMELIST-statement formatting; iolist is prohibited (Section 10.4.5.3).
NML=name	is the alternative form of the NAMELIST-statement format specifier (Section 10.4.5.3)
END=s	is an optional end-of-file transfer specifier (Section 10.4.6).
ERR=s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT=ios	is an optional I/O status specifier (Section 10.4.8).
iolist	is a data transfer I/O list (Section 10.4.9).

10.5.1 Formatted READ Transfers

A formatted READ transfer uses a READ statement that specifies that the transferred data is edited during the transfer, such that the external and internal representation of the data are different. The three types of formatted READ statements are: FORMAT-statement, list-directed, and NAMELIST-statement.

There are two types of access to the device from which the READ statement transfers data. They are sequential and direct. If you want to perform a direct-access formatted READ from a device, you must use FORMAT-statement formatting. List-directed and NAMELIST formatting can only be used with sequential-access formatted READ statements.

10.5.1.1 Sequential FORMAT-Statement READ - This section describes the sequential-access (FORMAT-statement) formatted READ statement.

This statement has the following forms:

```
READ (UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

READ (un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

READ (un,f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

If an I/O list is included in these forms, it specifies that data is transferred from logical unit un, formatted according to the specification given by f, and transferred into the elements of the specified I/O list.

If an I/O list is not included, the input record is skipped. (If the FORMAT statement specifies slash editing, more than one record can be skipped. H or apostrophe editing can cause data transfers to occur to the FORMAT statement itself. See Section 12.4.)

The following example contains two READ statements: the first contains an I/O list; the second does not:

```
     READ (22,5) A,Z,J
5    FORMAT (2F10.2,I5)
     READ (22,5)
     END
```

In this example, the first READ statement reads one record from logical unit 22, formats the data according to the FORMAT statement, and assigns the values to the variables A, Z, and J. The second READ statement skips one input record on logical unit 22.

The default unit forms of this READ statement operates in the same way as the first forms, except that data transfers reference the card reader, which is the default logical unit for these forms.

The default unit forms of this statement are:

```
READ f[,iolist]

READ (UNIT=*,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

10.5.1.2 Direct-Access FORMAT-Statement READ - This section describes the direct-access (FORMAT-statement) formatted READ statement.

The forms of this statement are:

```
     READ (UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

     READ (un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

     READ (un,f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]
```

If an I/O list is included in these forms, the data specified by record rn is transferred, according to the format specifications given in f, into the elements of the I/O list. These forms can be used only with disk files that have been opened by an OPEN statement that specifies ACCESS='DIRECT', ACCESS='RANDOM', or ACCESS='RANDIN' (see Section 11.3.1).

If the record specified by rn has not been written, an error results (except for IMAGE mode files).

The following example shows this form of the READ statement.

```
           OPEN(22,RECORDSIZE=25,ACCESS='DIRECT')
           READ (22,5,REC=10)A,Z,J
     5     FORMAT (2F10.2,I5)
           END
```

In this example, the READ statement reads record 10 from logical unit 22, formats the data according to the FORMAT statement, and assigns the values to variables A, Z, and J.

The alternative forms of this READ statement operate in the same way as the first forms. The only difference between the forms is the way in which the unit and record specifications are expressed.

The alternative forms for this statement are:

```
     READ (un'rn,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

     READ (un'rn,f[,ERR=s][,IOSTAT=ios])[iolist]
```

In the alternative forms, the unit and record references do not contain the keywords UNIT= and REC=. Instead the unit number is specified first; a single quote (') is specified next; followed by a record number, a comma, and finally the format reference.

10.5.1.3 Sequential List-Directed READ - This section describes the sequential-access (list-directed) formatted READ statement.

This statement has the following forms:

```
     READ (UNIT=un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

     READ (un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

     READ (un,*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

With these forms, the data is transferred from logical device un and is formatted according to the data types of the elements of the I/O list. If the I/O list is not included, a record is skipped.

The default unit forms of this statement are:

```
READ *[,iolist]

READ (UNIT=*,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

With these forms, the data is transferred from the card reader (the default device), and is formatted according to the data types of the elements in the specified I/O list.

The following example shows this form of the READ statement:

```
CHARACTER*14 C
DOUBLE PRECISION T
COMPLEX D,E
LOGICAL L,M
READ (1,*) I,R,D,E,L,M,J,K,S,T,C,A,B
END
```

The external record to be read contains the following:

```
4 6.3 (3.4,4.2), (3,2 ) , T,F,,3*14.6 ,'ABC,DEF/GHI''JK'/
```

Upon execution of the program unit, the following values are assigned to the I/O list elements:

I	4
R	6.3
D	(3.4,4.2)
E	(3.0,2.0)
L	.TRUE.
M	.FALSE.
K	14
S	14.6
T	14.6D0
C	ABC,DEF/GHI'JK

A, B, and J are unchanged.

10.5.1.4 Sequential NAMELIST-Statement READ - This section describes the sequential-access (NAMELIST-statement) formatted READ statement.

This statement has the following forms:

```
READ (UNIT=un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])

READ (UNIT=un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])

READ (un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])

READ (un,NML=name[,END=s][,ERR=s][,IOSTAT=ios])

READ (un,name[,END=s][,ERR=s][,IOSTAT=ios])
```

With these forms, the data is transferred from the specified unit into the locations specified by the NAMELIST list. The formatting is controlled by the implicit data types of the NAMELIST list items. We suggest that you use the NAMELIST form of the READ statement to transfer data from files created by the NAMELIST form of the WRITE statement (Section 10.6.1.4).

where:

e is a simple input or output item (see Section 10.4.9.1) or an implied DO list (see Section 10.4.9.2).

The I/O statement assigns values to, or transfers values from, the list elements in the order in which they appear (from left to right).

10.4.9.1 Simple List Elements - A simple input list item can be one of the following:

1. A variable name

2. An array element name

3. A character substring name

4. An array name

For example:

```
READ (5,10) J,K(3),CH(1:3)
```

A simple output list item can be one of the above, or it can be one of the following:

1. A constant

2. A function reference

3. An expression

For example:

```
WRITE (5,10) J,K(3),(L+4)/2,CH(1:3)
```

An input list item cannot be an expression. However, it can contain expressions as subscripts or substring bounds.

I/O list items can be of the following types:

1. Integer

2. Real

3. Double-precision

4. Complex

5. Logical

6. Character

7. Octal

8. Double Octal

9. Hollerith

When you use an unsubscripted array name in an I/O list, an input statement reads enough data to fill every element of the array; an output statement writes all the values in the array. Data transfer begins with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly from lower to upper bound. For example, the following defines a two-dimensional array:

```
     DIMENSION ARRAY(3,3)
```

If the name ARRAY with no subscripts appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on through ARRAY(3,3).

In an input statement, variables in the I/O list can be used in array subscripts later in the list, for example:

```
              READ (1,1250) J,K,ARRAY(J,K)
     1250     FORMAT (I1,1X,I1,1X,F6.2)
```

The input record contains the following values:

```
     1,3,721.73
```

When the READ statement is executed, the first input value is assigned to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Then the value 721.73 is assigned to ARRAY(1,3). Variables that are to be used as subscripts in this way must appear before (to the left of) their use as the array subscripts in the I/O list.

10.4.9.2 Implied DO Lists - An implied DO list is an I/O list element that functions as though it were part of an I/O statement within a DO loop. Implied DO lists can be used to:

1. Specify iteration of part of an I/O list

2. Transfer part of an array

3. Transfer array elements in a sequence different from the order of subscript progression

As in explicit DO loops, zero-trip implied DO loops are possible (see Section 9.3).

An implied DO list has the form:

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

where:

dlist	is an I/O list. Dlist can also contain other implied DO lists.
i	is the index control variable that can represent a subscript appearing in a preceding subscript list.
e1,e2,e3	are the indexing parameters that specify, respectively, the initial, terminal, and increment values that control the range of i. If e3 is omitted (with its preceding comma), a value of 1 is assumed.

The following example shows this form of the READ statement:

```
     NAMELIST /DATA/A,Z,J
     READ (22,DATA)
     END
```

In this example, the NAMELIST statement associates the NAMELIST name DATA with a list of three items. The corresponding READ statement reads input data and assigns values to the specified namelist items.

10.5.2 Unformatted READ Transfers

Unformatted READ transfers move data from a specified file to locations in memory. Unlike formatted READ transfers, unformatted transfers do not involve any editing of the data.

The two types of unformatted data transfers enable you to access a specified file either sequentially or directly.

> NOTE
>
> The OPEN statement MODE specifier enables you to specify in which form the unformatted data file exists (see Section 11.3.20). If you execute an unformatted READ statement without having first specified the MODE in an OPEN statement, the data file is assumed to be BINARY. (For additional information on unformatted data file forms, see Section 11.2.)

10.5.2.1 Sequential Unformatted READ - This section describes the sequential-access unformatted READ statement.

This statement has the following forms:

```
     READ (UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

     READ (un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is present, the data is transferred as one logical record from the specified logical unit. This type of read should only be used to read files that have been created by unformatted WRITE statements.

If you omit the I/O list portion of the statement, the statement skips one logical record on input from the specified unit.

The following example shows this type of READ statement used both with and without the I/O list:

```
     READ (22)A,Z,J
     READ (22)
     END
```

In this example, the first READ statement reads one record from logical unit 22 and assigns values to variables A, Z, and J. The second READ statement skip one record from logical unit 22.

10.5.2.2 Direct-Access Unformatted READ - This section describes the direct-access unformatted READ statement.

This statement has the following forms:

```
     READ (UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

     READ (un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]
```

If an I/O list is included in these forms, the data, in the form of one logical record, is transferred from the specified unit into the elements of the I/O list. Only files that have been output by an unformatted WRITE statement should be transferred by this form of the READ statement. In addition, for FORTRAN binary files, if the record specified by rn has not been written, an error results.

These forms can be used only with disk files that have been opened by an OPEN statement that specifies ACCESS='DIRECT', ACCESS='RANDOM', or ACCESS='RANDIN' (see Section 11.3.1).

The alternative form of this READ statement functions the same as the first forms. The only difference between the forms is in the way that the unit and record are specified.

The alternative form of this statement is:

```
     READ (un'rn[,ERR=s][,IOSTAT=ios])[iolist]
```

In this form, the unit and record references do not contain the keywords UNIT= and REC=. Instead the unit number is specified first; a single quote (') is specified next; then the record number is specified last.

The following example demonstrates the use of the unformatted READ statement:

```
     OPEN (22,ACCESS='DIRECT',RECORDSIZE=3)
     READ (22,REC=10)A,Z,J
     READ (22'12)B,X,K
     END
```

In this example, the first READ statement reads record 10 from logical unit 22 and assigns values to the variables A, Z, and J. The second READ statement reads record 12 from logical unit 22 and assigns values to the variables B, X, and K.

WRITE Statement

10.6 WRITE STATEMENT

WRITE statements transfer data from memory to a file. The various forms of the WRITE statement enable it to be used in sequential, append, and direct-access transfer modes for formatted, unformatted, list-directed, and NAMELIST-controlled data transfers.

Table 10-5 summarizes all forms of the WRITE statement.

Table 10-5: Summary of WRITE Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	WRITE(UNIT=un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, f[,ERR=s][,IOSTAT=ios])[iolist]
Sequential Formatted (List Directed)	WRITE(UNIT=un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, *[,ERR=s][,IOSTAT=ios])[iolist]

Sequential Formatted
(NAMELIST Statement)
	WRITE(UNIT=un,FMT=name[,ERR=s][,IOSTAT=ios])
	WRITE(UNIT=un,NML=name[,ERR=s][,IOSTAT=ios])
	WRITE(un,FMT=name[,ERR=s][,IOSTAT=ios])
	WRITE(un,NML=name[,ERR=s][,IOSTAT=ios])
	WRITE(un, name[,ERR=s][,IOSTAT=ios])

Sequential Formatted (Default Unit)	WRITE f[,iolist] WRITE *[,iolist] WRITE(UNIT=*,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(UNIT=*,FMT=*[,ERR=s][,IOSTAT=ios])[iolist]
Sequential Unformatted	WRITE(UNIT=un[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un[,ERR=s][,IOSTAT=ios])[iolist]
Direct Formatted	WRITE(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un, f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn,FMT=f [,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn, f [,ERR=s][,IOSTAT=ios])[iolist]
Direct Unformatted	WRITE(UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un'rn [,ERR=s][,IOSTAT=ios])[iolist]

Key:

UNIT=un	is a FORTRAN logical unit number (Section 10.4.3).
UNIT=*	is a default unit specification (Section 10.4.3).
REC=rn	is a direct-access record number (Section 10.4.4).
un'rn	is an alternate way of specifying Logical Unit Number and record number for a direct-access transfer (Section 10.4.4).
FMT=f	is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).
FMT=*	is list-directed formatting; iolist is optional (Section 10.4.5.2).
FMT=name	is NAMELIST-statement formatting; iolist is prohibited (Section 10.4.5.3).
NML=name	is the alternative form of the NAMELIST-statement format specifier (Section 10.4.5.3)
ERR=s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT=ios	is an optional I/O status specifier (Section 10.4.8).
iolist	is a data transfer I/O list (Section 10.4.9).

10.6.1 Formatted WRITE Transfers

A formatted WRITE transfer uses a WRITE statement that specifies that the transferred data is edited during the transfer, such that the external and internal representations of the data are different. The three types of formatted WRITE statements are: FORMAT-statement, list-directed, and NAMELIST-statement.

There are two types of access to the device to which the WRITE statement transfers data. They are sequential and direct. If you want to perform a direct-access formatted WRITE to a device, you must use FORMAT-statement formatting. List-directed and NAMELIST-statement formatting can only be used for sequential-access formatted WRITE statements.

10.6.1.1 Sequential FORMAT-Statement WRITE - This section describes the sequential-access (FORMAT-statement) formatted WRITE statement.

This statement has the following forms:

```
WRITE (UNIT=un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un,f[,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is included in these forms, the data specified by the elements of the I/O list are output to a file on logical unit un. The output data is formatted in this file according to the FORMAT specifications given in f.

A blank record is written if the I/O list is not specified, and one of the following is true:

1. The FORMAT statement is empty.
2. No slash, H, or apostrophe editing descriptors occur alone.
3. No slash, H, or apostrophe editing descriptors precede the first repeatable edit descriptors.

   See Section 12.4.

The following example contains a sequential formatted WRITE that contains an I/O list, and one that does not:

```
     A=11.4
     Z=13.9
     J=5
     WRITE (22,5)A,Z,J
5    FORMAT (1X,2F10.2,I5)
     WRITE (22,15)
15   FORMAT (' PAGE NO. 1')
     END
```

The following is written to logical unit 22:

```
     11.40     13.90     5
PAGE NO. 1
```

The default unit forms of this statement are:

```
WRITE f[,iolist]

WRITE (UNIT=*,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]
```

If an I/O list is included in these forms, the data, specified by the elements within the I/O list, are transferred to the default device (line printer). The transferred data are formatted according to the FORMAT specification given by f.

The following example shows both forms of this WRITE transfer:

```
          A=11.4
          Z=13.9
          J=5
          WRITE 5,A,Z,J
     5    FORMAT (1X,2F10.2,I5)
          WRITE 15
     15   FORMAT (' PAGE NO. 1')
          END
```

The following is written to the default device (line printer):

```
          11.40     13.90     5
     PAGE NO. 1
```

10.6.1.2 Direct-Access FORMAT-Statement WRITE - The direct-access (FORMAT-statement) formatted WRITE statement is described in this section.

This statement has the following forms:

```
     WRITE(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

     WRITE(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

     WRITE(un,f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]
```

If you include an I/O list in these forms, the data in the I/O list is written starting at record rn to a file on logical unit un. The formatting is controlled by the FORMAT specifications given at f.

Only disk files that have been opened by an OPEN statement that specifies ACCESS='DIRECT' or ACCESS='RANDOM' (see Section 11.3.1) can be accessed by a WRITE statement of this form.

If you omit the I/O list portion of this statement, at least one blank record (specified by REC=rn) is written to logical unit un.

The following example shows a direct-access formatted WRITE statement that contains an I/O list, and one that does not:

```
          A=11.4
          Z=13.9
          J=5
          OPEN(22,RECORDSIZE=25,ACCESS='RANDOM')
          WRITE (22,5,REC=10)A,Z,J
     5    FORMAT (2F10.2,I5)
          WRITE (22,15,REC=11)
     15   FORMAT (' PAGE NO. 1')
          END
```

The following is written to logical unit 22:

```
          11.40    13.90    5
     PAGE NO. 1
```

The alternative forms of this WRITE statement operate the same way as the first forms. The only difference between the forms is in the way that the logical unit and the record number are expressed.

The alternative·forms of this statement are:

```
     WRITE (un'rn,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

     WRITE (un'rn,f[,ERR=s][,IOSTAT=ios])[iolist]
```

In these forms, the unit and record references do not contain the keywords UNIT= and REC=. Instead the unit number is specified first; a single-quote (') is specified next, followed by a record number, a comma, and finally the format reference.

10.6.1.3 **Sequential List-Directed WRITE** - This section describes the sequential-access (list-directed) formatted WRITE statement.

This statement has the following forms:

```
     WRITE (UNIT=un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist]

     WRITE (un,FMT=*[,ERR=s][,IOSTAT=ios])[iolist]

     WRITE (un,*[,ERR=s][,IOSTAT=ios])[iolist]
```

These forms of the WRITE statement specify that the data identified in the I/O list is written to logical unit un. Because the transfer is list-directed (FMT=*), the data is formatted according to the implicit data types of the variables in the I/O list. If the I/O list is not included, a blank record is written.

The default unit forms of this statement are:

```
     WRITE *[,iolist]

     WRITE (UNIT=*,FMT=*[,ERR=s][,IOSTAT=ios])[iolist]
```

The default unit forms function in the same way as the first forms, except that the output is written to the default device (line printer).

The following example shows the list-directed WRITE statement:

```
     DIMENSION A(4)
     DATA A/4*3,4/
     WRITE (1,*) 'ARRAY VALUES FOLLOW'
     WRITE (1,*) A,4
     END
```

The following is written to logical unit 1:

```
     ARRAY VALUES FOLLOW
     4*3,400000, 4
```

10.6.1.4 Sequential **NAMELIST-Statement WRITE** - This section describes the sequential-access (NAMELIST-statement) formatted WRITE statement.

This statement has the following forms:

```
     WRITE (UNIT=un,FMT=name[,ERR=s][,IOSTAT=ios])

     WRITE (UNIT=un,NML=name[,ERR=s][,IOSTAT=ios])

     WRITE (un,FMT=name[,ERR=s][,IOSTAT=ios])

     WRITE (un,NML=name[,ERR=s][,IOSTAT=ios])

     WRITE (un,name[,ERR=s][,IOSTAT=ios])
```

These forms of the WRITE statement transfer data defined in the referenced NAMELIST statement (FMT=name or NML=name) to the file on the logical unit specified by un.

The following example demonstrates the NAMELIST form of the WRITE statement:

```
     CHARACTER*19 NAME(2)
     DATA NAME/2*' '/
     REAL PITCH, ROLL, YAW, POSIT(3)
     LOGICAL DIAGNO
     INTEGER ITERAT
     NAMELIST /PARAM/ NAME, PITCH, ROLL, YAW, POSIT, DIAGNO, ITERAT
     ACCEPT (FMT=PARAM)
     WRITE (UNIT=1,FMT=PARAM)
     END
```

The input contains the following:

```
     b$PARAM NAME(2)(10:)='HEISENBERG',
     bPITCH=5.0, YAW=0.0, ROLL=5.0,
     bDIAGNO=.TRUE.
     bITERAT=10$END
```

The WRITE statement writes the following:

```
$PARAM
NAME= '                   ', '         HEISENBERG', PITCH=
5.000000, ROLL= 5.000000, YAW= 0.0000000E+00, POSIT= 3*0.0000000E+00,
DIAGNO= T, ITERAT= 10
$END
```

10.6.2 Unformatted WRITE Transfers

Unformatted WRITE transfers move data from memory to a file. Unlike formatted WRITE transfers, unformatted WRITE transfers do not involve any editing of the data.

The two types of unformatted data transfers enable you to write to a file either sequentially or directly.

NOTE

The MODE specifier of the OPEN statement enables you to specify the type of unformatted data file you want to create (see Section 11.3.20). If you execute an unformatted WRITE statement without having first specified the MODE in an OPEN statement, the data file is by default BINARY. For additional information on unformatted data file forms, see Section 11.2.

10.6.2.1 Sequential Unformatted WRITE - This section describes the sequential-access unformatted WRITE statement.

This statement has the following forms:

```
WRITE (UNIT=un[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un[,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is present in these forms, the data is written as one logical record to the file contained on the specified logical unit un.

If you omit the I/O list in this statement, the statement writes one blank logical record to the file contained on the specified logical unit un.

The following example shows this form of the WRITE statement with the I/O list and without the I/O list:

```
WRITE (22)A,Z,K
WRITE (22)
END
```

In this example, the first WRITE statement writes a record to the file connected to logical unit 22 containing the values of the variables A, Z, and K. The second WRITE statement writes one blank record to the file connected to logical unit 22.

10.6.2.2 Direct-Access Unformatted WRITE - This section describes the direct-access unformatted WRITE statement.

This statement has the following forms:

```
WRITE (UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]
```

These forms can be used only with disk files which have been opened by an OPEN statement that specifies ACCESS='DIRECT' or ACCESS='RANDOM' (see Section 11.3.1). If an I/O list is included in these forms, the data, in the form of one logical record, is transferred from the memory to record rn of the file on the specified logical unit.

If the I/O list is not specified, the statement outputs one logical blank record.

The following example shows this type of WRITE statement with an I/O list and without an I/O list:

```
     OPEN(22,ACCESS='DIRECT',RECORDSIZE=3)
     WRITE (22,REC=10)A,Z,K
     WRITE (22,REC=12)
```

In this example, the first WRITE statement writes the values of the variables A, Z, and K to record 10 on logical unit 22. The second WRITE statement writes one logical blank record to record 12 on logical unit 22.

The alternative form of this type of WRITE statement operates in the same way as the first forms. The difference between the forms is in the way that the unit and the record are specified.

The alternative form of this statement is:

```
     WRITE (un'rn[,ERR=s][,IOSTAT=ios])[iolist]
```

In this form, the unit and record references do not contain the keywords UNIT= and REC=. Instead the unit number is specified first; a single quote (') is specified next; then the record number is specified last.

REREAD
Statement

10.7 REREAD STATEMENT

The REREAD statement causes the last record read from the last sequential formatted READ or ACCEPT statement to again be accessed and processed. You cannot use the REREAD feature until an input (READ) transfer has been accomplished. You can use the REREAD statement only for sequential-access formatted data transfers. The REREAD statement can be used with both FORMAT-statement formatting and list-directed formatting.

Once a record has been accessed by a formatted READ statement, the record transferred can be reread as many times as desired. You can use the same or a new format specification for each successive REREAD statement.

Table 10-6 summarizes all the forms of the REREAD statement.

Table 10–6: Summary of REREAD Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	REREAD (FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] REREAD f[,iolist]
Sequential Formatted (List Directed)	REREAD (FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] REREAD *[,iolist]

Key:

FMT=f	is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).
FMT=*	is list-directed formatting; iolist is optional (Section 10.4.5.2).
END=s	is an optional end-of-file transfer specifier (Section 10.4.6).
ERR=s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT=ios	is an optional I/O status specifier (Section 10.4.8)
iolist	is a data transfer I/O list (Section 10.4.9)

10.7.1 Sequential FORMAT-Statement REREAD

This section describes the sequential-access (FORMAT-statement) REREAD statement.

The first form of this statement is:

```
     REREAD (FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is specified in this form, the previous record is transferred from the logical unit (specified in the previous formatted READ statement) to the memory locations specified by the elements in the I/O list. The transferred record is formatted according to the FORMAT specifications given in f.

If you omit the I/O list from this statement, the input record is skipped. (If the FORMAT statement specifies slash editing, more than one record can be skipped. H or apostrophe editing can cause data transfers to occur to the FORMAT statement itself. See Section 12.4.)

The second form of this REREAD statement operates in the same way as the first form. The difference between the two forms is in the way the FORMAT specifiers are expressed.

The second form of this statement is:

```
     REREAD f[,iolist]
```

In this form, the keyword form of the FORMAT specifier (FMT=) is not used in the FORMAT reference. Whenever you use the keyword form of this specifier, you must enclose the keyword list in parentheses.

The following example shows the formatted REREAD being used:

```
       CHARACTER J*5
       DIMENSION J(5)
1      READ (20,5)A,X,I
5      FORMAT (2F10.2,I5)
10     REREAD 15,J
15     FORMAT (5A5)
       END
```

In the above sequence, statement 1 reads the two real variables A and X, and the integer I. Statement 10 rereads the last record input from unit 5 as a character string of 25 characters, five per word, and puts five characters per element into the array J.

10.7.2 Sequential List-Directed REREAD

This section describes the sequential-access (list-directed) REREAD statement.

The first form of this statement is:

```
REREAD (FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

In this form, the last record read by a formatted READ statement is transferred from the logical unit (specified in the formatted READ statement) into the memory locations identified by the elements of the I/O list. Since the formatting is list-directed, the format of the data is controlled by the data types of the elements in the I/O list. If no I/O list is included, no data is transferred.

The second form of this statement operates in the same way as the first form. The difference between the two forms is the way in which the formatting is specified.

The second form of this statement is:

```
REREAD *[,iolist]
```

The following example shows the list-directed form of the REREAD statement:

```
READ (20,*) A
REREAD *,B
END
```

In this example, the READ statement reads data from logical unit 20 into variable A. The REREAD statement rereads the data from logical unit 20 into variable B.

ACCEPT Statement

10.8 ACCEPT STATEMENT

The ACCEPT statement enables you to input data from your terminal into memory. You can use the ACCEPT statement only for sequential-access formatted data transfers. This statement can be used with both FORMAT-statement and list-directed formatting.

Table 10-7 summarizes all forms of the ACCEPT statement.

Table 10–7: Summary of ACCEPT Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	ACCEPT(FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT f[,iolist]
Sequential Formatted (List Directed)	ACCEPT(FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT *[,iolist]

Key:

FMT=f	is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).
FMT=*	is list-directed formatting; iolist is optional (Section 10.4.5.2).
END=s	is an optional end-of-file transfer specifier (Section 10.4.6).
ERR=s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT=ios	is an optional I/O status specifier (Section 10.4.8).
iolist	is a data transfer I/O list (Section 10.4.9).

10.8.1 Sequential FORMAT-Statement ACCEPT

This section describes the sequential-access (FORMAT-statement) ACCEPT statement.

The first form of this statement is:

```
     ACCEPT (FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

If you include the I/O list in this form, the data is taken from the terminal and stored in the memory locations identified in the I/O list. The transferred data is formatted according to the FORMAT specifications given in f.

If you omit the I/O list from this form, the input record is skipped. (If the FORMAT statement specifies slash editing, more than one record can be skipped. H or apostrophe editing can cause data transfers to occur to the FORMAT statement itself. See Section 12.4.)

The second form of this statement operates in the same way as the first form. The difference between the two forms is in how the FORMAT reference is expressed.

The second form of this statement is:

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

In this form, the keyword portion of the FORMAT specifier (FMT=) is omitted.

The following example shows both forms of the FORMAT-statement ACCEPT.

```
          ACCEPT  (FMT=35)A,Z,J
     35   FORMAT  (2F10.2,I5)
          ACCEPT  15,B
     15   FORMAT  (F10.2)
          END
```

In this example, the first ACCEPT statement accepts the values of the variables A, Z, and J from the terminal in the form of FORMAT statement 35. The second ACCEPT statement accepts the value of variable B from the terminal in the form of FORMAT statement 15.

10.8.2 Sequential List-Directed ACCEPT

The list-directed ACCEPT statement transfers data entered from the terminal into variables specified in the I/O list. The formatting of the transferred data is controlled by the data types of the items in the I/O list.

The first form of this statement is:

```
     ACCEPT (FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

In this form, the data is transferred from the terminal into the memory locations identified in the I/O list. Since the transfer is list-directed, the data is formatted according to the data types of the items in the I/O list. If no I/O list is included, a line is skipped.

The second form of this statement operates in the same way as the first form. The difference in the two forms is in how the list-directed formatting reference is specified.

The second form of this statement is:

```
     ACCEPT *[,iolist]
```

In the following example, both forms of the list-directed ACCEPT statement are used to take information, character-by-character, from the terminal. This example additionally shows the list-directed TYPE statement being used to print the ACCEPTed data at the terminal:

```
PROGRAM ACCTST
ACCEPT *,I,J,K
TYPE *,K,I,J
ACCEPT (FMT=*)G,H,F
TYPE *,H,F,G
END

EXECUTE ACCEPT.FOR
FORTRAN: ACCEPT
ACCTST
LINK:   Loading
[LNKXCT ACCTST execution]
23456 9876 12
12, 23456, 9876
12.34 98.16 789.67
98.16000, 789.6700, 12.34000
CPU time 0.2   Elapsed time 40.4
```

TYPE
Statement

10.9 TYPE STATEMENT

The TYPE statement enables you to output data to your terminal. Use the TYPE statement only for sequential-access formatted data transfers. This statement can be used with both FORMAT-statement and list-directed formatting.

Table 10-8 summarizes all the forms of the TYPE statement.

Table 10-8: Summary of TYPE Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	TYPE (FMT=f[,ERR=s][,IOSTAT=ios])[iolist] TYPE f[,iolist]
Sequential Formatted (List Directed)	TYPE (FMT=*[,ERR=s][,IOSTAT=ios])[iolist] TYPE *[,iolist]

Key:

FMT=f is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).

FMT=* is list-directed formatting; iolist is optional (Section 10.4.5.2)

ERR=s is an optional error transfer specifier (Section 10.4.7)

IOSTAT=ios is an optional I/O status specifier (Section 10.4.8).

iolist is a data transfer I/O list (Section 10.4.9)

10.9.1 Sequential FORMAT-Statement TYPE

This section describes the sequential-access (FORMAT-statement) TYPE statement.

The first form of this statement is:

```
TYPE (FMT=f[,ERR=s][,IOSTAT=ios])[iolist]
```

If you include the I/O list in this form, the data is transferred from the I/O list to your terminal. The transferred data is formatted according to the FORMAT specifications given in f.

A blank record is written if the I/O list is not specified, and one of the following is true:

1. The FORMAT statement is empty.

2. No slash, H, or apostrophe editing descriptors occur alone.

3. No slash, H, or apostrophe editing descriptors precede the first repeatable edit descriptors.

(See Section 12.4 for more information on format editing).

The second form of this statement operates in the same way as the first form. The difference between the two forms is in how the FORMAT reference is expressed.

The second form of this statement is:

```
TYPE f[,iolist]
```

In this form, the keyword portion of the FORMAT specifier (FMT=) is omitted.

The following example shows both forms of the FORMAT-statement TYPE:

```
         A=11.4
         Z=13.9
         J=5
         K=10
         TYPE (FMT=5)A,Z,J
    5    FORMAT (1X,2F10.2,I5)
         TYPE 15,K
    15   FORMAT (1X,I10)
         END
```

The following is typed on your terminal upon execution:

```
    LINK:    Loading
    [LNKXCT TEST40 execution]
         11.40     13.90    5
            10
    CPU time 0.22   Elapsed time 2.00
```

10.9.2 Sequential List-Directed TYPE

The list-directed TYPE statement transfers data from a program to the terminal. The formatting of the transferred data is controlled by the data types of the items in the I/O list.

The first form of this statement is:

```
    TYPE(FMT=*[,ERR=s][,IOSTAT=ios])[iolist]
```

In this form, the data is transferred from the program to the terminal. Since the transfer is list-directed, the data is formatted according to the data types of the items in the I/O list. If no I/O list is included, a blank record is written.

The second form of this statement operates in the same way as the first form. The difference between the two forms is in how the list-directed formatting reference is specified.

The second form of the statement is:

```
    TYPE *[,iolist]
```

The following example shows both forms of the list-directed TYPE statement:

```
    A=11.4
    Z=13.9
    J=5
    K=10
    TYPE (FMT=*),A,Z,J
    TYPE *,K
    END
```

The following is typed on the terminal upon execution:

```
    LINK:    Loading
    [LNKXCT TEST41 execution]
    11.40000,  13,90000, 5
    10
    CPU time 0.20   Elapsed time 0.87
```

10.10 PRINT STATEMENT

The PRINT statement transfers data from memory to the line printer. You can use the PRINT statement only for sequential-access formatted data transfers. This statement can be used with both FORMAT-statement formatting and list-directed formatting.

Table 10-9 summarizes all forms of the PRINT statement.

Table 10-9: Summary of PRINT Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	PRINT(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] PRINT f[,iolist]
Sequential Formatted (List Directed)	PRINT(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] PRINT *[,iolist]

Key:

FMT=f is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).

FMT=* is list-directed formatting; iolist is optional (Section 10.4.5.2).

ERR=s is an optional error transfer specifier (Section 10.4.7).

IOSTAT=ios is an optional I/O status specifier (Section 10.4.8).

iolist is a data transfer I/O list (Section 10.4.9).

10.10.1 Sequential FORMAT-Statement PRINT

This section describes the sequential-access (FORMAT-statement) PRINT statement.

The first form of this statement is:

```
     PRINT (FMT=f[,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is included in this form, the data identified by the I/O list is transferred from memory to the line printer. The formatting of the transferred data is controlled by the FORMAT specifications given in f.

A blank record is written if the I/O list is not specified, and one of the following is true:

1. The FORMAT statement is empty.

2. No slash, H, or apostrophe editing descriptors occur alone.

3. No slash, H, or apostrophe editing descriptors precede the first repeatable edit descriptor.

   See Section 12.4.

The second form of this statement operates in the same way as the first form. The difference between the two forms is in how the FORMAT specifier is expressed.

The second form of this statement is:

```
     PRINT f[,iolist]
```

The following example shows two PRINT statements; one with an I/O list and one without:

```
          A=7.6
          B=12.5
          C=20.9
          PRINT 10
          PRINT 20,A,B,C
     10   FORMAT (' Beginning of test')
     20   FORMAT (' Values are:',3F)
          END
```

The following is printed to the line printer upon execution:

```
     Beginning of test
     Values are:      7.6000000     12.5000000     20.9000001
```

10.10.2 Sequential List-Directed PRINT

This section describes the sequential-access (list-directed) PRINT statement.

The first form of this statement is:

```
     PRINT (FMT=*[,ERR=s][,IOSTAT=ios])[iolist]
```

This form of the PRINT statement specifies that the data identified by the elements of the I/O list is output on the line printer. The data is formatted according to the data types of the elements in the I/O list. If no I/O list is included, a blank record is written.

The second form of the list-directed PRINT statement operates in the same way as the first form. The difference between the two forms is in the way that the formatting is expressed.

The second form of this statement is:

```
     PRINT *[,iolist]
```

The following example shows the use of the list-directed PRINT statement:

```
     D=1
     E=40
     F=23.3
     PRINT *,D,E,F
     END
```

The following is printed to the line printer upon execution:

```
     1.000000, 40.00000, 23.30000
```

PUNCH Statement

10.11 PUNCH STATEMENT

The PUNCH statement transfers data from memory to the paper tape punch. You can use the PUNCH statement only for sequential-access formatted data transfers. This statement can be used with both FORMAT-statement formatting and list-directed formatting.

Table 10-10 summarizes all forms of the PUNCH statement.

Table 10–10: Summary of PUNCH Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	PUNCH(FMT = f[,ERR = s][,IOSTAT = ios])[iolist] PUNCH f[,iolist]
Sequential Formatted (List Directed)	PUNCH(FMT = *[,ERR = s][,IOSTAT = ios])[iolist] PUNCH *[, olist]

Key:

FMT = f	is FORMAT-statement formatting; iolist is optional (Section 10.4.5.1).
FMT = *	is list-directed formatting; iolist is optional (Section 10.4.5.2).
ERR = s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT = ios	is an optional I/O status specifier (Section 10.4.8).
iolist	is a data transfer I/O list (Section 10.4.9).

10.11.1 Sequential FORMAT-Statement PUNCH

This section describes the sequential-access (FORMAT-statement) PUNCH statement.

The first form of this statement is:

```
PUNCH (FMT=f[,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is specified in this form, the data identified by the items in the I/O list are transferred to the paper tape punch. The formatting of the data is controlled by the FORMAT specifications given in f.

A blank record is written if the I/O list is not specified, and one of the following is true:

1. The FORMAT statement is empty.
2. No slash, H, or apostrophe editing descriptors occur alone.
3. No slash, H, or apostrophe editing descriptors precede the first repeatable edit descriptor.

   See Section 12.4.

The second form of this statement operates in the same way as the first form. The difference between the two forms is in the way that the format specification is referenced.

The second form of this statement is:

```
PUNCH f[,iolist]
```

The following example shows the formatted PUNCH statement:

```
     PUNCH 10,A,B,C
10   FORMAT (3F)
```

10.11.2 Sequential List-Directed PUNCH

This section describes the sequential-access (list-directed) PUNCH statement.

The first form of this statement is:

```
PUNCH (FMT=*[,ERR=s][,IOSTAT=ios])[iolist]
```

This form of the PUNCH statement transfers the data identified by the elements of the I/O list to the paper tape punch. Since the transfer is list-directed, the formatting of the data is controlled by the data types of the items within the I/O list. If no I/O list is included, a blank record is written.

The second form of this statement operates in the same way as the first form. The difference between the two forms is in the way that the list-directed format reference is written.

The second form of this statement is:

```
     PUNCH *[,iolist]
```

The following example shows the list-directed PUNCH statement:

```
     PUNCH *,D,E,F
```

INTERNAL FILES AND ENCODE/DECODE Statements

10.12 INTERNAL FILES AND ENCODE/DECODE STATEMENTS

Internal READ/WRITE statements and ENCODE/DECODE statements are used for internal I/O.

Table 10-11 summarizes all the forms of the internal READ/WRITE and ENCODE/DECODE statements.

Table 10–11: Summary of Internal READ/WRITE and ENCODE/DECODE Statement Forms

Data Access	Statement Construct
Sequential Formatted (FORMAT Statement)	ENCODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist] DECODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist] READ(UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] READ(un,f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] WRITE(UNIT=un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist] WRITE(un,f[,ERR=s][,IOSTAT=ios])[iolist]

Key:

UNIT=un	is an Internal File identifier (Section 10.4.3.2).
c	is the total number of characters being transferred.
f	is a FORMAT–statement formatting reference.
a	is the name of the array from which or to which data is being transferred.
END=s	is an optional END–of–file specifier.
ERR=s	is an optional error transfer specifier (Section 10.4.7).
IOSTAT=ios	is an optional I/O status specifier (Section 10.4.8).
iolist	is a data transfer I/O list (Section 10.4.9).

10.12.1 Internal READ and WRITE Statements

The internal READ statement transfers data from an internal file to I/O list elements. The internal WRITE statement transfers data from I/O list elements to an internal file. Internal READ and WRITE statements are always formatted.

NOTE

> The DECODE statement can be used as an alternative to the internal READ statement, and the ENCODE statement can be used as an alternative to the internal WRITE statement. (See Section 10.3.1.1 for more information on internal files.)

The internal READ statement has the following forms:

```
READ (UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

READ (un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]

READ (un,f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]
```

In the above forms, un is an internal file identifier (see Section 10.4.3.2).

If an I/O list is included in these forms, it specifies that data is transferred from internal file identifier, un, formatted according to the specification given by f, and transferred into the elements of the specified I/O list.

If an I/O list is not included, the input record is skipped. (If the FORMAT statement specifies slash editing, more than one record can be skipped. Apostrophe or H editing can cause data transfers to occur to the FORMAT statement itself. See Section 12.4.)

The following example demonstrates the use of the internal READ statement:

```
     CHARACTER*9 STRING
     STRING = '3.14 6.02'
     READ(STRING,10) PI, A
10   FORMAT(F4.2, 1X, F4.2)
     WRITE(5,20) PI, A, PI+A
20   FORMAT(' PI=', F6.3, 5X, 'A=', F6.3, 5X, 'PI+A=', F6.3)
     STOP
     END
```

The READ statement in this example is an internal file read. It extracts the two numbers that are encoded in the character variable STRING, converts the numbers to floating point, and then stores them into the two variables PI and A. The following is printed at the terminal when the above program is executed:

```
EXECUTE IR.FOR
LINK:   Loading
[LNKXCT IR execution]

PI= 3.140     A= 6.020     PI+A= 9.160
CPU time 0.19   Elapsed time 0.40
```

The internal WRITE statement has the following forms:

```
WRITE (UNIT=un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un,FMT=f[,ERR=s][,IOSTAT=ios])[iolist]

WRITE (un,f[,ERR=s][,IOSTAT=ios])[iolist]
```

If the I/O list is included in these forms, the data specified by the elements of the I/O list are output to a file on internal file identifier un. The output data is formatted in this file according to the FORMAT specifications given in f.

A blank record is written if the I/O list is not specified, and one of the following is true:

1. The FORMAT statement is empty.

2. No slash, H, or apostrophe editing descriptors occur alone.

3. No slash, H, or apostrophe editing descriptors preceded the first repeatable edit descriptors.

   See Section 12.4.

The following example demonstrates the use of the internal WRITE statement:

```
     CHARACTER*20 CHARS(3)
     INTEGER PHNE(3)
     PHNE(1) = 617
     PHNE(2) = 481
     PHNE(3) = 4054
     WRITE(CHARS,10) (I, PHNE(I), I=1,3)
10   FORMAT( 'PHNE(', I1, ')=', I4)
     WRITE(5,20) (I, CHARS(I), I=1,3)
20   FORMAT(' Record ', I1, ' of CHARS is "', A20, '"')
     STOP
     END
```

The first WRITE statement in the above program is an internal file write. Since the character variable being written to is a three element array, the internal file is a file of three records. When this program is executed, the following is output to the terminal:

```
EXECUTE IW.FOR
LINK:   Loading
[LNKXCT IW execution]

Record 1 of CHARS is "PHNE(1)= 617        "
Record 2 of CHARS is "PHNE(2)= 481        "
Record 3 of CHARS is "PHNE(3)=4054        "
CPU time 0.24   Elapsed time 0.82
```

10.12.2 ENCODE and DECODE Statements

The DECODE statement can be used as an alternative to an internal READ, and the ENCODE statement can be used as an alternative to the internal WRITE.

The ENCODE statement transfers data from the variables of a specified I/O list into a specified array. ENCODE operations are similar to those performed by a WRITE statement.

The DECODE statement transfers data from a specified array into the variables of an I/O list. DECODE operations are similar to those performed by a READ statement.

ENCODE and DECODE statements have the following forms:

```
ENCODE (c,f,a[,ERR=s][,IOSTAT=ios])[iolist]

DECODE (c,f,a[,ERR=s][,IOSTAT=ios])[iolist]
```

where:

c specifies the number of characters in each internal record of the array. This argument can be any integer expression, and must be the first specification in the statement.

NOTE

Five characters per word are stored in the array without regard to the type of the array.

f specifies either a FORMAT-statement or a numeric array that contains format specifications. This must be the second specification.

a specifies the array, array element, variable, or character substring reference that is to be used in the transfer operations, and it must contain at least c characters. More than one element of the array can be used by the ENCODE/DECODE.

iolist specifies an I/O list of the standard form.

When multiple records are stored by ENCODE or read by DECODE, each new record starts c characters after the previous record; no CR/LF (carriage return/line feed) is inserted between records.

NOTE

If the array contains fewer characters than required by the format and the I/O list, the variables following the array in memory are used. If the processing of the I/O list requires more characters in a single record than are specified by the character count c, blanks are used.

The following example shows how the ENCODE and DECODE statements are used:

```
        DIMENSION B(4),A(2)
        A(1)=300.45
        A(2)=3.0
        C='12345'

        DO 2 J=1,2

        ENCODE(16,10,B)J,A(J)
10      FORMAT(1X,'A(',I1,') = ', F8.2)
```

```
 5      TYPE 11,B
11      FORMAT(4A5)

 2      CONTINUE

        DECODE(5,12,C)B
12      FORMAT(3F1.0,1X,F1.0)

        TYPE 13,B
13      FORMAT(4F5.2)

        END
```

During the first iteration of the DO loop, the ENCODE statement has transferred the contents of variable J and array element A(1) into array B. The formatting of the data being transferred is specified by the FORMAT statement at line 10.

After the first iteration of the DO loop, the contents of array B are:

```
B(1) = ' A(1)'
B(2) = ' =   '
B(3) = '300.4'
B(4) = '5    '
```

The TYPE statement at line 5 types array B on the terminal during the first iteration of the DO loop.

During the second iteration of the DO loop, the data is transferred from variable J and array element A(2) into array B. After the second iteration, the contents of array B are:

```
B(1) = ' A(2)'
B(2) = ' =   '
B(3) = '  3.0'
B(4) = '0    '
```

The TYPE statement at line 5 types array B on the terminal during the second iteration of the DO loop.

The DECODE statement:

1. Extracts the digits 1, 2, and 3 from C

2. Converts them to floating-point values

3. Stores them in B(1), B(2), and B(3)

4. Skips the next character (the digit 4)

5. Extracts the digit 5 from C

6. Converts it to a floating-point value

7. Stores the value in B(4)

The following shows what is printed at the terminal when the above program is executed:

```
EXECUTE T.FOR
LINK:   Loading
[LNKXCT T execution]
A(1) =  300.45
A(2) =    3.00
1.00 2.00 3.00 5.00
CPU time 0.1   Elapsed time 0.8
```

CHAPTER 11

FILE-CONTROL AND DEVICE-CONTROL STATEMENTS

11.1 FILE-CONTROL STATEMENTS

Prior to transferring any data using one of the forms of data transfer statements, you can establish a connection between a logical unit and a file by using the OPEN statement. After the completion of a data transfer, you can terminate the connection between the logical unit and the file before ending the program by using the CLOSE statement.

The OPEN statement enables you to explicitly connect a logical unit to a file prior to the first data transfer, and also to specify a variety of characteristics about the file and the data transfers.

After the last data transfer is completed, the CLOSE statement enables you to explicitly disconnect the logical unit from the file and, optionally, to specify a variety of characteristics about the CLOSE.

If you do not precede an I/O statement with an OPEN statement, FOROTS automatically performs an "implicit OPEN" (see Section 11.2.1).

Similarly, if you do not specify a CLOSE statement to explicitly disconnect a file from a logical unit, FOROTS performs an "implicit CLOSE" (see Section 11.4.1) when your program terminates.

You need not specify the OPEN and CLOSE statements if the actions performed by the implicit OPEN or CLOSE are satisfactory.

OPEN Statement

11.2 OPEN STATEMENT

The OPEN statement is used to specify characteristics of a file that you wish to read or write. An example of an OPEN statement is:

```
     OPEN (UNIT=20,FILE='MYDATA.DAT')
```

The specifiers inside the parentheses give information about the file and determine how the file is opened.

The UNIT specifier (in the example above, "UNIT=20") is required in an OPEN statement. All other specifiers are optional, including the FILE specifier in the example shown above. You can supply many other optional specifiers (see Section 11.3 for a description of OPEN statement specifiers). The order in which the specifiers appear does not affect the execution of the OPEN statement.

By using the OPEN specifiers, you are able to define certain characteristics of each data transfer, including:

1. The name of the data file

2. The type of access required

3. The data format of the file

4. The disposition of the data file

5. The data file record and block sizes

In addition, a DIALOG argument permits you to establish a dialog mode of operation when the OPEN statement containing it is executed. In a dialog mode, interactive terminal/program communication is established, enabling the user to define or redefine the values of the OPEN statement specifiers.

When a file is open for output (STATUS='NEW' or ACCESS='SEQOUT'), a null file is created on the device specified by FILE= DEVICE=, or if none, the first structure in the job's search list.

An OPEN statement is referred to as a "deferred" OPEN statement if both of the following are true:

- The OPEN statement specifies STATUS='UNKNOWN' (or does not specify a STATUS value).

- The OPEN statement specifies ACCESS='SEQINOUT' or 'SEQUENTIAL' (or does not specify an ACCESS value).

The actual opening of the file is deferred until the first data transfer statement (READ, WRITE, PRINT, PUNCH, or SKIPRECORD). The actual opening of the file means the determination of the physical device, and for TOPS-20, the generation number (if not explicitly specified).

If the first data transfer statement is a READ or SKIPRECORD the first file that matches the file specification given in the OPEN statement is opened. If no file exists that matches the file specification given, a null file is created on the device specified by FILE= or DEVICE=, or if none, the first structure in the job's search list. The file is positioned as if a READ or SKIPRECORD statement had been executed, and an end-of-file error will be generated (see END=, Section 10.4.6).

If the first data transfer statement is a WRITE, PRINT, or PUNCH statement, a new file (with a new generation on TOPS-20) will be created on the device specified by FILE= or DEVICE=, or if none, the first structure in the job's search list.

If the file specified in the OPEN statement does not exist, and either a CLOSE statement is executed or the program runs to completion, a null file is created on the device specified by FILE= or DEVICE=, or if none, the first structure in the job's search list.

11.2.1 Implicit OPEN

When the OPEN statement has not been executed before a data transfer that references the unit number, an implicit OPEN is performed.

An implicit OPEN has almost exactly the same effect as if you had put an OPEN statement with the following format in the program just before the data transfer statement:

```
     OPEN (UNIT=un,STATUS='UNKNOWN',FORM=fm)
```

where:

un — is the unit number specified in the data transfer statement.

fm — is 'UNFORMATTED' if the data transfer statement is an unformatted READ or WRITE statement; otherwise fm is 'FORMATTED'.

In addition, if the data transfer statement has an ERR specifier, the implicit OPEN has this same qualifier included. This is also true of the IOSTAT specifier.

NOTE

> The default for the BLANK specifier is different depending on whether the OPEN is implicit or explicit (see Section 11.3.3).

11.2.2 OPEN on a Connected Unit

If the OPEN statement contains a STATUS=OLD specifier (see Section 11.3.29), then its action depends on whether a file is already OPEN on the unit, and whether the file specified by the OPEN is the same file that is currently on the unit. If the file specified by the OPEN is different from the OPEN file, the connected file is closed and the new file is opened. If the file specified by the OPEN is the same as the connected file, the file is not closed, and the file pointer is not moved. This action is not affected by the /F66 compiler switch (described in Chapter 16).

11.3 OPEN STATEMENT SPECIFIERS

All of the OPEN statement specifiers are optional, except the UNIT specifier, which is required. Some specifiers have default values that can depend on the unit number or the values of other specifiers.

Table 11-1 summarizes the specifiers in the OPEN statement and the type of value required by each. A section number is provided to refer to detailed descriptions of each specifier. The CLOSE statement specifiers are summarized in Table 11-5.

Table 11–1: Summary of OPEN Statement Specifiers and Arguments

Argument	Possible Value	Section
ACCESS=	Character expression with one of the following values: 'SEQIN', 'SEQOUT', 'SEQINOUT', 'SEQUENTIAL', 'DIRECT', 'RANDOM' 'RANDIN', 'APPEND'	11.3.1
ASSOCIATEVARIABLE	Integer variable or integer array element	11.3.2
BLANK =	Character expression with one of the following values: 'NULL', 'ZERO'	11.3.3
BLOCKSIZE	Integer expression	11.3.4
BUFFERCOUNT=	Integer expression	11.3.5
CARRIAGECONTROL	Character expression with one of the following values: 'FORTRAN', 'LIST', 'DEVICE'	11.3.6
DENSITY	Character expression with one of the following values: '200', '556', '800', '1600', '6250', 'SYSTEM'	11.3.7
DEVICE	Character expression	11.3.8
DIALOG		11.3.9
DIALOG	Character expression	11.3.10
DIRECTORY (TOPS-10)	Character expression	11.3.11
DIRECTORY (TOPS-20)	Character expression	11.3.12
DISPOSE	Character expression with one of the following values: 'SAVE', 'DELETE', 'PRINT', 'KEEP', 'LIST', 'PUNCH', 'EXPUNGE'	11.3.13
ERR=	Statement number	11.3.14
FILE=	Character expression	11.3.15
FILESIZE INITIALIZE	Integer expression	11.3.16
FORM=	Character expression with one of the following values: 'FORMATTED', 'UNFORMATTED'	11.3.17
IOSTAT=	Integer variable or integer array element	11.3.18
LIMIT	Integer expression	11.3.19
MODE	Character expression with one of the following values: 'ASCII', 'LINED', 'BINARY', 'IMAGE', 'DUMP'	11.3.20
NAME	Character expression	11.3.21
PADCHAR	A character expression in which the first character is used	11.3.22
PARITY	Character expression with one of the following values: 'ODD', 'EVEN'	11.3.23
PROTECTION (TOPS-10)	Integer expression	11.3.24

Table 11-1: Summary of OPEN Statement Specifiers and Arguments (Cont.)

Argument	Possible Value	Section
PROTECTION (TOPS-20)	Integer expression	11.3.25
READONLY		11.3.26
RECL - RECORDSIZE	Integer expression	11.3.27
RECORDTYPE	Character variable, array element, or substring reference	11.3.28
STATUS - TYPE	Character expression with one of the following values: 'OLD', 'NEW', 'SCRATCH', 'EXPUNGE', 'UNKNOWN', 'KEEP', 'DELETE'	11.3.29
TAPEFORMAT	Character expression with one of the following values: CORE DUMP or INDUSTRY	11.3.30
UNIT -	Integer expression	11.3.31
VERSION	Octal constant, integer variable, or integer array element	11.3.32

NOTE

For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric array name as the value of each of the following specifiers:

DIALOG=
DIRECTORY
NAME

When a numeric array name is used, FOROTS assumes that it contains a string of characters terminated by a null character.

In addition, you can specify a numeric variable as the value of the DEVICE and FILE specifiers. If the variable is single precision, FOROTS assumes that it contains 5 characters; if it is double precision, FOROTS assumes that is contains 10 characters.

The use of numeric array names and numeric variables in place of character variables is a nonstandard feature.

ACCESS
Specifier

11.3.1 ACCESS Specifier

The ACCESS specifier describes the type of data transfer statements allowed. Records within files can be accessed directly (randomly) and sequentially.

The form of the ACCESS specifier is:

```
ACCESS = acc
```

where:

acc is a character expression having a value equal to one of the following:

```
'SEQIN'
'SEQOUT'
'SEQINOUT'
'SEQUENTIAL'
'DIRECT'
'RANDOM'
'RANDIN'
'APPEND'
```

ACCESS has a number of arguments, each of which specifies a method of data access. SEQUENTIAL is the default access unless the device (UNIT) opened is a read-only device, in which case the default is SEQIN. If the device opened is a write-only device, the default access is SEQOUT.

The arguments to the ACCESS specifier are:

SEQIN (Implies STATUS='OLD') The specified data file is opened for read-only sequential access. When ACCESS='SEQIN' is specified, it is equivalent to specifying ACCESS='SEQUENTIAL' and READONLY (see Section 11.3.26).

SEQOUT The specified data file is opened for output and sequential access. If the specified file already exists, it is superseded (TOPS-10), or a new generation is created (TOPS-20).

SEQUENTIAL The specified data file is opened for sequential access. Records can be read from or written to the file in sequential order. However, when a record is written to the file, it becomes the last record of the file. Any data following that record becomes inaccessible.

Records can also be written to the file and then read, as long as a device-positioning statement (BACKSPACE or REWIND, Section 11.8) is used before the READ statement.

SEQINOUT Same as SEQUENTIAL

DIRECT The specified data file may be read from and/or written to in units of fixed-length records. The record to be accessed next is specified in the data transfer statement by a record number.

The relative position of each record is independent of the previous READ or WRITE statement. The RECL specifier (see Section 11.3.27) is required for random-access operations. You must specify a disk device when the DIRECT argument is used.

RANDIN	(Implies STATUS='OLD') The specified data file is opened for read-only direct access. More than one user can read the same file at the same time with ACCESS='RANDIN'. When ACCESS='RANDIN' is specified, it is equivalent to specifying ACCESS='RANDOM' and READONLY (see Section 11.3.26).
RANDOM	Same as DIRECT
APPEND	The specified file is opened for sequential write-only access. APPEND is the same as SEQOUT except that the file is positioned at its end after the OPEN statement. Reading an APPEND mode file is illegal. REWIND and BACKSPACE are illegal for files opened with APPEND access.

ASSOCIATEVARIABLE
Specifier

11.3.2 ASSOCIATEVARIABLE Specifier

This specifier enables you to declare a variable whose value is the number of the next record that will be read from or written to the file.

For example, after the execution of an OPEN statement and prior to the first data transfer, the associate variable is set to 1.

In a data transfer after the first record is transferred, the value of the associate variable is 2.

The form of the ASSOCIATEVARIABLE specifier is:

ASSOCIATEVARIABLE= Integer variable or integer array element

If you are using the ASSOCIATEVARIABLE specifier in a program that makes use of the LINK overlay facility, please read the paragraphs that follow.

If the variable you specify as the ASSOCIATEVARIABLE is declared in a FORTRAN subroutine, then that subroutine must be loaded in the root link of the overlay structure. If the subroutine cannot be loaded in the root link of the overlay structure, declare your ASSOCIATEVARIABLE in a COMMON statement so that the ASSOCIATEVARIABLE will operate properly.

The reasons for these steps are:

1. When the overlay facility is used to load FORTRAN modules, the local variables in the modules are grouped with the routine in which they are declared.

2. When FORTRAN subroutines are loaded by the overlay facility, they are divided into sets called overlay links.

3. Only one overlay link, the one specified to be the root link, is always resident in memory. The other overlay links are loaded in memory as required.

Accessing a file opened with an ASSOCIATEVARIABLE changes the value of the specified variable. If this variable is in a nonresident overlay link when the access is made, program execution produces unpredictable results. Moreover, this variable is reset to zero each time its overlay link is removed from memory.

Only variables declared in routines loaded into the root link are always resident. Variables declared in COMMON statements and those declared in the main program are always resident and can always be used as an associate variable.

NOTE

> For more information on the LINK overlay facility, see the LINK Programmer's Reference Manual, and Chapter 15 of this manual.

BLANK Specifier

11.3.3 BLANK Specifier

The BLANK specifier applies only when reading formatted (FORMAT-statement) numeric fields that have a field width specified. BLANK enables you to specify how blanks in formatted numeric fields are treated in a read transfer (either as zero or ignored).

The form of the BLANK specifier is:

```
     BLANK = blnk
```

where:

blnk is a character expression having a value equal to either 'NULL' or 'ZERO'.

The arguments to the BLANK specifier are:

NULL specifies that all blank characters within numeric formatted input fields are ignored. The exception is that a field of all blanks has a value of zero.

ZERO specifies that all blanks are treated as zeros.

If an OPEN statement is executed and the BLANK specifier is not given, the default is BLANK='NULL'.

If no explicit OPEN statement is executed before a data transfer on a unit, the default is BLANK='ZERO' for all devices except terminals. For terminals, the default is always BLANK='NULL' regardless of whether or not the OPEN statement is given.

The BLANK specifier is overridden if a corresponding data transfer statement references a format list that contains either the BN or BZ descriptor. In this case, the BN or BZ descriptor in the format list overrides the setting in the OPEN statement until the end of the format list, or until the setting is changed within the format list. (The BN or BZ descriptors are described in Section 12.4.9.)

Example:

```
      OPEN(UNIT=1,DEVICE='DSK',FILE='FOO.DAT',BLANK='ZERO')
      READ(1,10)K
   10 FORMAT(I5)
      CLOSE(UNIT=1)

      OPEN(UNIT=1,DEVICE='DSK',FILE='FOO.DAT',BLANK='NULL')
      READ(1,10)L
      CLOSE(UNIT=1)

      END
```

In the above example, if FOO.DAT contains 123bb, K has the value 12300 and L has the value 123.

BLOCKSIZE Specifier

11.3.4 BLOCKSIZE Specifier

The BLOCKSIZE specifier enables you to specify a physical storage block size for magnetic tape files.

NOTE

> BLOCKSIZE specifies the physical record length, and RECL(RECORDSIZE) specifies the logical record length.

The argument is an integer expression, and for CORE-DUMP tape format, the value assigned represents the number of words in the physical block. For INDUSTRY tape format, the value represents the number of bytes in the physical block. (See the TAPEFORMAT specifier, Section 11.3.30.)

The form of the BLOCKSIZE specifier is:

```
   BLOCKSIZE= Integer expression
```

BUFFERCOUNT Specifier

11.3.5 BUFFERCOUNT Specifier

The BUFFERCOUNT specifier enables you to define the number of I/O buffers used in the data transfer.

The BUFFERCOUNT is the number of pages used in disk transfers, and is ignored for nondisk transfers.

The form of the BUFFERCOUNT specifier is:

    BUFFERCOUNT= Integer expression

If a BUFFERCOUNT is not specified, or is assigned a value of zero, the buffercount is four pages.

NOTE

> If MODE='DUMP' is specified, BUFFERCOUNT is ignored.

The BUFFERCOUNT specifier does not affect the operation of the program, but it can affect execution time and memory requirements.

For random I/O, the buffercount specifies the maximum number of buffers which are in memory (not yet written to disk) during I/O operations.

NOTE

> For TOPS-20 extended addressing, all I/O buffers must fit in FOROTS's section.

CARRIAGECONTROL
Specifier

11.3.6 CARRIAGECONTROL Specifier

The CARRIAGECONTROL specifier enables you to decide how the first character of each record encountered during an output data transfer operation is treated. (Section 12.2.3 describes carriage-control specifiers.)

The form of the CARRIAGECONTROL specifier is:

    CARRIAGECONTROL = cc

where:

cc is a character expression having a value equal to one of the following:

    'FORTRAN'
    'LIST'
    'TRANSLATED'
    'DEVICE'

The arguments to the CARRIAGECONTROL specifier are:

FORTRAN — specifies that the FORTRAN data file attribute is set in the file's access information, so that when the file is printed, the first character of each record is replaced with a carriage return and the corresponding printer-control vertical motion character(s) (see Table 12-3). The record terminator at the end of the record will be discarded.

LIST — specifies that the first character is output with no replacement.

TRANSLATED — specifies that the first character of each record is replaced with a carriage return and the corresponding printer-control vertical motion character(s) (see Table 12-3). No record terminator is written at the end of the record.

Note that the last data record in the file has no ending carriage return/line feed sequence unless a blank record is written.

For fixed-length files, TRANSLATED is treated as LIST.

DEVICE — specifies that the first character will only be replaced for a carriage-control device (such as LPT or TTY).

The default value is CARRIAGECONTROL='DEVICE'.

NOTE

> The line printer software assumes that the first character of all data files is a carriage-control character if the file has the extension .DAT or if the /FILE:FORTRAN switch is specified.

DENSITY Specifier

11.3.7 DENSITY Specifier

The DENSITY specifier is ignored except when used with magnetic tape; it permits you to specify the tape density. If you do not specify a tape density, FORTRAN assumes that you have set the density at monitor level or that you are satisfied with the system default for the device.

The form of the DENSITY specifier is:

```
    DENSITY = dens
```

where:

dens is a character expression having a value equal to one of the following:

'200'
'556'
'800'
'1600'
'6250'
'SYSTEM'

SYSTEM specifies that the density is the default density for the magnetic tape device being used.

DEVICE
Specifier

11.3.8 DEVICE Specifier

The DEVICE specifier enables you to specify the name of the device involved in the data transfer. A logical name always takes precedence over a physical name. The DEVICE arguments can specify I/O devices located at remote stations, as well as logical devices.

The form of the DEVICE specifier is:

```
DEVICE= Character expression
```

If you omit this option, the logical name un (where un is the decimal unit number) is tried. If this is not successful, the standard (default) device associated with the unit is used (see Table 10-3).

DIALOG
Specifier

11.3.9 DIALOG Specifier

The DIALOG specifier enables you to type in additional OPEN specifiers when the OPEN statement is actually executed.

If the DIALOG specifier is found in an OPEN list, then each time the OPEN statement is executed FOROTS suspends execution and prompts the terminal with an asterisk.

The form of the DIALOG specifier is:

```
DIALOG
```

You can respond to the asterisk prompt by entering a file specification, DIALOG switches (see below), or a file specification followed by DIALOG switches. The file specification may be a full file specification including the device, directory name, and so on.

NOTE

> A DIALOG switch is any OPEN specifier (except DIALOG, DIALOG=, UNIT, NAME, FILE, IOSTAT, or ERR) preceded by a slash (/).

For example, when FOROTS enters DIALOG mode, you can type a string such as:

```
     *DSK:FOO.BAR/MODE:BINARY/ACCESS:DIRECT
```

DIALOG=
Specifier

11.3.10 DIALOG= Specifier

The DIALOG= specifier enables you to include all or a portion of the OPEN specifiers in a character expression. The contents of the character expression are interpreted as if you had given the DIALOG specifier (see above); and, when the asterisk prompt was given, you had typed in the same string as is contained in the character expression, followed by a carriage return.

The form of the DIALOG= specifier is:

```
     DIALOG= Character expression
```

Example:

```
         OPEN(UNIT=1,DIALOG='DSK:FOO.BAR/ACCESS:SEQOUT/MODE:ASCII')
         I=22
         WRITE(1,100)I
     100 FORMAT(I5)
         END
```

DIRECTORY (TOPS-10)
Specifier

11.3.11 DIRECTORY Specifier (TOPS-10)

On TOPS-10, the DIRECTORY specifier is ignored except for disk files. It specifies the location of the User File Directory (UFD) and, optionally, the Sub File Directory (SFD), either of which can contain the file specified in the OPEN statement.

The form of the DIRECTORY specifier is:

```
     DIRECTORY= Character expression
```

The UFD is the directory in which a user's files are stored; the SFD exists within the UFD. An SFD is often used to group files into separate subdirectories.

The following is a sample of the UFD and SFD specification:

```
     10,7,SFDA,SFDB
```

In the sample specification, 10,7 is the project-programmer number. This is an adequate directory specification if the file is in the UFD specified by 10,7. The SFDA and SFDB specify two levels of subfile directories. The complete directory specification indicates that the file is located in subfile directory SFDB. As indicated, the path to SFDB is through the UFD 10,7 and through the SFD SFDA.

NOTE

> Refer to the TOPS-10 Monitor Calls Manual for a complete description of directories and multilevel directory structures.

The following is an example of a character expression specification:

```
     DIRECTORY='10,7,SFD1,SFD2,SFD3'

                Project     SubFile
                Programmer  Directory
                Number      Path
```

The following is an example of how to assemble a specification from individual elements:

```
     CHARACTER*10 PROJ,PROG,PATH1,PATH2
     CHARACTER*1 COMM

     PROJ= '10'
     PROG= '7'
     PATH= 'SFDA'
     PATH= 'SFDB'
     COMM= ','
     OPEN(UNIT=1,DIRECTORY=PROJ//COMM//PROG//COMM//PATH1//COMM//PATH2)
```

The above specification is equivalent to the following character expression:

```
     '10,7,SFDA,SFDB'
```

DIRECTORY (TOPS–20)
Specifier

11.3.12 DIRECTORY Specifier (TOPS-20)

On TOPS-20, the DIRECTORY specifier is ignored except for disk files. The DIRECTORY specifier enables you to define the path through the directory structure to a file specified in the OPEN statement.

The form of the DIRECTORY specifier is:

```
     DIRECTORY=  Character expression
```

The argument to the DIRECTORY specifier is a character expression whose elements comprise the directory path specification, for example:

```
     OPEN(UNIT=22,DIRECTORY= 'GUEST')
     !Looks for DSK:<GUEST>FOR22.DAT

     or

     CHARACTER*12 ID
     ID= 'GUEST.CLASS3'
     OPEN(UNIT=22,DIRECTORY=ID)
     !Looks for DSK:<GUEST.CLASS3>FOR22.DAT
```

DISPOSE
Specifier

11.3.13 DISPOSE Specifier

The DISPOSE specifier enables you to specify an action to occur when the file is closed.

The form of the DISPOSE specifier is:

```
     DISPOSE = dis
```

where:

dis is a character expression having a value equal to one of the following:

```
     'KEEP'
     'SAVE'
     'DELETE'
     'EXPUNGE'
     'PRINT'
     'LIST'
     'PUNCH'
```

The DISPOSE specifier must have one of the following values:

KEEP — Specifies that the file is to be left where the OPEN statement specifies. DISPOSE='KEEP' is the default.

SAVE — Same as KEEP.

DELETE — Specifies on TOPS-10 that, if the device is either a DECtape or disk, delete the file; otherwise, take no action.

On TOPS-20, if the device involved is a disk, delete the file; otherwise, take no action.

EXPUNGE — On TOPS-10, same as DELETE. On TOPS-20, if the device involved is a disk, expunge the file; otherwise, take no action.

PRINT — Specifies that the file will be printed and kept. The file must be on disk.

LIST — Specifies that the file will be printed and deleted. The file must be on disk.

PUNCH — Specifies that the file will be punched on the paper tape punch and kept. The file must be on disk.

ERR Specifier

11.3.14 ERR Specifier

The ERR specifier enables you to designate a statement number of an executable statement, in the current program unit, to which control passes if an error occurs during the execution of an I/O statement.

If an error occurs and no ERR specifier or IOSTAT specifier (see Section 11.3.18) is supplied, the program types an error message. If the program is running under batch, it is aborted.

If the program is not running under batch, it enters DIALOG mode after processing all of the other specifiers, as if you had supplied the DIALOG specifier (see Section 11.3.9). This is true regardless of whether or not the OPEN statement was explicitly executed or implied by the execution of the first data transfer statement for a unit.

The form of the ERR specifier is:

```
    ERR= s
```

where:

s — is the number of an executable statement to which program control passes if an error occurs during the execution of the statement that includes the ERR specifier.

The subroutine ERRSNS can be called to pinpoint the error. See Appendix D for FOROTS error values returned by ERRSNS.

**FILE
Specifier**

11.3.15 FILE Specifier

The FILE specifier enables you to name the file involved in the data transfer operation. You can specify a full file specification.

The form of the FILE specifier is:

```
     FILE= Character expression
```

The value of the character expression is any legal TOPS-10 or TOPS-20 file specification. (See the TOPS-10 Operating System Commands Manual or the TOPS-20 User's Guide.)

If you omit the period and extension, the extension .DAT is assumed. If just the extension is omitted, a null extension is assumed. Thus, if you want a filename without an extension, remember to use the period.

If a filename is not specified, a default name is generated that has the form:

```
     FORxx.DAT
```

where:

xx   is the FORTRAN logical unit number (decimal) or the logical unit name for the default statements ACCEPT, PRINT, PUNCH, READ, WRITE, or TYPE.

**FILESIZE
(INITIALIZE)
Specifier
(TOPS-10 only)**

11.3.16 FILESIZE (INITIALIZE) Specifier (TOPS-10 Only)

The FILESIZE (or INITIALIZE) specifier is used for disk operations only. It enables you to estimate the number of words that an output file is going to contain.

The form of the FILESIZE specifier is:

```
     FILESIZE= Integer expression
```

The value assigned as a FILESIZE argument can be a integer expression, and is rounded up to the next higher block boundary (multiple of 128).

The value specified by FILESIZE= is used as an estimate only. The effect of FILESIZE= is to help the monitor try to choose the best place on the disk to put the file.

**FORM
Specifier**

11.3.17 FORM Specifier

The FORM specifier designates whether the records in a data transfer operation are formatted or unformatted. You should not mix formatted (character) and unformatted (binary) records in the same file.

The form of the FORM specifier is:

```
FORM = ft
```

where:

ft is a character expression having a value equal to 'FORMATTED' or 'UNFORMATTED'.

The arguments to the FORM specifier are:

FORMATTED specifies that the records being transferred contain character (formatted) data.

UNFORMATTED specifies that the records being transferred contain binary (unformatted) data.

If FORM is not specified and MODE is 'ASCII' or 'LINED', the default value for FORM is 'FORMATTED'. Otherwise, if MODE is 'BINARY', 'IMAGE', or 'DUMP' (TOPS-10 only), the default value for FORM is 'UNFORMATTED'.

If both FORM and MODE are specified and they are incompatible, then DIALOG mode is entered, and you are asked to correct the incompatibility. In the following example, MODE='BINARY' and FORM='FORMATTED' are specified in the same OPEN statement. As shown below, when the program is executed, interactive DIALOG mode is automatically entered to enable the user to correct the incompatibility.

```
          PROGRAM TRIMP
          OPEN(UNIT=1,MODE='BINARY',FORM='FORMATTED')
          WRITE(UNIT=1,FMT=101)
     101  FORMAT(1X,'This is a test.')
          END
```

```
     EXECUTE TRIMP
     LINK:    Loading
     [LNKXCT TRIMP execution]
     ?OPEN unit 1  DSK:FOR01.DAT at MAIN.+4 in TRIMP (PC 165)
     ?Incompatible attributes /MODE:BINARY /FORM:FORMATTED
```

```
[Enter correct file specs]
*/MODE:ASCII
CPU time 0.3   Elapsed time 11.4
TYPE FOR01.DAT
This is a test.
```

If neither FORM nor MODE is specified the default value for FORM depends on the access. If the access is SEQUENTIAL (or is defaulted), the default for FORM is FORMATTED. If the access is DIRECT or RANDOM, the default for FORM is UNFORMATTED.

NOTE

> For ASCII devices (line printer, plotter, terminal, industry magnetic tape), the FORM= specifier has no meaning and is ignored; both formatted and unformatted data transfers are legal (see Section 10.3.3).

IOSTAT Specifier

11.3.18 IOSTAT Specifier

The IOSTAT specifier identifies an integer variable that is used to store the I/O status code during the execution of a statement.

The form of the IOSTAT specifier is:

```
IOSTAT= Integer variable or integer array element
```

If no error occurs during the execution of the statement, the defined variable is set to zero.

If an error does occur during the execution of the statement, the defined variable is assigned a positive integer value that corresponds to the number of the FOROTS error that occurred (see Appendix D for FOROTS error codes).

When an error occurs, no error message is typed; instead, the program either continues at the ERR= statement number (if the ERR specifier is included), or continues at the statement immediately following the OPEN statement (if no ERR specifier is included).

LIMIT
Specifier

11.3.19 LIMIT Specifier

The LIMIT specifier designates the number of output units (such as pages) for spooled print or punch requests, which result from using DISPOSE='PRINT', DISPOSE='PUNCH', or DISPOSE='LIST' (see Section 11.3.13).

The form of the LIMIT specifier is:

```
     LIMIT= Integer expression
```

MODE
Specifier

11.3.20 MODE Specifier

The MODE specifier defines the data mode of an external file or record.

The form of the MODE specifier is:

```
     MODE = mod
```

where:

mod is a character expression having a value equal to one of the following:

```
'ASCII'
'LINED'
'BINARY'
'IMAGE'
'DUMP'
```

After a MODE has been assigned (either explicitly or by default), it cannot be changed until the file is closed and then reopened.

The default value of MODE depends on the values of FORM and ACCESS. If FORM is FORMATTED, then the default MODE is ASCII. If FORM is UNFORMATTED, then the default MODE is BINARY. If ACCESS is SEQUENTIAL and no FORM= is specified, then the default MODE is ASCII; if ACCESS is DIRECT or RANDOM, and no FORM= is specified, then the default MODE is BINARY. (See Section 11.3.17 for details on FORM, and Section 11.3.1 for details on ACCESS.)

Character data is supported in formatted BINARY and IMAGE mode files; it is not supported in DUMP mode files.

The possible values of MODE are:

ASCII — Specifies the data to be 7-bit ASCII characters. Records are terminated with a line feed, form feed, or vertical tab.

LINED — Takes effect on input only. For output, this mode defaults to ASCII. LINED specifies the data to be 7-bit ASCII characters with optional line sequence numbers. FORTRAN removes the line sequence numbers, if present, before supplying the data to the user. (The line sequence number can be obtained by using the function LSNGET, see Section 13.3.1.)

Note that a page mark in a file containing line sequence numbers is a separate record. FORTRAN removes the blank sequence number, and the carriage return/form feed sequence is read as a blank record.

BINARY — Specifies that data is formatted as a FORTRAN binary data file. A BINARY file is composed of 36-bit words (see Section 10.3.2). The first word of each record is written by FOROTS and specifies the beginning of the binary record; this 36-bit value is called a type 1 Logical Segment Control Word (LSCW).

Each binary record can contain one or more type 2 LSCWs. The type 2 LSCW, written by FOROTS under certain conditions, is used to mark a record that spans internal buffer boundaries.

A third FOROTS-written word, the type 3 LSCW, is always written as the last 36-bit value in each BINARY record.

All data in a "BINARY" transfer remains equivalent in its external form with its internal representation.

IMAGE — Specifies an unformatted binary mode. Like the BINARY form of unformatted transfers, IMAGE specifies that data is transferred as 36-bit values, with the internal and external representation of the data remaining the same.

Unlike BINARY files, IMAGE files do not contain record information (LSCWs); they contain only the data transferred. IMAGE files can be backspaced if a record size is specified.

DUMP
TOPS-10 — Corresponds to TOPS-10 DUMP mode I/O. (See the *TOPS-10 Monitor Calls Manual*.) Record size is ignored. Character data cannot be written into or read from DUMP mode files. Note that there is little or no performance advantage to using DUMP, as FOROTS uses DUMP mode internally for all disk files.

NOTE

> For ASCII devices (line printer, plotter, terminal, industry magnetic tape), the MODE= specifier has no meaning and is ignored; both formatted and unformatted data transfers are legal (see Section 10.3.3).

Table 11-2 summarizes the different MODE arguments that are supported on different devices.

Table 11-2: DEVICE and MODE Cross-Table

	MODE=				
Device	'ASCII'	'LINED'	'BINARY'	'IMAGE'	(TOPS-10) 'DUMP'
Disk (sequential)	X	X	X	X	X
Disk (direct)	X		X	X	
DECtape	X	X	X	X	
Terminal	X				
Magtape	X	X	X	X	
Line Printer	X			X	
Card Reader	X		X	X	
Card Punch	X		X	X	
Paper Tape Reader	X		X	X	
Paper Tape Punch	X		X	X	

NAME
Specifier

11.3.21 NAME Specifier

The NAME specifier is used to specify a full file specification. You can use this specifier instead of the DEVICE, FILE, and/or DIRECTORY specifiers.

The form of the NAME specifier is:

NAME= Character expression

Examples of the NAME specifier are:

```
(TOPS-10):   NAME='DSK:FOO.BAR[10,34]'

(TOPS-20):   NAME='PS:<SMITH>BILLING.DAT'
```

The NAME specifier can not be given in DIALOG mode. Also, the OPEN statement cannot have both a DIALOG= specifier and a NAME specifier.

PADCHAR Specifier

11.3.22 PADCHAR Specifier

The PADCHAR specifier is used only with formatted output data transfer operations. PADCHAR enables you to specify a character that will be used to pad fixed-length formatted records, on output only, to their specified record length (see the RECL(RECORDSIZE) specifier, Section 11.3.27).

The form for the PADCHAR specifier is:

PADCHAR= A character expression in which the first character is used

The default pad character is space. The pad character is ignored if fixed-length records are not being used (that is, if the RECL(RECORDSIZE) specifier is absent), or if formatted I/O is not being done.

NOTE

To specify a null character for the pad character, you must use the function CHAR (see Chapter 13), since the compiler does not allow null character constants, for example:

```
OPEN (UNIT=1,PADCHAR=CHAR(0))
```

PARITY Specifier

11.3.23 PARITY Specifier

The PARITY specifier is only used for magnetic tape operations. It permits you to specify parity to be observed (odd or even) during the transfer of data.

The form of the PARITY specifier is:

PARITY = par

where:

par is a character expression having a value equal to 'ODD' or 'EVEN'.

PROTECTION (TOPS-10)
Specifier

11.3.24 PROTECTION Specifier (TOPS-10)

This option specifies a protection code to be assigned to the data file being transferred. The protection code is a 3-digit octal value indicating the level of access to the file.

Each of the three numbers in the protection code has a specific meaning. The number in the leftmost position designates the file owner's protection; the middle number designates the project member's protection; and the rightmost number designates the protection for all others users on the system. The system default for the file protection is assigned if a protection is not specified.

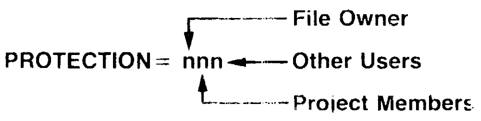
On TOPS-10, the form of the PROTECTION specifier is:

    PROTECTION= Integer expression

Figure 11-1 illustrates the TOPS-10 3-digit octal file protection code.

NOTE

> When setting the protection code for a file open for output, be sure not to set the protection such that the file is protected against writing by the program; if you should do this, the OPEN statement will fail.
>
> To protect the file against writing by the owner, you should give the PROTECTION specifier in the CLOSE statement.



File owner — An octal digit in the first position specifies the file access for the file owner. The file owner is the user whose programmer number matches the directory in which the file is contained.

Project Members — An octal digit in the second position specifies the file access for the project members. Project members are users whose project number matches the directory in which the file is contained.

Other Users — An octal value in the third position specifies the file access for all users other than the file owner or a project member.

Figure 11-1: TOPS-10 File Protection Number

Table 11-3 lists all the possible values for each field of the protection code. Refer to the TOPS-10 Monitor Calls Manual for more information.

NOTE

The File Daemon referred to in Table 11-3 provides extended file protection. The File Daemon allows any user to specify who can and who cannot access their files (if applicable). (Refer to the TOPS-10 Operating System Commands Manual.)

Table 11-3: TOPS-10 Protection Code Values

Octal Value	Meaning in Owner Field
0	The file owner can execute, read, append to, update, write, rename, or change the protection of the file. The File Daemon is not called on a protection failure.
1	The file owner can execute, read, append to, update, write, or rename the file. The File Daemon is not called on a protection failure.
2	The file owner can execute, read, append to, update, or write the file. The File Daemon is not called on a protection failure.
3	The file owner cannot access the file. However, the owner can use the RENAME monitor call to change the file protection. The File Daemon is not called on a protection failure.
4	The file owner can execute, read, append to, update, write, rename, or change the protection of the file. The monitor calls the File Daemon on a protection failure.
5	The file owner can execute, read, append to, update, or write the file. The monitor calls the File Daemon on a protection failure.
6	The file owner can execute or read the file. The monitor calls the File Daemon on a protection failure.
7	The file owner cannot access the file. However, the owner can use the RENAME monitor call to change the file protection. The monitor calls the File Daemon on a protection failure.

Octal Value	Meaning in Project Member and Other User Fields
0	The project member (or other) can execute, read, append to, update, write, rename, or change the protection of the file.
1	The project member (or other) can execute, read, append to, update, write, or rename the file.
2	The project member (or other) can execute, read, append to, update, or write the file.
3	The project member (or other) can execute, read, append to, or update the file.
4	The project member (or other) can execute, read, or append to the file.
5	The project member (or other) can execute or read the file.
6	The project member (or other) can only execute the file.
7	The project member (or other) has no access to the file.

In TOPS-10, PROTECTION can be an integer expression. If the argument is assigned a zero value or is not specified, the default protection code established for the TOPS-10 installation is used.

PROTECTION (TOPS-20)
Specifier

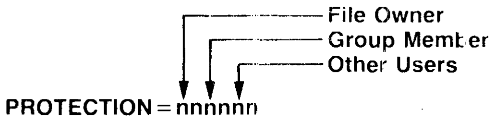
11.3.25 PROTECTION Specifier (TOPS-20)

The PROTECTION specifier enables you to designate an octal protection code for the file. The protection code is a 6-digit octal value that designates the access to the file of the owner, group members, and other system users.

On TOPS-20, the form of the PROTECTION specifier is:

```
     PROTECTION= Integer expression
```

Figure 11-2 illustrates a TOPS-20 protection code.



File Owner — The leftmost two digits designate the file access for the file owner. The file owner is the user who is connected to the directory in which the file resides.

Group Member — The middle two digits designate the file access for group members. Group members are established by the system administrator. A group membership enables a user to share files among other users in the same group.

Other Users — The rightmost two digits designate the file access for all users on the system, other than the file owner or a group member.

Figure 11-2: TOPS-20 Protection Number

Table 11-4 lists the possible protection values and their meanings in the TOPS-20 file protection code.

Table 11-4: TOPS-20 Protection Code Values

Octal Value	Meaning in Owner, Group Member, or Other User Fields
77	Permits full access to the file.
40	Permits read-only access to the file.
20	Permits write and delete access to the file.
10	Permits execute-only access to the file.
04	Permits append-only access to the file.
02	Permits listing of the file specification using the DIRECTORY command.
00	Permits no access to the file.

On TOPS-20, PROTECTION specifies a protection code to be assigned to the data file being transferred. The protection code determines the level of access that three classes of users have to the file.

PROTECTION takes an integer expression as an argument. If PROTECTION is assigned a zero value or is not specified, the default protection code established for your connected directory is used.

READONLY
Specifier

11.3.26 READONLY Specifier

The READONLY specifier is used to specify that the program will only read from the file. Output to the file is illegal and will cause an error at execution time.

The form of the READONLY specifier is:

```
READONLY
```

RECL
(RECORDSIZE)
Specifier

11.3.27 RECL (RECORDSIZE) Specifier

The RECL (or RECORDSIZE) specifier enables you to specify the number of characters or words in each record transferred. RECL is required for all files opened for direct access (ACCESS='DIRECT', 'RANDOM', or 'RANDIN'). (See Section 11.3.1.)

The form of the RECL specifier is:

```
RECL= Integer expression
```

For an ASCII transfer (MODE='ASCII' or 'LINED'), the value assigned to RECL specifies the number of characters in each record.

For output to disk or CORE-DUMP tape files, in addition to these characters, FOROTS adds a carriage return/line feed to each record, followed by enough null characters to fill the current word, so that records are word-aligned. RECL is enforced on output by padding short records with the padding character for formatted records. Long records are truncated.

FORTRAN enforces a specified RECL for all input operations. If the RECL specified in an OPEN statement is different from the actual size of the records, FOROTS reads the number of characters specified by RECL.

For input to disk or CORE-DUMP tape files, specifying a record size directs FOROTS to read records that are word-aligned. The calculation of the actual recordsize is the size specified, plus two for the carriage return/line feed, plus the number of nulls necessary to word-align the record.

For INDUSTRY tapes, with RECORDTYPE='FIXED', RECL specifies the exact number of characters in each record; there are no terminators or padding characters. For INDUSTRY tapes with RECORDTYPE='VARIABLE', RECL specifies the maximum record size in the file, excluding the RCW.

When the record is read, regardless of the contents of the record, it is interpreted as specified by the rules above; there are RECL characters of data, and the rest are ignored. No interpretation is done of the characters in the data part of the record. These characters appear in the FOROTS line buffer exactly as they appear in the file, including nulls and control characters.

In the case of MODE='LINED', the value of RECL excludes the five characters and tab in each line-sequence number.

In a binary transfer (MODE='BINARY', or 'IMAGE'), the value assigned to RECL specifies the number of 36-bit words in each record. For MODE='BINARY', the value in RECL excludes the LSCWs written by FOROTS.

NOTE

If MODE='DUMP' is specified, RECL is ignored.

RECORDTYPE
Specifier

11.3.28 RECORDTYPE Specifier

The RECORDTYPE specifier defines the format of the records in a magnetic tape file.

The form of the RECORDTYPE specifier is:

```
     RECORDTYPE = rtype
```

where:

```
     rtype      is a character expression
```

The possible values for RECORDTYPE are:

FIXED is not defined for 36-bit format (CORE-DUMP). For INDUSTRY magnetic tape, this is the standard ANSI "F" tape record format. Thus, the record data is written with no terminators or carriage control characters, one record directly after another, to form exactly fixed-length blocks. For this record format, a RECORDSIZE must be specified in the OPEN statement.

For this record format, all physical blocks on the tape are the same size except for the last block, which may be a short block.

STREAM For all tape formats, this record format specifies that, for formatted files, a standard stream record terminator (CRLF) is placed at the end of each record, and that a standard stream terminator is expected on input to delimit records. For unformatted files, the behavior is identical to that of MODE='IMAGE'.

For this record format, all physical blocks on the tape are the same size except for the last block, which may be a short block.

In 36-bit mode (CORE-DUMP), the bytes in the last word after the last character of data will be null. For all tape formats, if a RECORDSIZE is specified for a formatted file, the actual record read or written consists of the data, CRLF, and the number of nulls necessary to word-align the record.

VARIABLE is not defined for 36-bit tape format (CORE-DUMP). For INDUSTRY magnetic tape, this is the standard ANSI "D" tape record format. Thus, records are variable length, and the record data is written with no terminators or carriage control characters, preceded by a 4 byte record size (referred to as Record Control Word or RCW).

Records are variable length, not to exceed the block size specified in the OPEN statement. If a RECORDSIZE is specified in the OPEN statement, this is the maximum number of bytes specified in the 4-byte RCW that precedes each record. The record actually read or written is four characters less than specified in the RCW.

STATUS
(TYPE)
Specifier

11.3.29 STATUS (TYPE) Specifier

The STATUS (or TYPE) specifier lets you specify whether or not the file being opened must exist, or what to do with the opened file when it is closed.

The form of the STATUS specifier is:

```
STATUS = sta
```

where:

sta is a character expression whose value is equal to one of the following:

```
'EXPUNGE'
'OLD'
'NEW'
'SCRATCH'
'UNKNOWN'
'KEEP'
'DELETE'
```

The arguments to STATUS are:

EXPUNGE On TOPS-10, this specifies that the file is deleted when it is closed. On TOPS-20, this specifies that the file is deleted and expunged when it is closed.

> NOTE
>
> On TOPS-10, any delete also expunges a file from storage. On TOPS-20, a DELETE operation marks the file as deleted; an EXPUNGE operation immediately erases the file from storage.

OLD Specifies that the file being opened must already exist. If the file does not exist, an error results.

NEW On TOPS-10, STATUS='NEW' specifies that the file must not exist. If the file does exist, an error results. An error also occurs if you specify STATUS='NEW' with ACCESS='SEQIN','SEQUENTIAL','SEQINOUT' (to a read-only device), or 'RANDIN' (see Section 11.3.1).

On TOPS-20, the STATUS='NEW' specifier acts differently, depending on what you have in the directory before STATUS='NEW' is executed.

Also, the way you specify the file in the OPEN statement which contains the STATUS='NEW' specifier influences the way the STATUS='NEW' specifier operates. The following list describes the ways that this specifier can operate when used on TOPS-20.

1. If the file specified in the OPEN statement does not currently exist in the directory, and no generation number is specified, then the STATUS='NEW' specifier creates the specified file and gives it a generation number of 1.

2. If the file specified in the OPEN statement contains a name, extension, and generation number that does not exist, the specified file is used.

3. If the file specified in the OPEN or CLOSE statement contains a name, extension, and generation number that is exactly the same as an existing file in your directory, then STATUS='NEW' causes an error, and no file is created.

4. If you did not specify a generation number, but the file specified has the same name and extension as an existing file in your directory, then the file with the same name and extension and the next highest generation will be created.

SCRATCH Specifies that the file will be automatically deleted when the file is closed. STATUS='SCRATCH' implies DISPOSE='DELETE'.

A SCRATCH file is always given a name by FOROTS. The name of the file is inaccessible to the FORTRAN program.

If STATUS='SCRATCH' is used, you must not specify FILE, DIRECTORY, PROTECTION, or VERSION. If your program is writing a file with STATUS='SCRATCH', and the file is being written to disk, you can retain it by executing a CLOSE statement that renames the file to a specified name.

If a file is opened with STATUS='SCRATCH', the access must be ACCESS='SEQUENTIAL' (or ACCESS='RANDINOUT') or ACCESS='DIRECT' (or 'RANDOM') (see Section 11.3.1).

UNKNOWN Specifies that a file opened for an input operation must exist. When a file is opened for output with STATUS='UNKNOWN', if the file exists, it is superseded; if it does not exist, it is created.

UNKNOWN is the default for STATUS. This value is used unless you specify STATUS or unless the value of STATUS is otherwise determined by the ACCESS specifier.

KEEP Specifies that the file is not deleted. Specifying STATUS='KEEP' is equivalent to specifying DISPOSE='SAVE' and STATUS='UNKNOWN'.

DELETE On TOPS-10, specifies that the file will be erased when the file is closed.

On TOPS-20, specifies that the file will be deleted when the file is closed. The file is erased when a TOPS-20 EXPUNGE command is given. To undelete a deleted file, use the TOPS-20 UNDELETE command.

TAPEFORMAT
Specifier

11.3.30 TAPEFORMAT SPECIFIER

The TAPEFORMAT specifier defines the physical format of the magnetic tape.

The form of the TAPEFORMAT specifier is:

TAPEFORMAT = fmode

where:

tmode is a character expression having a value equal to 'CORE-DUMP' or 'INDUSTRY'.

The values for the TAPEFORMAT specifier are:

CORE-DUMP specifies the "DIGITAL-compatible" tape format, which is 36-bits stored in five frames on a 9-track tape. The SET TAPE RECORDSIZE (TOPS-20) or SET BLOCKSIZE (TOPS-10) command is interpreted to be the number of 36-bit words in the magnetic tape blocks on the tape, and is used if no BLOCKSIZE specifier is given in the OPEN statement. If a BLOCKSIZE specifier is given in the OPEN statement, it is interpreted to be the number of 36-bit words for both formatted and unformatted files.

INDUSTRY specifies characters are read or written in standard industry tape format, one character per tape frame.

UNIT
Specifier
(required)

11.3.31 UNIT Specifier (Required)

The UNIT specifier defines the FORTRAN logical unit number to be used. FORTRAN devices are identified by assigned decimal numbers within the range 0-99 (see Table 10-3). UNIT must be an integer expression.

The form of the UNIT specifier is:

```
UNIT= Integer expression
```

If the unit specifier is the first specifier given in the OPEN statement, the keyword UNIT= is optional. For example the following statement opens a file on unit 20:

```
OPEN (20,FILE='MYFILE')
```

NOTE

The FORTRAN standard logical unit assignments are described in Section 10.4.3.1. Although the range of logical unit numbers shown in Table 10-3 is 0-99, the range of UNIT numbers is an installation-defined parameter.

VERSION (TOPS-10)
Specifier

11.3.32 VERSION Specifier (TOPS-10)

Use the VERSION specifier for disk operations only; it enables you to assign a 12-digit octal version number to an output file.

The form of the VERSION specifier is:

```
     VERSION=Integer expression
```

11.4 CLOSE STATEMENT

The CLOSE statement disassociates an active file from a logical unit and releases the memory occupied by I/O buffers and other unit-related data. The CLOSE statement can also change some of the file characteristics that were assigned during the OPEN, such as the name, protection, directory, and disposition of the file.

Once a CLOSE statement has been executed, you must use another OPEN statement (or implicit OPEN) to regain access to the closed file.

The form of the CLOSE statement is:

```
     CLOSE (closelist)
```

where:

closelist — is a list of CLOSE statement specifiers. This list must contain the UNIT specifier (see Section 11.5.9) and can optionally contain other specifiers.

11.4.1 Implicit CLOSE

An implicit CLOSE occurs when FOROTS automatically closes a logical unit without execution of a CLOSE statement. This can happen when a program terminates, or when you execute an OPEN for a unit that is already connected to another file.

11.5 CLOSE STATEMENT SPECIFIERS

All of the CLOSE statement specifiers are optional, except the UNIT specifier which is required. Some CLOSE statement specifiers have the same formats as the corresponding specifiers in the OPEN statement.

Table 11-5 summarizes the specifiers in the CLOSE statement, and the type of value required by each. A section number is provided to refer to detailed descriptions of each specifier.

Table 11–5: Summary of CLOSE Statement Specifiers and Arguments

Argument	Possible Value	Section
DEVICE=	Character expression	11.5.1
DIALOG		11.5.2
DIALOG=	Character expression	11.5.3
DIRECTORY=	Character expression	11.5.1
DISPOSE=	Character expression with one of the following values: 'SAVE', 'DELETE', 'PRINT', 'KEEP', 'LIST', 'PUNCH', 'EXPUNGE'	11.5.4
ERR=	Statement number	11.5.5
FILE=	Character expression	11.5.1
IOSTAT=	Integer variable or integer array element	11.5.6
LIMIT=	Integer expression	11.5.7
NAME=	Character expression	11.5.1
PROTECTION=	Integer expression	11.5.1
STATUS= TYPE=	Character expression with one of the following values: 'KEEP', 'DELETE', 'EXPUNGE'	11.5.8
UNIT=	Integer expression	11.5.9

NOTE

For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric array name as the value of each of the following specifiers:

DIALOG=
DIRECTORY
NAME

When a numeric array name is used, FOROTS assumes that it contains a string of characters terminated by a null character.

In addition, you can specify a numeric variable as the value of the DEVICE and FILE specifiers. If the variable is single precision, FOROTS assumes that it contains 5 characters; if it is double precision, FOROTS assumes that is contains 10 characters.

The use of numeric array names and numeric variables in place of character variables is a nonstandard feature.

DEVICE, DIRECTORY, FILE, NAME, and PROTECTION Specifiers

11.5.1 DEVICE, DIRECTORY, FILE, NAME, and PROTECTION Specifiers

The CLOSE statement file identification specifiers can be used when you want to rename the output file when it is closed. Their formats are the same as the corresponding specifiers in the OPEN statement.

If any of these specifiers are given in the CLOSE statement, the file is renamed when it is closed. If some, but not all of the file identification parameters are specified on a CLOSE statement, only the specified parameters are changed when the file is renamed.

Example:

```
     OPEN(20,ACCESS='SEQOUT',FILE='FOO.BAR')
         .
         .
         .
     CLOSE(20,FILE='NEWFIL')
```

The above sequence renames the output file to DKSB:NEWFIL.BAR.

Refer to the following sections under the OPEN statement:

FILE - see Section 11.3.15
NAME - see Section 11.3.21
DEVICE - see Section 11.3.8
DIRECTORY (TOPS-10) - see Section 11.3.11
DIRECTORY (TOPS-20) - see Section 11.3.12
PROTECTION (TOPS-10) - see Section 11.3.24
PROTECTION (TOPS-20) - see Section 11.3.25

DIALOG Specifier

11.5.2 DIALOG Specifier

The DIALOG specifier enables you to type in additional CLOSE specifiers when the CLOSE statement is actually executed.

If the DIALOG specifier is found in the CLOSE list, then each time the CLOSE statement is executed, FOROTS suspends execution and prompts the terminal with an asterisk.

The form of the DIALOG specifier is:

```
     DIALOG
```

You can respond to the asterisk prompt by entering a file specification, DIALOG switches (see below), or a file specification followed by DIALOG switches. The file specification can be a full file specification including the device, directory name, and so on.

If you enter a file specification that is different from the file specification currently assigned to the file, FOROTS RENAMES the file after it is closed to the new name.

NOTE

> A DIALOG switch is any CLOSE specifier (except DIALOG, DIALOG=, UNIT, NAME, FILE, IOSTAT, or ERR) preceded by a slash (/).

DIALOG = Specifier

11.5.3 DIALOG= Specifier

The DIALOG= specifier enables you to include all or a portion of the CLOSE specifiers in a character expression. The contents of the character expression are interpreted as if you had given the DIALOG specifier (see above); and, when the asterisk prompt was given, you had typed in the same string as is contained in the character expression, followed by a carriage return.

The form of the DIALOG= specifier is:

```
     DIALOG= Character expression
```

Example:

```
     CLOSE (UNIT=20,ERR=10,DIALOG='/DISPOSE:DELETE')
```

When DIALOG= is given in the CLOSE list, it is processed after all other specifiers except DIALOG.

DISPOSE Specifier

11.5.4 DISPOSE Specifier

The DISPOSE specifier enables you to specify an action to occur when the file is closed.

The form of the DISPOSE specifier is:

```
     DISPOSE = dis
```

where:

[illegible] is a character expression having a value equal to one of the following:

```
'KEEP'
'SAVE'
'DELETE'
'EXPUNGE'
'PRINT'
'LIST'
'PUNCH'
```

The DISPOSE specifier must have one of the following values:

KEEP — Specifies that the file is to be left on the connected unit. DISPOSE='KEEP' is the default. You can not specify DISPOSE='KEEP' if in the corresponding OPEN statement you specified STATUS='SCRATCH' (see Section 11.3.29).

SAVE — Same as KEEP.

DELETE — Specifies on TOPS-10 that, if the device is either a DECtape or disk, delete the file; otherwise, take no action.

On TOPS-20, if the device involved is a disk, delete the file; otherwise, take no action.

EXPUNGE — On TOPS-10, same as DELETE. On TOPS-20, if the device involved is a disk, expunge the file; otherwise, take no action.

PRINT — Specifies that the file will be printed and kept (the file will not be kept if you specify the CLOSE statement STATUS='DELETE' or 'EXPUNGE'). The file must be on disk.

LIST — Specifies that the file will be printed, deleted, and expunged (the file will not be deleted if you specify the CLOSE statement STATUS='KEEP'). The file must be on disk.

PUNCH — Specifies that the file will be punched on the paper tape punch and kept. The file must be on disk.

NOTE

The value of the CLOSE statement DISPOSE specifier supersedes the value of the OPEN statement DISPOSE specifier and the OPEN statement STATUS specifier if STATUS='EXPUNGE', 'KEEP', and 'DELETE'.

ERR Specifier

11.5.5 ERR Specifier

The ERR specifier enables you to designate a statement label of an executable statement, in the current program unit, to which control passes if an error occurs during the execution of an I/O statement.

The form of the ERR specifier is:

```
     ERR= s
```

where:

s is the number of an executable statement to which program control passes if an error occurs during the execution of the statement in which the ERR specifier is included.

The ERR specifier works the same way when it is given in a CLOSE as it does when given in an OPEN statement (see Section 11.3.14).

IOSTAT Specifier

11.5.6 IOSTAT Specifier

The IOSTAT specifier identifies an integer variable that is used to store the I/O status code during the execution of a statement.

The form of the IOSTAT specifier is:

```
     IOSTAT= Integer variable or integer array element
```

LIMIT Specifier

11.5.7 LIMIT Specifier

The LIMIT specifier designates the number of output units (such as pages) for spooled print or punch requests, which result from using DISPOSE='PRINT', DISPOSE='PUNCH', or DISPOSE='LIST' (see Section 11.5.4).

The form of the LIMIT specifier is:

```
     LIMIT= Integer expression
```

STATUS Specifier

11.5.8 STATUS Specifier

The STATUS specifier tells FOROTS what to do with a file when it is closed. In the CLOSE statement, STATUS values are a subset of the DISPOSE specifier (see Section 11.5.4) values.

NOTE

> The ANSI-77 standard does not have DISPOSE and only allows STATUS='KEEP' or STATUS='DELETE'.

The form of the STATUS specifier is:

```
     STATUS = sta
```

where:

sta is a character expression having a value equal to one of the following:

```
     'KEEP'
     'DELETE'
     'EXPUNGE'
```

The arguments to STATUS are:

KEEP Specifies that the file is not deleted.

DELETE On TOPS-10, specifies that the file is deleted.

On TOPS-20, specifies that the file is marked for deletion when the file is closed. The file is erased when a TOPS-20 EXPUNGE command is given. To undelete a deleted file, use the TOPS-20 UNDELETE command.

EXPUNGE On TOPS-10, the same as delete. On TOPS-20, this specifies that the file is deleted and expunged.

NOTE

> The value of the CLOSE statement STATUS specifier supersedes the value of the OPEN statement DISPOSE specifier and OPEN statement STATUS specifier if STATUS='EXPUNGE', 'KEEP', and 'DELETE'.

UNIT
Specifier
(Required)

11.5.9 UNIT Specifier (Required)

The UNIT specifier tells FOROTS which logical unit number is to be closed. This specifier corresponds to the UNIT specifier in the OPEN statement (see Section 11.3.31) and to the UNIT specifiers in the data transfer statements (see Section 10.4.3).

The form of the UNIT specifier is:

```
     UNIT= Integer expression
```

If the unit specifier is the first specifier given in the CLOSE statement, the keyword UNIT= is optional. For example, to close a file on unit 20, you can use the following command:

```
     CLOSE (20)
```

11.6 OPEN AND CLOSE STATEMENT EXAMPLES

The following are examples of OPEN and CLOSE statements:

```
     OPEN (UNIT=1,DEVICE='DSK',ACCESS='SEQIN',MODE='BINARY')
```

causes a disk file named FOR01.DAT (since no FILE= option was specified) to be opened on unit 1 for sequential input in binary mode.

```
     OPEN (UNIT=3,DEVICE='DSK',FILE='PAYROL.DAT',
     1      ACCESS='RANDOM',MODE='ASCII',RECORDSIZE= 80,
     2      ASSOCIATEVARIABLE=I,ERR=240)
```

Causes a disk file named PAYROL.DAT to be opened on unit 3 for random I/O operations in ASCII mode. The records in PAYROL.DAT are 80 characters long; the ASSOCIATE VARIABLE for this file is I. If an error occurs during the execution of this OPEN statement, the OPEN terminates, and control is transferred to the statement labeled 240.

```
     CLOSE (UNIT=3,DISPOSE='DELETE')
```

The above statement causes the file associated with unit 3 to be closed and deleted.

11.7 INQUIRE STATEMENT

The INQUIRE statement inquires about specific properties of a file name or of a logical unit number on which a file might be opened. The INQUIRE statement has two forms: one inquires by file, and the other inquires by unit.

11.7.1 INQUIRE by File

An INQUIRE by file is an INQUIRE statement containing the following:

- A FILE= keyword
- An associated file specification
- No UNIT= keyword

It is used to obtain information about a file based on the file name. INQUIRE by file can be used to get information on the following files:

- Files that are "connected"; meaning files for which an OPEN statement has been executed or for which a data transfer statement has been executed.
- Files that are not "connected."

INQUIRE by file has the following form:

```
     INQUIRE (FILE=fi[,flist])
```

where:

fi   is a character expression whose value specifies the name of the file to be inquired about.

flist is a list that can contain at most one of each of the inquiry specifiers (see Section 11.7.3).

INQUIRE by file may be used any time during the execution of a program. It can be used before a file is opened to find out about the existence of the file, or after the file is opened to find out other attributes of the file. It can also be used to find the unit number on which the file is opened. If the same file is opened on more than one unit, the smallest number on which the file is opened is returned.

The determination of whether a file specified in an INQUIRE statement is opened on a unit is the following:

1. The file specification given in the INQUIRE statement is used to lookup the file.
2. If the file exists, the file specification, expanded with the physical device name and generation (TOPS-20 only), is compared with the file specification for each open unit, in ascending order, until there is an exact string match.

3. If the file does not exist, the specification given in the INQUIRE statement (with a default of DSK: added if necessary for the device name), is compared with the file specification for each open unit, in ascending order, until there is an exact string match. Note that this match will only be successful for 'deferred' OPEN files, since non-deferred OPEN files are always established in the specified directory immediately. Therefore, the file exists (see item 2 above).

NOTE

> If a file exists, INQUIRE by file will not generally match the file with a unit for which a 'deferred' OPEN has been done, since the file specification for the unit has not been expanded. For example, the file's logical device name has not been replaced by a physical device name.

(See Section 18.8 for information on FOROTS and INQUIRE by file.)

11.7.2 INQUIRE by Unit

INQUIRE by unit is an INQUIRE statement containing a UNIT= keyword and no FILE= keyword. It is used to find out information about the file that may be "connected" to the specified unit.

INQUIRE by unit has the following form:

```
     INQUIRE ([UNIT=]u,ulist)
```

where:

u is the number of the logical unit to be inquired about. The unit need not exist, nor need it be connected to a file. If the unit is connected to a file, the inquiry encompasses both the connection and the file.

ulist is a list that can contain at most one of each of the inquiry specifiers (see Section 11.7.3).

If the optional UNIT= keyword if omitted, u must be the first item in the list.

INQUIRE by unit can be used at any time during the execution of a program. It can be used before a file is opened to find out if there is another file open on the unit, or after the file is opened to find out other attributes of the file.

11.7.3 Inquiry Specifiers

The specifiers described in the following sections may be used in either form of the INQUIRE statement.

11.7.3.1 ACCESS Specifier - The ACCESS specifier has the following form:

```
     ACCESS = acc
```

where:

acc is a character variable, array element, or substring reference. It is assigned the value 'SEQUENTIAL' if the file is connected for sequential access, or 'DIRECT' if the file is connected for direct access. If there is no connection, acc is 'UNKNOWN'.

11.7.3.2 BLANK Specifier - The BLANK specifier has the following form:

```
BLANK = blk
```

where:

blk is a character variable, array element, or substring reference. It is assigned the value 'NULL' if the file was last opened with BLANK='NULL', and is assigned the value 'ZERO' if the file was opened with BLANK='ZERO'. If the file is not open, blk is 'UNKNOWN'.

11.7.3.3 CARRIAGECONTROL Specifier - The CARRIAGECONTROL Specifier has the following form:

```
CARRIAGECONTROL = cc
```

where:

cc is a character variable, array element, or substring reference. It is assigned the following values:

1. 'FORTRAN' if the file has the FORTRAN carriage-control attribute

2. 'LIST' if the file has the implied carriage-control attribute

3. 'NONE' if the file has no carriage-control attribute

4. 'TRANSLATED' if the file has FORTRAN carriage-control characters being translated directly into vertical motion characters.

5. UNKNOWN if the CARRIAGECONTROL value cannot be determined, or the file is not open.

11.7.3.4 DIRECT Specifier - The DIRECT specifier has the following form:

```
DIRECT = dir
```

where:

dir is a character variable, array element, or substring reference. It is assigned the following values:

1. 'YES' if DIRECT is an allowed access method for the file

2. 'NO' if DIRECT is not an allowed access method for the file

3. 'UNKNOWN' if the processor is unable to determine whether DIRECT is an allowed access method

11.7.3.5 **ERR Specifier** - The ERR specifier has the following form:

```
ERR = s
```

where:

s is the label of an executable statement. ERR is a control specifier; if an error occurs during execution of the INQUIRE statement, control is transferred to the statement whose label is s.

11.7.3.6 **EXIST Specifier** - The EXIST specifier has the following form:

```
EXIST = ex
```

where:

ex is a logical variable or logical array element. It is assigned the value .TRUE. if the specified file or unit exists, and the value .FALSE. if the specified file or unit does not exist.

11.7.3.7 **FORM Specifier** - The FORM specifier has the following form:

```
FORM = fm
```

where:

fm is a character variable, array element, or substring reference. It is assigned the value 'FORMATTED' if the file is connected for formatted I/O, and 'UNFORMATTED' if the file is connected for unformatted I/O. If there is no connection, fm is 'UNKNOWN'.

11.7.3.8 **FORMATTED Specifier** - The FORMATTED specifier has the following form:

```
FORMATTED = fmd
```

where:

fmd is a character variable, array element, or substring reference. It is assigned the value 'YES' if formatted is an allowed form for the file. It is assigned the value 'NO' if formatted is not an allowed form of the file, and the value 'UNKNOWN' if the form cannot be determined.

11.7.3.9 IOSTAT Specifier - The IOSTAT specifier has the following form:

```
IOSTAT = ios
```

where:

ios is an integer variable or integer array element. It is assigned a processor-dependent positive integer value if an error occurs during execution of the INQUIRE statement, or assigned the value zero if there is no error condition.

11.7.3.10 NAME Specifier - The NAME specifier has the following form:

```
NAME = nme
```

where:

nme is a character variable, array element, or substring reference. It is assigned the name of the file being inquired about.

The value assigned to nme is not necessarily identical to the value specified with FILE=. For example, the value that the processor returns may contain a directory name or generation number (TOPS-20 only). However, the value that is assigned is always valid for use with FILE= in an OPEN statement.

NOTE

FILE and NAME are synonyms when used with the OPEN statement, but not when used with the INQUIRE statement.

For INQUIRE by unit, FOROTS returns the full, expanded file specification if any of the following is true:

- If there is a file open on the specified unit with STATUS other than UNKNOWN or SCRATCH.
- If there is a file open on the specified unit with ACCESS other than SEQUENTIAL.
- An I/O transfer statement has been executed using the specified unit and the unit has not been closed.

FOROTS returns the string given in the OPEN for NAME= with defaults applied if both of the following are true:

- If the file is open on the specified unit as a result of an OPEN statement in which STATUS='UNKNOWN' and ACCESS='SEQUENTIAL' are specified or implied.
- No I/O transfer statement has been executed using the specified unit.

If STATUS='SCRATCH', FOROTS returns blanks for NAME=.

If there has been no OPEN statement, and no I/O transfer statement has been executed using the specified unit, FOROTS returns for NAME= the default file specification for that unit.

For INQUIRE by file, FOROTS returns the full, expanded file specification if the file exists in the specified directory. If the file does not yet exist on the specified directory, but has been opened by a 'deferred' OPEN, FOROTS returns the string given in the INQUIRE statement, with defaults applied for the device and generation number (TOPS-20 only). Otherwise, blanks are returned.

11.7.3.11 NAMED Specifier - The NAMED specifier has the following form:

NAMED = nmd

where:

nmd is a logical variable or logical array element. It is assigned the value .TRUE. if the specified file has a name, and the value .FALSE. if the file does not have a name.

11.7.3.12 NEXTREC Specifier - The NEXTREC specifier has the following form:

NEXTREC = nr

where:

nr is an integer variable or integer array element. It is assigned an integer value that is one more than the last record number read or written on the specified direct access file. If no records have been read or written, the value of nr is one. If the file is not connected for direct access, or if the position is indeterminate because of an error condition, nr is zero.

11.7.3.13 NUMBER Specifier - The NUMBER specifier has the following form:

NUMBER = num

where:

num is an integer variable or integer array element. It is assigned the number of a logical unit currently connected to the specified file. If there is no logical unit connected to the file, num is not defined. If more than one unit is connected to the file, the smallest unit number is returned.

11.7.3.14 OPENED Specifier - The OPENED specifier has the following form:

```
OPENED = od
```

where:

od is a logical variable or logical array element. It is assigned the value .TRUE. if the specified file is opened on a unit or if the specified unit is opened; it is assigned the value .FALSE. if the file or unit is not open.

11.7.3.15 RECL (RECORDSIZE) Specifier - The RECL (RECORDSIZE) specifier has the following form:

```
RECL = rcl
```

where:

rcl is an integer variable or integer array element. If the file (or unit) is opened, rcl is the record length for fixed-length record files. In all other cases, rcl is zero. If the file is opened for formatted I/O, rcl is expressed in characters, and in words if the file is unformatted.

11.7.3.16 RECORDTYPE Specifier - The RECORDTYPE specifier has the following form:

```
RECORDTYPE = rtype
```

where:

rtype is a character variable, array element, or substring reference. It is assigned the value 'FIXED' if the file has fixed-length records, assigned the value 'VARIABLE' if the file has variable-length records, and assigned the value 'STREAM' if the file is a stream file (default for disk and magnetic tape). If the processor cannot determine the record type, rtype is assigned the value 'UNKNOWN'.

11.7.3.17 SEQUENTIAL Specifier - The SEQUENTIAL specifier has the following form:

```
SEQUENTIAL = seq
```

where:

seq is a character variable, array element, or substring reference. It is assigned the following values:

1. 'YES' if SEQUENTIAL is an allowed access method for the specified file

2. 'NO' if SEQUENTIAL is not an allowed access method

3. 'UNKNOWN' if the processor cannot determine whether SEQUENTIAL is an allowed access method

11.7.3.18 UNFORMATTED Specifier - The UNFORMATTED specifier has the following form:

```
UNFORMATTED = unf
```

where:

unf is a character variable, array element, or substring reference. It is assigned the value 'YES' if unformatted is an allowed form for the file. It is assigned the value 'NO' if unformatted is not an allowed form of the file, and the value 'UNKNOWN' if the form cannot be determined.

Table 11-6 summarizes the form and use of the FORTRAN device control statements.

Table 11-6: Summary of FORTRAN Device Control Statements

Statement Form	Section
FIND (UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios]) FIND (un'rn[,ERR=s][,IOSTAT=ios])	11.8.1
REWIND un REWIND (UNIT=un[,ERR=s][,IOSTAT=ios]) REWIND (un[,ERR=s][,IOSTAT=ios])	11.8.2
UNLOAD un UNLOAD (UNIT=un[,ERR=s][,IOSTAT=ios]) UNLOAD (un[,ERR=s][,IOSTAT=ios])	11.8.3
BACKSPACE un BACKSPACE (UNIT=un[,ERR=s][,IOSTAT=ios]) BACKSPACE (un[,ERR=s][,IOSTAT=ios])	11.8.4

ENDFILE un
ENDFILE (UNIT=un[,ERR=s][,IOSTAT=ios])
ENDFILE (un[,ERR=s][,IOSTAT=ios])
	11.8.5

SKIPRECORD un SKIPRECORD (UNIT=un[,END=s][,ERR=s][,IOSTAT=ios]) SKIPRECORD (un[,END=s][,ERR=s][,IOSTAT=ios])	11.8.6
SKIPFILE un SKIPFILE (UNIT=un[,ERR=s][,IOSTAT=ios]) SKIPFILE (un[,ERR=s][,IOSTAT=ios])	11.8.7
BACKFILE un BACKFILE (UNIT=un[,ERR=s][,IOSTAT=ios]) BACKFILE (un[,ERR=s][,IOSTAT=ios])	11.8.8

11.8 DEVICE CONTROL STATEMENTS

Device control statements enable you to position external devices. For example, when performing data transfers with magnetic tape, you use device control statements to position the tape. The device control statements may be used for both formatted and unformatted files.

The following list contains all of the device control statements, and the section in which each statement is described.

1. FIND (Section 11.8.1)

2. REWIND (Section 11.8.2)

3. UNLOAD (Section 11.8.3)

4. BACKSPACE (Section 11.8.4)

5. ENDFILE (Section 11.8.5)

6. SKIPRECORD (Section 11.8.6)

7. SKIPFILE (Section 11.8.7)

8. BACKFILE (Section 11.8.8)

NOTE

> The results of the BACKSPACE and SKIPRECORD statements are unpredictable when used on list-directed and NAMELIST formatted data.

The general form of every device control statement is:

```
keyword un

or

keyword (UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])

or

keyword (un[,END=s][,ERR=s][,IOSTAT=ios])
```

where:

keyword — is the statement name.

un — is the FORTRAN logical unit number. Table 10-3 lists the default logical unit numbers. If you enclose the logical unit specification in parentheses, you can include the keyword portion (UNIT=) of the logical unit specification. The keyword form of the unit specifier makes the specification positionally independent in the parenthetical list.

END=s — is the optional end-of-file specifier (see Section 10.4.6).

ERR=s is the optional error specifier (see Section 10.4.7).

IOSTAT=ios is the optional I/O error status specifier (see Section 10.4.8).

The operations performed by the device control statements are usually used only for magnetic tape devices (MTA). In FORTRAN, however, the device control operations are simulated for disk devices.

FIND Statement

11.8.1 FIND Statement

In earlier versions of FORTRAN-10/20, the FIND statement could be used during direct-access READ/WRITE operations to reduce the time necessary to do data transfers and to locate records in a direct-access file. For FORTRAN-10/20 Version 10, the FIND statement has no effect, except for setting the current record number and, therefore, the associate variable.

The first form of the FIND statement is:

```
FIND (UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])
```

The second form of the FIND statement operates in the same way as the first form. The difference between the two forms is in the way that the unit number and record number are specified.

The second form of this statement is:

```
FIND(un'rn[,ERR=s][,IOSTAT=ios])
```

In this form, the unit number and record number do not contain the keywords UNIT= and REC=. Instead, the unit number is specified first, followed by a single quote ('), and finally the record number.

The following example demonstrates how the FIND statement is used:

```
FIND(UNIT=1,REC=100)
```

This statement positions the file so that the next READ statement reads record 100.

REWIND Statement

11.8.2 REWIND Statement

The REWIND statement, used for disk files, causes a specified file to be positioned at its initial point. For magnetic tape, execution of the REWIND statement positions the magnetic tape at its initial point.

The forms of the REWIND statement are:

```
REWIND un

or

REWIND (UNIT=un[,ERR=s][,IOSTAT=ios])

or

REWIND (un[,ERR=s][,IOSTAT=ios])
```

where:

un is the logical unit number of the device on which the REWIND is being performed. Table 10-3 lists the default logical unit numbers.

REWIND is illegal for files opened with ACCESS='APPEND'.

UNLOAD Statement

11.8.3 UNLOAD Statement

The UNLOAD statement rewinds and unloads the magnetic tape associated with the specified unit.

The forms of the UNLOAD statement are:

```
UNLOAD un

or

UNLOAD (UNIT=un[,ERR=s][,IOSTAT=ios])

or

UNLOAD (un[,ERR=s][,IOSTAT=ios])
```

where:

un is the logical unit number of the device on which the UNLOAD is being performec. Table 10-3 lists the default logical unit numbers.

BACKSPACE Statement

11.8.4 BACKSPACE Statement

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. If the preceding record is an ENDFILE record (see Section 11.8.5), the file is positioned before the ENDFILE record.

The BACKSPACE statement cannot be used for direct-access files, append-access files, or files that are formatted with list-directed or NAMELIST-statement formatting.

The forms of the BACKSPACE statement are:

```
BACKSPACE un

or

BACKSPACE (UNIT=un[,ERR=s][,IOSTAT=ios])

or

BACKSPACE (un[,ERR=s][,IOSTAT=ios])
```

where:

un is the logical unit number of the device on which the BACKSPACE is being performed. Table 10-3 lists the default logical unit numbers.

ENDFILE Statement

11.8.5 ENDFILE Statement

The ENDFILE statement closes the file on the specified unit. On magnetic tape, an 'ENDFILE record' is written and is then positioned after the end-of-file mark.

For disk, the file is closed and then positioned at the end of the file, and an end-of-file status is set. This status is the equivalent of the file being positioned after an 'ENDFILE record'.

Thus, for both disk and magnetic tape, a BACKSPACE operation given after an end file operation positions the file after the last data record (that is, before the physical (or virtual) 'ENDFILE record').

The ENDFILE statement can be used only with sequential access files.

The forms of the ENDFILE statement are:

```
ENDFILE un

or

ENDFILE (UNIT=un[,ERR=s][,IOSTAT=ios])

or

ENDFILE (un[,ERR=s][,IOSTAT=ios])
```

where:

un is the logical unit number of the device on which the ENDFILE is being performed. Table 10-3 lists the default logical unit numbers.

SKIPRECORD Statement

11.8.6 SKIPRECORD Statement

The SKIPRECORD statement skips the record immediately following the current (last accessed) record. If the SKIPRECORD statement is executed prior to accessing any records, then the first record in the file is skipped. You cannot use SKIPRECORD on direct-access files.

The forms of the SKIPRECORD statement are:

```
SKIPRECORD un

or

SKIPRECORD(UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])

or

SKIPRECORD(un[,END=s][,ERR=s][,IOSTAT=ios])
```

where:

un is the logical unit number of the device on which the SKIPRECORD is being performed. Table 10-3 lists the default logical unit numbers.

SKIPFILE
Statement

11.8.7 SKIPFILE Statement

This statement is used only for magnetic tape operations. Unless an end-of-file has been encountered, the SKIPFILE statement advances to the beginning of the next file. If an end-of-file has been encountered, SKIPFILE skips the next file. If the number of SKIPFILE executions exceeds the number of files available to be skipped, an error occurs.

The forms of the SKIPFILE statement are:

```
SKIPFILE un

or

SKIPFILE (UNIT=un[,ERR=s][,IOSTAT=ios])

or

SKIPFILE (un[,ERR=s][,IOSTAT=ios])
```

where:

un — is the logical unit number of the device on which the SKIPFILE is being performed. Table 10-3 lists the default logical unit numbers.

BACKFILE
Statement

11.8.8 BACKFILE Statement

This statement is used only for magnetic tape operations. If an end-of-file has been encountered, the BACKFILE statement positions to the start of the file whose end-of-file was detected. Otherwise, the BACKFILE statement positions to the start of the file that precedes the current (last accessed) file.

The forms of the BACKFILE statement are:

```
     BACKFILE un

     or

     BACKFILE (UNIT=un[,ERR=s][,IOSTAT=ios])

     or

     BACKFILE (un[,ERR=s][,IOSTAT=s])
```

where:

un is the logical unit number of the device on which the BACKFILE is being performed. Table 10-3 lists the default logical unit numbers.

NOTE

On a magnetic tape with multiple files, the position of the tape after an ENDFILE record of one file is equivalent to the position at the beginning of the next file.

CHAPTER 12

FORMATTED DATA TRANSFERS

Data transfers can be either formatted or unformatted. When the internal (memory) representation of the data is translated to a different external (peripheral storage) representation during a data transfer, that data transfer is considered formatted.

Conversely, when the internal and external representations of the data are the same, that data transfer is considered unformatted.

A formatted data transfer involves editing of data as it is transferred to and from memory. FORTRAN provides you with three ways for specifying how the data is formatted during a formatted data transfer. These are:

1. FORMAT-Statement Formatting

2. List-Directed Formatting

3. NAMELIST-Statement Formatting

Of the three types, FORMAT-statement formatting provides you with the most control over how the data is formatted. Section 12.1 describes FORMAT-statement formatting.

List-directed formatting means that the formatting is controlled by the data types of the I/C list elements. Section 12.5 describes list-directed formatting.

NAMELIST-statement formatting is the third method for formatting the data; the formatting is controlled by the data types of the namelist elements. In this form, the I/O list is defined in a NAMELIST statement and referenced by the data transfer statement. Section 12.6 describes NAMELIST-statement formatting.

FORMAT-Statement Formatting

12.1 FORMAT-STATEMENT FORMATTING

A FORMAT statement directs the editing of data during its transfer between internal and external storage. Every formatted (FORMAT statement) data transfer statement contains a reference to one of the following:

1. A line containing a FORMAT statement with a format list

2. A numeric array containing a format list

3. A character expression containing a format list

4. An integer, real, or logical variable that has been assigned a FORMAT statement number with an ASSIGN statement

The format list is made up of format specifiers.

During execution of a formatted data transfer statement, items in the I/O list are associated with specifiers in the referenced format list. The specifiers dictate how the various data items are formatted.

Section 12.1.1 describes how to create a format list in a FORMAT statement; Section 12.1.2 describes how to create a format specification as a character expression. Section 12.1.3 describes how to create a numeric array that contains a format list. Section 12.1.4 describes how to specify a FORMAT statement using an ASSIGNed variable.

12.1.1 Specifying a Format List in a FORMAT Statement

The general form of a FORMAT statement is:

```
n      FORMAT fs
```

where:

n — is the required statement number. This number, referenced in the control-information list of an I/O statement (see Section 10.4.2) provides the association between the data transfer statement and the FORMAT statement.

fs — is a format specification. The form of a format specification is:

```
([format list])
```

where:

format list — is a list of items which may take any of the following forms:

```
[r] ed

  or

ned

  or

[r] (fl)
```

where:

r — is a nonzero, unsigned, integer constant called a repeat specification.

ed — is a repeatable edit descriptor (see Section 12.2.1).

ned — is a nonrepeatable edit descriptor (see Section 12.2.2).

fl — is a nonempty format list.

The only placement restrictions for FORMAT statements are that they follow PROGRAM, FUNCTION, SUBPROGRAM, or BLOCK DATA statements, and that they precede the END statement.

The following example illustrates FORMAT-statement formatting. The FMT specifier in the WRITE statement references the label of FORMAT statement 101. This FORMAT statement contains a list of edit descriptors (X, I, and F) that dictate the formatting of the data in I/O list (variables J, Y, and Z).

```
          J=2
          Y=3.0
          Z=5.1
          WRITE(UNIT=5,FMT=101)J,Y,Z
    101   FORMAT(1X,I,F,F)
```

12.1.2 Specifying a Format Specification as a Character Expression

You can store format specifications in character variables, character arrays, character array elements, character substrings, or character expressions.

A character format specification must be of the form described in Section 12.1.1. Note that the form begins with a left parenthesis and ends with a right parenthesis. Character data may follow the right parenthesis that ends the format specification, with no effect on the format specification. Blank characters may precede the format specification.

If the format identifier is a character array name, the format specification may be contained in more than the first element of the array. (A character array format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering (see Section 4.3.3).)

However, if a character array element name is specified as a format identifier, the length of the format specification must not exceed the length of the array element.

The following example shows the same format specification used in the examples in Section 12.1.1. This time, however, instead of referencing the format specification by statement number, or referencing the name of a numeric array, the data transfer statement references the name of the character variable in which the format specification is contained.

```
      J=2
      Y=3.0
      Z=5.1
      CHARACTER FORNAM*10
      FORNAM = '(1X,I,F,F)'
      WRITE(UNIT=5,FMT=FORNAM)J,Y,Z
```

12.1.3 Specifying a Format Specification in a Numeric Array

An alternative to using FORMAT statements is to store the format specification in a numeric array.

The format specifications are associated with a data transfer statement by referencing the array name containing the format specification, instead of a statement label of a FORMAT statement.

The following example shows the same format specification used in the example in Section 12.1.1. This time, however, instead of referencing the format specification by statement number, the data transfer statement references the name of the numeric array in which the format specification is contained.

```
      INTEGER FORNAM(2)
      FORNAM(1)='(1X,I'
      FORNAM(2)=',F,F)'
      J=2
      Y=3.0
      Z=5.1
      WRITE(UNIT=5,FMT=FORNAM)J,Y,Z
      END
```

In the above example the format specification is stored in both words of array FORNAM. This is because the format contains ten characters: the first five are in FORNAM(1); and the last five are in FORNAM(2).

NOTE

> When storing a format specification in an array, always include the outer most parentheses enclosing the format specifiers. Note that the word FORMAT should not be included in the string.

12.1.4 Specifying a FORMAT Statement Using an ASSIGNed Variable

Integer, real, or logical variables that have been ASSIGNed FORMAT statement numbers can be used as format specifiers. (See Section 8.3 for information on the ASSIGN (statement label) assignment statement.)

The variable is assigned a statement number by an ASSIGN statement. The format specifier references the variable that refers to the statement number it has been assigned.

The following example shows the same format specification used in the examples in Sections 12.1.1, 12.1.2, and 12.1.3. This time, however, instead of referencing the format specification by statement number, the data transfer statement references a variable that has been assigned a statement number by an ASSIGN statement.

```
          ASSIGN 101 TO IFORMT
          WRITE(UNIT=5,FMT=IFORMT)J,Y,Z
     101  FORMAT(1X,I,F,F)
```

12.1.5 The Ordering and Interpretation of Format List Items

For standard conforming programs, all items within the format list should be separated by commas, with the exception of the following cases:

1. Between a P edit descriptor and an immediately following F, E, D, or G edit descriptor (See Section 12.4.11)

2. Before or after a slash edit descriptor (See Section 12.4.5)

3. Before or after a colon edit descriptor (See Section 12.4.6)

In FORTRAN-10/20, the use of commas to delimit format edit descriptors within a format list is optional as long as no ambiguity exists. For example,

```
     FORMAT (3X,A2)
```

can be written as

```
     FORMAT (3XA2)
```

But the specification

```
     FORMAT (I22I5)
```

is ambiguous, since it can represent

```
     FORMAT (I22,I5)     or     FORMAT (I,22I5)
```

and requires the comma to eliminate ambiguity.

FORMAT-Statement Edit Descriptors

12.2 EDIT DESCRIPTORS

Edit descriptors within the format list describe the manner of editing performed on the data being transferred.

For example, when you transfer integers from a file to memory, you use an I edit descriptor. When the data transfer statement is executed, an item in the I/O list is associated with the I edit descriptor in the format list, and the following results:

1. Before being stored in memory, the data is converted to an internal integer format by the I edit descriptor in the format list.

2. The memory location in which the data is stored is identified by the I/O list element.

The following sample program demonstrates how an integer is read from the terminal (external device) into the memory location identified in the I/O list of the ACCEPT statement.

```
        PROGRAM FORMAT

        TYPE *,'Please enter a two digit number:'
        ACCEPT 101,K
101     FORMAT(I2)

        TYPE 102,K
102     FORMAT(1X,I5)
        END
```

The sample output below shows what happens when the user executes the above program. The user enters 78 in response to the ACCEPT statement. This causes the integer value to be stored in the variable K according to the I edit descriptor in FORMAT statement 101. Then the type statement causes the value of variable K to be printed at the terminal according to the I edit descriptor in FORMAT statement 102.

```
EXECUTE TEST
LINK: Loading
[LNKXCT FORMAT execution]

Please enter a two digit number:
78
   78
CPU Time 0.1   Elapsed Time 8.7
```

The I edit descriptor is an example of a repeatable edit descriptor. FORTRAN has two types of edit descriptors: repeatable (Section 12.2.1) and nonrepeatable (Section 12.2.2). The third type of item that appears in a format list is the carriage-control specifier (Section 12.2.3).

Repeatable Edit Descriptors

12.2.1 Repeatable Edit Descriptors

A repeatable edit descriptor may be preceded by an optional, unsigned, nonzero, integer constant that specifies a repeat count. This integer is called a repeat specification.

Using a repeat specification in an edit descriptor gives you a shorthand way to specify multiple fields with a single specification. For example, without using the repeat specification, if you wanted to specify four fields, each of which contain an integer value that is six characters long, you might construct the following FORMAT statement:

```
     101    FORMAT (1X,I6,I6,I6,I6)
```

If you use the repeat specification, however, you need only specify the edit descriptor and field width a single time, as follows:

```
     101    FORMAT (1X,4I6)
```

These two FORMAT statements are equivalent.

Table 12-1 lists the repeatable edit descriptors. Each descriptor listed in the table is shown in its complete form. The key at the bottom of Table 12-1 describes all the optional elements in each edit descriptor. The right-most column of Table 12-1 references the section in which each edit descriptor is discussed.

Table 12–1: Repeatable FORTRAN Edit Descriptors

Edit Descriptor	Descriptor Type	Refer to:
[r]I[w[.m]]	Integer	Section 12.4.11.1
[r]F[w.d]	Floating Point	Section 12.4.11.2
[r]E[w.d[Ee]]	Scientific Notation	Section 12.4.11.3
[r]D[w.d[Ee]]	Scientific Notation	Section 12.4.11.3
[r]G[w.d[Ee]]	General Conversion	Section 12.4.11.4
F,E,D,G (Two successive)	Complex	Section 12.4.11.5
[r]O[w[.m]]	Octal	Section 12.4.11.6
[r]Z[w[.m]]	Hexadecimal	Section 12.4.11.7
[r]L[w]	Logical	Section 12.4.12
[r]A[w]	Character or Hollerith	Section 12.4.13
[r]R[w]	Hollerith	Section 12.4.14

Key:

r is a nonzero, unsigned, integer constant called a repeat specification.

w is a nonzero, unsigned, integer constant which is equal to the total number of characters in the numeric field being described. The numeric edit descriptors are described in Section 12.4.11.

.m is an unsigned, integer constant which specifies the minimum number of digits to be output to the field being described. If necessary, leading zeros are output. The value of m must not exceed the value of w.

If m is zero and the value of the internal data item is zero, the output field consists of only blank characters, regardless of the sign control in effect.

.d is a nonzero, unsigned, integer constant which specifies the total number of digits to the right of the decimal point in the numeric field being described. If .d is specified, w must also be specified. The maximum value is 63 digits.

e is a nonzero, unsigned, integer constant which is equal to the total number of digits in the exponent field of the numeric field being described. The maximum value is 15 digits.

Nonrepeatable Edit Descriptors

12.2.2 Nonrepeatable Edit Descriptors

A nonrepeatable edit descriptor can not be preceded by a repeat specification. The nonrepeatable edit descriptors provide a variety of editing possibilities, such as positioning within a record, including character constants in a FORMAT statement, and delimiting records within a single format descriptor.

Table 12-2 lists the nonrepeatable edit descriptors. The format, function, and section number where each descriptor is discussed are listed in the table.

Table 12–2: Nonrepeatable FORTRAN Edit Descriptors

Edit Descriptor	Function	Refer to:
'h1...hn'	Character Data	Section 12.4.1
nHh	Hollerith Data	Section 12.4.2
Tc TLc TRc	In-Record Positioning	Section 12.4.3.1
|n|X	In-Record Positioning	Section 12.4.3.2
$	(Dollar sign) Prevents record from terminating with END OF LINE	Section 12.4.4
/	(Slash) Record Delimiter	Section 12.4.5
:	(Colon) Format-Control Termination	Section 12.4.6
S SP SS	Plus sign control for output of positive numeric fields	Section 12.4.7
kP	Scaling Factor for Numeric Fields	Section 12.4.8
BN BZ	Specifies the handling of blanks during the input of Numeric Fields	Section 12.4.9
Q	Input Only Descriptor — returns the number of characters left in the current record.	Section 12.4.10

Key:

n is a nonzero, unsigned, integer constant which is equal to a number of spaces (X descriptor) or the total number of characters (H descriptor).

h is a character capable of representation by the processor. This type of character is described in Appendix B.

c is a nonzero, unsigned, integer constant which is equal to a number of characters within a record relative to the current position.

k is an optionally signed integer constant which declares the scaling factor for the field being described.

Carriage-Control Specifiers

12.2.3 Carriage-Control Specifiers

In a data output transfer, the first character of each record can be used for carriage control. A carriage-control specifier dictates the action of the printing mechanism on output devices. For example, carriage-control specifiers determine the vertical spacing for line-printer output.

NOTE

> The CARRIAGECONTROL specifier of the OPEN statement enables you to decide how the first character of each record is treated. Depending on the value of the CARRIAGECONTROL specifier, the first character can be:
>
> 1. Replaced with the appropriate printer-control character(s).
>
> 2. Disregarded as a carriage-control character and, instead, be transferred as part of the record.
>
> For more information on the CARRIAGECONTROL specifier, see Section 11.3.6

The carriage-control specifier may be written as a character constant. The following example shows the blank carriage-control character in a FORMAT statement:

```
              WRITE(5,101)

     101      FORMAT(' ','This is a string')

              END
```

When this example is executed, the string in the format list is printed on unit 5, the terminal, as follows:

```
     This is a string
```

If you omit the carriage-control specifier from a data output transfer format list, FOROTS interprets the first character to be output to the record as the carriage-control character. Using the example above, if we omit the blank specifier, FOROTS assumes that the first character encountered (in this case, the "T" in "This") is the carriage-control character. Executing this example, after removing the carriage-control specifier, causes the first character to be stripped from the character constant. Thus, the output at the terminal is:

```
     his is a string
```

The carriage-control characters are summarized in Table 12-3. The $ (dollar sign) output edit descriptor modifies the action of the carriage-control specifier (see Section 12.4.4).

Table 12–3: Carriage–Control Specifiers

Specifier	Format List Form	Printer Character	Octal Value	Effect on Carriage Control
blank	' '	LF	012	Skip to next line (form feed after 60 lines on printer).
plus	'+'			Suppress line feed; overprint the line.
zero	'0'	LF,LF	012,012	Skip a line.
one*	'1'	FF	014	Form feed to top of next page.
two*	'2'	DLE	020	Space to next half page.
three*	'3'	VT	013	Space to next one-third of a page.
minus	'–'	LF,LF,LF	012,012,012	Skip two lines.
asterisk*	'*'	DC3	023	Skip to next line; suppress form feed. (Continous print)
period*	'.'	DC2	022	Triple space, with a form feed after every 20 lines printed.
comma*	','	DC1	021	Double space, with a form feed after every 30 lines printed.
slash*	'/'	DC4	024	Space to next one-sixth of a page.

* Indicates carriage-control specifiers for which the effect on carriage control is device dependent. The effect described is for a line printer with a standard form setup.

Note — This table assumes a standard form setup for your line printer (or other output device).

I/O List & FORMAT List Interaction

12.3 INTERACTION OF INPUT/OUTPUT LIST AND FORMAT LIST

This section describes how the I/O list and the format list interact during a data transfer.

12.3.1 General Description

Format control is initiated by execution of a formatted data transfer statement. The actions performed by format control depend on the interaction of the edit descriptors in the format specification and the I/O list elements in the data transfer statement.

The following example shows how the I/O list elements in a data transfer statement interact with the edit descriptors in the format list in a simple data transfer.

```
          READ (5,100) N,X,Y
    100   FORMAT (I5,F12.0,F10.0)
```

In this example, the I/O list is

```
N,X,Y
```

and the format specification is

```
(I5,F12.0,F10.0)
```

The variables in the I/O list and the specifiers in the format are matched up as follows:

N	I5
X	F12.0
Y	F10.0

A formatted data transfer statement matches elements of the I/O list and specifiers in the format specification. The matching proceeds from left to right, one I/O list element to one repeatable edit descriptor.

In the above example, there are three elements in the I/O list and three format specifiers. However, the interactions can be more complicated than those in the example. A format specifier can be preceded by a repeat count, in which case it corresponds to more than one element in the I/O list. Also, an element of the I/O list can be an array name, in which case it can correspond to more than one format specifier. The number of elements in the I/O list and in the format specification do not have to be the same.

Table 12-4 details what happens in these more complex cases.

Table 12–4: Record, Format List, and I/O List Interaction

1. **Record ends**
   Format specification continues
   I/O list continues

Action:

The transfer continues as if the record was extended with blanks.

Example:

```
        READ (5,10) A,B,C
  10    FORMAT (3F10.0)
```

Record contains:

```
  40    10
```

Resulting values:

```
  A = 40.0
  B = 10.0
  C =  0.0
```

Note that this situation is not applicable to an output transfer.

2. **I/O list ends**
   Format specification continues

Action:

The format scan continues until it encounters a repeatable edit descriptor, a colon, or until the rightmost right parenthesis of the format is reached.

Nonrepeatable edit descriptors up to and including the first colon, if present, are processed if they are encountered during the scan.

Example:

```
        A = 12
        B = 123
        C = 1234
        D = 12345

        WRITE (5,20) A,B,C
  20    FORMAT (' A=',F3.0,', B=',F4.0,', C=',F5.0,',
        1    D=',F6.0)
```

Resulting output:

```
  A=12., B=123., C=1234., D=
```

Note that the ', D=' descriptor was processed. The format scan does not terminate until it encounters a descriptor which requires an I/O list element, and there is no I/O list element to supply. However, the colon edit descriptor will cause the format scan to stop if there is nothing left in the I/O list (see Section 12.4.6).

3. **Format specification ends**
   I/O list continues

Action:

A new record is started. The format scan continues, starting at the beginning of the last complete parenthesized group within the format specification. If there is no parenthesized group within the format, the format is restarted from the beginning (see Section 12.3 4).

Example:

```
        A=123.
        B=7654.125
        C=1.6125

        WRITE (5,10) A,B,C
  10    FORMAT (F15.6)
```

Table 12–4: Record, Format List, and I/O List Interaction (Cont.)

Resulting output:

```
 123.000000
7654.125000
   1.612500
```

4. I/O list ends
Format specification ends

Action:

On input, if there are any characters remaining in the record, they are ignored. On output, the record is simply terminated without any extra characters added.

Example:

```
      READ (5,10) GAMMA
10    FORMAT (F5.2)
```

Record contains:

12.34 value of GAMMA

Resulting values:

GAMMA = 12.34

The extra data (the comment) in the input record is ignored.

The execution of a formatted I/O statement proceeds by matching I/O list elements and FORMAT edit descriptors. The edit descriptors I, O, Z, R, F, E, D, G, L, A, and Q each correspond to one element of the I/O list. No I/O list element corresponds to H, X, P, T, :, $, S, SP, SS, BN, BZ, or apostrophe edit descriptors. If one of these descriptors is encountered, it is executed and the format scan continues.

12.3.2 Formatted Input

A formatted input statement begins by reading a record from the specified unit. The format is scanned from left to right. X, P, T, BN, BZ, and / edit descriptors are executed as they are encountered. If an I, O, Z, R, F, E, D, G, L, A, or Q descriptor is encountered, data is read into the corresponding I/O list element as specified by the edit descriptor.

If the I/O list contains no more elements, execution of the READ statement ends. Additional records will be read from the specified unit when a slash occurs in the format, or when the last right parenthesis of the format is reached, and I/O list elements remain to be filled.

When an input record is terminated by a slash or by the end of the format, any data left in the input record is discarded. If the input record is exhausted before the data transfers are completed, the remainder of the transfer is completed as if the record were extended with blanks.

12.3.3 Formatted Output

A formatted output statement begins by scanning the format. The H, X, P, T, BN, BZ, :, /, $, S, SS, SP, and apostrophe edit descriptors are executed as they are encountered. If an I, O, Z, R, F, E, D, G, L, or A descriptor is encountered, data is translated from the corresponding I/O list element, as specified by the edit descriptor, and placed in the output record.

If the I/O list contains no more elements, the output record is written to the specified unit, and execution of the WRITE statement ends. Additional records will be written to the specified unit when a slash occurs in the format, or when the last right parenthesis is reached, and I/O list elements remain to be transferred.

12.3.4 Embedded Format Specifications

Format specifications may contain embedded format specifications with optional repeat specifications. If a repeat specification is used, the entire format specification that it precedes is scanned the specified number of times during the I/O transfer. In the example:

```
      WRITE (1,100) A,B,C,D,E,F,G,H,I,J
  100 FORMAT (F10.2,4(I5,1X,I3),I8)
```

the variable A is matched with the format item F10.2. Then, the variable B is matched with I5, variable C with I3, variable D with I5, and so on for four iterations of the embedded format specification. Finally, the variable J is matched with the format item I8.

If no repeat specification is used preceding an embedded format specification, a repeat count of 1 is implied.

When the last right parenthesis of the format is reached, and more I/O list elements remain to be transferred, a new record is started and format scanning continues. The scanning continues at the beginning of the format specification whose right parenthesis is the next to last right parenthesis in the format. If there are no embedded format specifications, format scanning continues at the beginning of the format.

For example:

```
      DIMENSION A(100)
      INTEGER CASE
      CASE=33

      WRITE (1,100)CASE,(A(I),I=1,100)
  100 FORMAT ('THIS IS CASE ',I6,//,4(1X,F10.5))
```

After A(1) through A(4) are written, a new line is started and format scanning continues at the beginning of the embedded format specification. Thus, A(5) through A(8) are written; a new record is started, and so forth.

The output file would appear as follows:

```
     THIS IS CASE 33

     A(1)       A(2)       A(3)       A(4)
     A(5)       A(6)       A(7)       A(8)
     A(9)       A(10)      A(11)      A(12)
                     .
                     .
                     .
```

Example:

```
     DIMENSION A(5),DAT1(100),B(4,100),DAT2(100)
     WRITE (1,100) CASE,(A(K),K=1,5),(DAT1(J),(B(I,J),I=1,4),
          1  DAT2(J),J=1,100)
     100  FORMAT ('CASE',I5,//,5(1X,F10.3),/,(F10.3,4(3X,F15.5)
          1 ,1X,F10.3))
```

In this example, after A(1) through A(5), DAT1(1), B(1,1) through B(4,1), and DAT2(1) are written, a new record is started, and format scanning begins. The format scanning begins at the embedded format specification following the '/' (the specification whose right parenthesis is the next to last right parenthesis).

The output file would appear as follows:

```
     CASE    33

     A(1)       A(2)       A(3)       A(4)       A(5)
     DAT1(1)    B(1,1)     B(2,1)     B(3,1)     B(4,1)     DAT2(1)
     DAT1(2)    B(1,2)     B(2,2)     B(3,2)     B(4,2)     DAT2(2)
     DAT1(3)    B(1,3)     B(2,3)     B(3,3)     B(4,3)     DAT2(3)
                                .
                                .
                                .
```

12.4 FORMAT EDITING

Tables 12-1 and 12-2 describe forms of all the FORMAT edit descriptors. The edit descriptors enable you to specify the form of a record and to specify the editing of the data as it is transferred.

The edit descriptors are described according to the character used to accomplish a particular modification to the data or record in which the data are stored.

APOSTROPHE (')
Editing

12.4.1 Apostrophe (') Editing

The apostrophe (') edit descriptor (single-quote) enables you to include a character constant in a format list.

The form of the apostrophe edit descriptor is:

```
'h1...hn'
```

where:

'h1...hn' is a character constant.

To include an apostrophe as part of the character constant, you must use two successive apostrophes.

This descriptor is only used for output; the characters enclosed by the apostrophes are written.

Example:

```
      TYPE 10
10    FORMAT (' That''s the way!')
```

will output

```
That's the way!
```

H
Editing

12.4.2 H Editing

The H edit descriptor (also called the Hollerith descriptor) enables you to include character strings in a format list.

The form of the H edit descriptor is:

```
nHh1...hn
```

where:

n is a nonzero, unsigned, integer constant that indicates the total number of ASCII characters included in the string.

h1...hn is a string of ASCII characters. (The ASCII character set is described in Appendix B.)

You may transmit alphanumeric data directly from the FORMAT statement using either the H or apostrophe specifiers.

This descriptor is only used for output; the n characters that follow the H are written. For example, you can use the following statement sequence to print the words PROGRAM COMPLETE on the printer:

```
    PRINT 101
101 FORMAT (17Hb̸PROGRAMb̸COMPLETE)
```

The result of apostrophe editing is the same as Hollerith editing. For example, you may use the descriptors:

```
     101 FORMAT (17HƀPROGRAMƀCOMPLETE)
```

and

```
     101 FORMAT ('ƀPROGRAMƀCOMPLETE')
```

in the same manner.

Apostrophes can appear anywhere within a Hollerith edit descriptor without having to be represented by two apostrophes. However, if the H edit descriptor occurs within a character constant, the apostrophe is written as two apostrophes, which are counted as one character.

POSITIONAL Editing

12.4.3 Positional Editing

The positional edit descriptors specify the position at which the next character will be transmitted to or from the record. The positional edit descriptors are: T, TL, TR, and X.

The T edit descriptor specifies the character position within a record where the next character will be transmitted (see Section 12.4.3.1).

The TL and TR descriptors specify the number of character positions to the left or right, respectively, of the current position for the character position of the next character (see Section 12.4.3.1).

The X descriptor specifies the number of character positions to the right of the current position for the character position of the next character (see Section 12.4.3.2).

NOTE

> On output, a record is initially filled with blanks. Therefore, fields skipped by the positional editing descriptors will be blank-filled. However, the output record length is determined by actual output. Merely specifying a positional editing descriptor with no output will not change the record size. Thus, the record written with:
>
> ```
>      FORMAT (I6,50X,T10,I3)
> ```
>
> will have a record length of 13 characters, since no output was done after the 50X.

Examples:

The statement sequence:

```
       PRINT 2
     2 FORMAT (T50,'BLACK',T30,'WHITE')
```

causes the following line to be printed:

```
          WHITE                        BLACK
          ↑                            ↑
     (print position 30)      (print position 50)
```

The statement sequence:

```
1 FORMAT (T35,'MONTH')
  READ (2,1)
```

causes the first 34 characters of the input data associated with logical unit 2 to be skipped, and the next five characters to replace the characters M, O, N, T, and H in storage.

If an input record containing:

```
ABCb̸b̸b̸XYZ
```

is read with the format specification:

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read in that order.

You can use the field descriptor nX to introduce blanks into output records or to skip characters of input records. The letter X specifies the operation, and n is a positive integer that specifies the number of character positions to be either made blanks (output) or skipped (input).

The statement:

```
FORMAT (5Hb̸STEP,I5,10X,2HY=,F7.3)
```

may be used to print the line:

```
STEPb̸b̸b̸28b̸b̸b̸b̸b̸b̸b̸b̸b̸b̸Y=b̸-3.872
```

T, TL, and TR Editing

12.4.3.1 T, TL, and TR Editing - The T edit descriptor specifies that the transmission of the next character to or from a record is to occur at the specified character position.

The form of the T descriptor is:

```
Tc
```

where:

c is a positive, unsigned, integer constant that indicates the character position to or from which the next character will be transferred.

For example:

```
FORMAT (T20,I5,T8,I2)
```

specifies that the characters read or written for the I5 descriptor will start in character position 20, and the characters read or written for the I2 descriptor will start in character position 8. For output to carriage-control devices (line printer and terminal), Tn specifies that n-1 will be the next character written, since the character position 1 is the carriage-control character position.

The TL edit descriptor specifies that the transmission of the next character to or from the record is to occur at a position which is a specified number of positions backward from the current position.

The form of the TL edit descriptor is:

```
TLc
```

where:

c is a positive, unsigned, integer constant that indicates the character position, c positions backward from the current position, to or from which the next character will be transferred. If c would cause transmission to start at a position before the beginning of the current record, transmission will start instead at position one.

For example:

```
FORMAT (I5,T13,A5,TL10,I2)
```

specifies that the characters read or written for the A5 descriptor will start at character position 13, and that the characters read or written for the I2 descriptor at will start at character position 8.

The TR edit descriptor specifies that the transmission of the next character to or from a record will occur at a position that is a specified number of positions forward from the current position. The function of this form is identical to that of the X descriptor (see Section 12.4.3.2).

The form of the TR edit descriptor is:

```
TRc
```

where:

c is a positive, unsigned, integer constant that indicates the character position, c positions forward from the current position, from which the next character will be transferred.

Example using all three types of T descriptor:

```
        TYPE 10
10      FORMAT(' 2345678901234567890123456789012345678901234567890')

        TYPE 20
20      FORMAT(T29,'BLACK',T10,'WHITE')

        TYPE 30
30      FORMAT(T10,'9012',TR5,'8901')
        TYPE 10

        TYPE 40
40      FORMAT(T20,'<SECOND>',TL10,'>FIRST<')

        END
```

```
EXECUTE TEST2.FOR
FORTRAN: TEST2
MAIN.
LINK:   Loading
[LNKXCT TEST2 execution]

234567890123456789012345678901234567890
        WHITE                  BLACK
        9012     8901
234567890123456789012345678901234567890
                >FIRST<ND>

CPU time 0.1  Elapsed time 0.5
```

In FORMAT 20, 'BLACK' is written, then 'WHITE' is written to the left of it. In FORMAT 30, five positions are skipped between the two character strings being printed. In FORMAT 40, '<SECOND>' is written, the format goes back ten positions and writes '>FIRST<' over the previously written character string.

X
Editing

12.4.3.2 X Editing - The X edit descriptor specifies that the transmission of characters to or from a record will occur a specified number of characters forward from the current position.

The form of the X edit descriptor is:

```
[n]X
```

where:

n   is an optional, unsigned, positive, integer constant that indicates the number of characters forward from the current position, at which the next character will be transmitted. The default value is 1.

Example:

```
     TYPE 10
10   FORMAT(' 1234567890123456789012345678901234567890')

     TYPE 20
20   FORMAT(' A WORD OR TWO',10X,'OR THREE')

     END
```

```
EXECUTE TEST3.FOR
FORTRAN: TEST3
MAIN.
LINK: Loading
[LNKXCT TEST3 execution]

1234567890123456789012345678901234567890
A WORD OR TWO          OR THREE

CPU time 0.2    Elapsed time 2.1
```

In this example, ten positions are skipped between the printing of 'A WORD OR TWO' and 'OR THREE'.

$
(DOLLAR SIGN)
Editing

12.4.4 $ (Dollar Sign) Editing

The $ (dollar sign) output edit descriptor suppresses all carriage control at the end of the current record (for CARRIAGECONTROL='LIST') or at the beginning of the next record (for CARRIAGECONTROL='FORTRAN' or 'TRANSLATED').

This descriptor is used for interactive I/O; it leaves the terminal position at the end of the text so that a response will follow the output on the same line.

Example:

```
          WRITE (5,10)
          READ (5,*) N
     10   FORMAT (' Number of samples: ', $)
          WRITE (5,20)
          READ (5,*) X
     20   FORMAT (' Mean value:        ', $)
          END
```

If the user enters 100 for N and 1.23 for X, executing the program will produce the typescript:

```
          Number of samples: 100
          Mean value:        1.23
```

The $ edit descriptor can be used to append the output of several statements into a single line. For example:

```
          DO 10 I = 1,10
     10   WRITE (5,20) I
     20   FORMAT (1X,I3,$)
          WRITE (5,20)
          END
```

will produce one line of output:

```
     1  2   3   4   5   6   7   8   9   10
```

The $ edit descriptor is ignored for input.

12.4.5 / (Slash) Editing

The / (slash) edit descriptor indicates the end of data transfer for a record. Two consecutive slashes indicate the transmission of an empty record.

On input to a file connected for sequential access, the remaining portion of the current record is skipped, and the file is positioned at the beginning of the next record. This new record becomes the current record. On output to a file connected for sequential access, the current record is terminated, and a new record is created, which becomes the current and last record of the file.

A record that contains no characters may be written. Also, an entire record may be skipped on input.

If the file is connected for direct access, the record number is increased by one, and the file is positioned at the beginning of the record that has that record number. This record becomes the current record.

The following statements will write a record with no characters:

```
          WRITE (1,100)
     100  FORMAT(/)
```

To handle a group of I/O records where different records have different field descriptors, use a slash to indicate a new record. For example, the statement

```
     FORMAT (3O8/I5,2F8.4)
```

is equivalent to

```
     FORMAT (3O8)
```

for the first record, and

```
     FORMAT (I5,2F8.4)
```

for the second record.

You may omit separating commas when you use a slash. When n slashes appear at the beginning or end of a format, n blank records will be written on output or skipped on input. When n slashes appear in the middle of a format, n-1 blank records are written on output or n-1 records are skipped on input.

: (COLON)
Editing

12.4.6 : (Colon) Editing

The : (colon) edit descriptor terminates format control if there are no more items in the I/O list of the corresponding data transfer statement. The colon edit descriptor has no effect if there are any items left in the I/O list of the corresponding data transfer statement.

Example:

```
           X1 = 100
           X2 = 200
           X3 = 300

           TYPE 10, X1,X2
      10   FORMAT(F6.1,F6.1,:,'THIS SHOULD NOT PRINT',F6.1)

           TYPE 20,X1,X2,X3
      20   FORMAT(F6.1,F6.1,:,' BUT THIS SHOULD ',F6.1)
           END

      EXECUTE TEST4.FOR
      FORTRAN: TEST4
      MAIN.
      LINK:   Loading
      [LNKXCT TEST4 execution]

      100.0 200.0
      100.0 200.0 BUT THIS SHOULD  300.0
      CPU time 0.1   Elapsed time 1.5
```

S, SP, and SS
Editing

12.4.7 S, SP,and SS Editing

The S, SP, and SS edit descriptors control the output of the optional plus sign in numeric output fields. These descriptors are used as follows:

S indicates that the system-defined action is taken. The system-defined specification for Digital FORTRAN is SS.

SP indicates that the plus sign is printed in all positive numeric output fields.

SS indicates that the plus sign is not printed in positive numeric output fields. This descriptor is the default.

The S, SP, and SS edit descriptors affect only I, F, E, D, and G editing during the execution of a data transfer output statement; these edit descriptors have no effect during input transfers. These descriptors stay in effect until the end of the I/O transfer or until another S, SP, or SS is encountered.

Example:

```
          X1 = 100
          X2 = 200
          X3 = 300
          X4 = 400

          TYPE 10, X1,X2,X3,X4
     10   FORMAT(' ',S,F7.1, SP,F7.1, SS,F7.1, SP,F7.1)
          END

     EXECUTE TEST5.FOR
     FORTRAN: TEST5
     MAIN.
     LINK:    Loading
     [LNKXCT TEST5 execution]

       100.0 +200.0  300.0 +400.0
     CPU time 0.1   Elapsed time 0.2
```

In this example, X2 and X4 have plus signs because of the SP descriptors in front of the F descriptors that correspond to them.

P
Editing

12.4.8 P Editing

The P edit descriptor is used to specify a scale factor for F, E, D, and G edit descriptors.

The form of the P edit descriptor is:

```
     kP
```

where:

k	is an optionally signed integer constant representing the scale factor.

If a scale factor is not specified in a format list, a scale factor of zero is assumed. Once a scale factor is specified in a format list, that scale factor remains in effect for all F, E, D, and G edit descriptors until a new scale factor is specified, or to the end of the execution of the current I/O statement. Scale factors have no effect on I, Z, and O edit descriptors.

The scale factor affects the F, E, D, and G data transfers as follows:

On input:

If there is an exponent field, the scale factor k has no effect. If there is no exponent field on the number read in, the number is multiplied by 10**(-k) before being assigned to the input variable.

On output:

The basic real constant part of the quantity, in E and D editing only, is multiplied by 10**k and the exponent is reduced by k. For G editing, the scale factor has no effect unless the magnitude of the data item to be edited is outside of the range that permits the use of F editing. If G editing is specified, and the magnitude of the data item to be edited is such that E editing is required, the scale factor has the same effect as with E output editing (see Section 12.4.11.3).

The comma is optional between a P edit descriptor and immediately following F, E, D, or G edit descriptors.

For example, assume the data involved is the real number 26.451; the edit descriptor

```
     F8.3
```

produces the external field

```
     b̸b̸26.451
```

The addition of the scale factor of -1P, as in

```
     FORMAT (-1P,F8.3)
```

produces the external field

```
     b̸b̸b̸2.645
```

When you add a scale factor to D, E, and G (external field not a decimal fixed-point) edit descriptors, the scale factor multiplies the number by the specified power of ten, and the exponent is changed accordingly.

In input operations, type F (and type G, if the external field is decimal fixed-point) conversions are the only ones affected by scale factors.

When you add a scale factor to a D or E edit descriptor, it specifies a power of 10 so that the external form of the number has its mantissa multiplied by the specified power of 10; its exponent is adjusted accordingly.

For example, assume the data involved is the real number 12.49; the edit descriptor

```
     E11.3
```

produces the external field

```
     b̸b̸0.125E+02
```

The addition of the scale factor 2P, as in

```
     FORMAT (2P,E11.3)
```

produces the external field

```
b̸b̸12.49E+00
```

With a scale factor of zero, the number of significant digits printed by a format of the form:

Ew.d

or

Dw.d

is the number of digits to the right of the decimal point.

For a negative scale factor nP, for -d<n<0, there will be ABS(n) leading zeros and d-ABS(n) significant digits after the decimal point (for a total of d digits after the decimal point). If n≤-d, there will be d insignificant digits (zeros) to the right of the decimal point.

If the scale factor nP is positive, for 0<n<d+2 there will be n significant digits to the left of the decimal point and d-n+1 significant digits to the right of the decimal point (for a total of d+1 significant digits). If n>d+2, there will be d+1 significant digits and n-d-1 insignificant trailing zeros on the left of the decimal point.

If the data to be printed is 12.493, these formats produce results as follows:

FORMAT	OUTPUT	SIGNIFICANT DIGITS	REASON
E15.3	b̸b̸b̸b̸b̸b̸0.125E+02	3	n=0
1PE15.3	b̸b̸b̸b̸b̸b̸1.249E+01	4	n<d+2
-1PE15.3	b̸b̸b̸b̸b̸b̸0.012E+03	2	-d<n
2PE15.3	b̸b̸b̸b̸b̸b̸12.49E+00	4	n<d+2
-3PE15.3	b̸b̸b̸b̸b̸b̸0.000E+05	0	n≤-d
4PE15.3	b̸b̸b̸b̸b̸b̸1249.E-02	4	n<d+2
6PE15.3	b̸b̸b̸b̸124900.E-04	4	n≥d+2

Example:

```
        TYPE 10
10      FORMAT(' Type in a real number')

        ACCEPT 20,X1
20      FORMAT(2P,F)
        TYPE 30,X1
30      FORMAT(' Number read with P=2 =',F,
        1 /,' (Number read)*10**(-2)')

        TYPE 40,X1
40      FORMAT(/,' The above number written with P=2',/,
        1 ' is =',2P,F,/,' (Number above)*10**(2)')

        END
```

```
EXECUTE TESTP.FOR
FORTRAN: TESTP
MAIN.
LINK:   Loading
[LNKXCT TESTP execution]

Type in a real number
5.

Number read with P=2 =      0.0500000
(Number read)*10**(-2)

The above number written with P=2
is =       5.0000000
(Number above)*10**(2)
CPU time 0.2   Elapsed time 5.1
```

The number the program receives is (5.)*(10**(-2)) and the value typed out is (.05)*(10**(2)).

BN and BZ Editing

12.4.9 BN and BZ Editing

The BN and BZ edit descriptors specify how blanks other than leading blanks are interpreted only for numeric input fields where a width has been specified. These edit descriptors have no effect on numeric output fields.

The BZ descriptor specifies that blanks will be read as zeroes. The BN descriptor specifies that blanks will not be read as zeroes. The use of the BN or BZ edit descriptors in a format overrides the BLANK= specifier in the OPEN statement for the duration of the use of that format. (The BLANK= specifier is described in Section 11.3.3.)

For example:

```
     ACCEPT (FMT=101)A,B,C,D
101  FORMAT (BN,I5,F10.2,BZ,F10.2,F8.5)
```

reads the first two numbers of data, ignoring blanks embedded in the numbers. Then the program reads the second two numbers, substituting zeroes for blanks embedded in the numbers.

Q Editing

12.4.10 Q Editing

The Q edit descriptor sets a corresponding integer variable in the I/O list to the number of characters left in the record being transferred. This descriptor is for use with input transfers only. You can use multiple Q descriptors in the same format list. The Q edit descriptor is useful when you need to know the number of characters remaining in a record.

For example:

```
          TYPE *,'Enter text:'
          ACCEPT 100,L,J1
     100  FORMAT (A5,Q)
```

when used to read the data

```
     Enter text:
     HELLO THIS IS A TEST
```

would yield the value 15 for variable J1, since there are 20 characters in the data, and A5 reads 5 of them.

Numeric Editing

12.4.11 Numeric Editing

The I, F, E, D, G, Z, and O edit descriptors are used to specify the input and output of integer, real, complex, double-precision, hexadecimal, and octal data.

The numeric edit descriptors are repeatable, and can be used without specifying size. For output, if you use a numeric edit descriptor without specifying a field width, the defaults shown in Table 12-5 are used.

For input, the data is scanned until a blank, comma, or character illegal for the specified edit descriptor is encountered, except for A format, which uses the defaults shown in Table 12-5.

Table 12–5: Default Field Widths for Numeric Edit Descriptors

Edit Descriptor	Default Field Width
I	I15
F (single prec.)	*F15.7
F (double prec.)	*F25.18
E (single prec.)	E15.7
E (double prec.)	E25.18
D (single prec.)	D15.7
D (double prec.)	D25.18
G (single prec.)	G15.7
G (double prec.)	G25.18
O (single prec.)	O15
O (double prec.)	O25
L	L15
Z (single prec.)	Z15
Z (double prec.)	Z25
A (single prec.)	A5
A (double prec.)	A10
R (single prec.)	R5
R (double prec.)	R10

* If the default field width for F format is too small for the data, the field width expands to fit the data.

The following conventions apply to all I/O transfers using the numeric edit descriptors:

1. The interpretation of blanks is determined by a combination of any BLANK= specifier in the corresponding OPEN statement (see Section 11.3.3), and any BN or BZ edit descriptor (see Section 12.4.9) that is currently in effect in the format list. A field of all blanks is always equal to zero.

2. On input transfers, with F, E, D, and G editing, a decimal point appearing in the input field overrides the portion of the edit descriptor that specifies the location of the decimal point.

3. On output transfers, the representation of a positive or zero value in the field may be prefixed with a plus, as controlled by the S, SP, and SS edit descriptors (see Section 12.4.7). The representation of a negative value in the field is prefixed with a minus sign.

4. On output transfers, the representation of the transferred datum is right-justified in the specified field. If the number of characters produced by the editing is smaller than the specified field width, leading blanks are inserted in the field.

5. On output transfers, if the number of characters produced exceeds the field width, or if an exponent exceeds its length (as specified in the Ew.dEe or Gw.dEe forms), the entire field width, represented by w, is filled with asterisks.

The next sections describe the individual numeric edit descriptors.

I Editing

12.4.11.1 I Editing - The I edit descriptor specifies integer editing.

The form of the I edit descriptor is:

```
[r]I[w[.m]]
```

where:

r is an optional, nonzero, unsigned, integer constant indicating how many fields of I are being specified. The default is one field.

w is an optional, nonzero, unsigned, integer constant that is equal to the total number of digits in the integer field being described. If w is not specified, for output, the value is 15 (the default); for input, the data is scanned until a blank, comma, or character illegal for the I edit descriptor is encountered.

.m is an optional, unsigned, integer constant (separated from w by a period) that indicates the minimum number of digits to be output to the integer field being described. The default is one digit (I15.1). If necessary, leading zeroes are output.

The value of m must not exceed the value of w. If m is zero and the value of the internal data item is zero, the output field consists of only blank characters, regardless of the sign control in effect.

On input, the Iw.m and the Iw forms of the I edit descriptor are treated the same.

Example:

```
10   FORMAT(I,I8,2I9.5)
```

The first data item is output as a 1- to 15-digit right-justified integer in the first 15 columns. The second item is a 1- to 8-digit integer occupying the next 8 columns. The third and fourth items are 5- to 9-digit integers occupying 9 columns each, with leading zeroes appended to the data to make them 5 digits if necessary.

F Editing

12.4.11.2 F Editing - The F edit descriptor specifies real (floating-point) editing.

The form of the F edit descriptor is:

```
[r]F[w.d]
```

where:

r is an optional, nonzero, unsigned, integer constant indicating the number of fields of F being specified. The default is one field.

w is an optional, nonzero, unsigned, integer constant equal to the total number of digits in the F field being described. This total includes the digits to the right and left of the decimal point, the decimal point itself, and (if included) the sign. On input, if the decimal point is omitted, the rightmost d digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented.

If w is not specified, for output, the value for single-precision is 15 (the default), and the value for double-precision is 25 (the default). For input, the data is scanned until a blank, comma, or character illegal for the F edit descriptor is encountered.

.d is an optional, nonzero, unsigned, integer constant that specifies the total number of fractional digits in the field of width w. The default for single-precision is 7 digits; the default for double-precision is 18. The maximum is 63 digits.

NOTE

If the default field width for F format (with no width specified) is too small for the data, the field width expands to fit the data.

E and D Editing

12.4.11.3 E and D Editing - The E and D edit descriptors specify editing of real data.

The form of the E edit descriptor is:

```
[r]E[w.d[Ee]]
```

The form of the D edit descriptor is:

```
[r]D[w.d[Ee]]
```

where:

r is an optional, nonzero, unsigned, integer constant that is equal to the number of E or D fields being described. The defaults are one single-precision E field and one double-precision D field.

w is an optional, nonzero, unsigned, integer constant equal to the total number of digits in the E or D field being described. The total for both types of fields is equal to:

1. The total number of digits on both sides of the decimal point
2. The decimal point itself
3. The sign for the number (if included)
4. The exponent character (if included)
5. The digits in the exponent

If w is not specified, for output, the value for single precision is 15 (the default), and the value for double precision is 25 (the default). For input, the data is scanned until a blank, comma, or character illegal for the E or D edit descriptor is encountered.

d is an optional, nonzero, unsigned integer constant equal to the total number of fractional digits in the field being described (unless a scale factor greater than one is in effect). The defaults for both the E and D edit descriptors are 7 digits if single precision, and 18 if double precision. The maximum is 63 digits.

e is an optional, nonzero, unsigned, integer constant equal to the total number of digits in the E or D field being described. The default for both types of edit descriptors is two digits. The maximum is 15 digits.

For KL model B systems, if the program is compiled with the /GFLOATING switch (see Section 16.1.3 or 16.2.3), you may want to specify three digits to accommodate the exponent field of double-precision numbers.

G
Editing

12.4.11.4 G Editing - The G edit descriptor allows editing of integer, real, double-precision, logical, complex, or character data. With the exception of real, double-precision, and complex data, the type of conversion performed by the G edit descriptor depends on the type of the corresponding variable in the I/O list.

The form of the G edit descriptor is:

[r]G[w.d[Ee]]

where:

r	is an optional, nonzero, unsigned, integer constant that is equal to the number of G fields being described. The default is one.

w	is an optional, nonzero, unsigned, integer constant equal to the total number of digits in the G field being described. The total for both types of fields is equal to:

1. The total number of digits on both sides of the decimal point

2. The decimal point itself

3. The sign for the number (if included)

4. The exponent character (if included)

5. The digits in the exponent

If w is not specified, for output, the value for single precision is 15 (the default), and the value for double precision is 25 (the default). For input, the data is scanned until a blank, comma, or character illegal for the G edit descriptor is encountered.

d	is an optional, nonzero, unsigned integer constant equal to the total number of fractional digits in the field being described (unless a scale factor greater than one is in effect). The defaults are 7 digits if single precision, and 18 if double precision. The maximum is 63 digits.

e	is an optional, nonzero, unsigned, integer constant equal to the total number of digits in the G field being described. The default is two digits. The maximum is 15 digits.

For KL model B systems, if the program is compiled with the /GFLOATING switch (see Section 16.1.3 or 16.2.3), you may want to specify three digits to accommodate the exponent field of double-precision numbers.

For input, in the case of real, double-precision, and complex data, the G-format conversion is the same as for E-format conversion. For output, however, the type of conversion performed depends on the magnitude of the data items. Table 12-6 illustrates the conversion performed for various ranges of real, double-precision, and complex data.

Table 12–6: Effect of Data Magnitude on G-Format Output Conversions

Data Magnitude (m)	Effective Conversion
m .LT. 0.1.	Ew.d
0.1 .LE. m .LT. 1.0	F(w – n).d,n(x)
1.0 .LE. m .LT. 10.0	F(w – n).(d–1),n(x)
.	.
.	.
.	.
10**d–2 .LE. m .LT. 10**d–1	F(w – n).1,n(x)
10**d–1 .LE. m .LT. 10**d	F(w – n).0,n(x)
m .GE. 10**d	Ew.d

where:

x is a blank

n is 4 for Gw.d and e + 2 for Gw.dEe

where:

x is a blank.

n is 4 for Gw.d and e+2 for Gw.dEe

NOTE

In all numeric field conversions, the field width (w) you specify should be large enough to include the decimal point, sign, and, where applicable, the exponent character (E), the exponent sign, plus the exponent digits. This is in addition to the number of digits in the number to be represented.

If the specified width is too small to accommodate the converted number, the field will be filled with asterisks (*). If the number converted occupies fewer character positions than specified by w, it will be right-justified in the field, and leading blanks will be used to fill the field.

If the numeric data representation cannot fit into the field width F(w-n), the n spaces (n(x)) are removed from the right, and the numeric data representation is again processed into the field width Fw.

Examples of G output conversions (where the ƀ signifies a blank) are:

Format	Internal Value	External Representation
G13.6	0.01234567	ƀ0.123457E-01
G13.6	-1.12345678	-0.123457ƀƀƀƀ
G13.6	1.23456789	ƀƀ1.23457ƀƀƀƀ
G13.6	12.34567890	ƀƀ12.3457ƀƀƀƀ
G13.6	123.45678901	ƀƀ123.457ƀƀƀƀ
G13.6	-1234.56789012	ƀ-1234.57ƀƀƀƀ
G13.6	12345.67890123	ƀƀ12345.7ƀƀƀƀ
G13.6	123456.78901234	ƀƀ123457.ƀƀƀƀ
G13.6	-1234567.89012345	-0.123457E+07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	ƀƀƀƀƀ0.012346
F13.6	-0.12345678	ƀƀƀƀ-0.123457
F13.6	1.23456789	ƀƀƀƀƀ1.234568
F13.6	12.34567890	ƀƀƀƀ12.345679
F13.6	123.45678901	ƀƀƀ123.456789
F13.6	-1234.56789012	ƀ-1234.567890
F13.6	12345.67890123	ƀ12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	*************

Complex Editing

12.4.11.5 Complex Editing - A complex number consists of a pair of separate real numbers. The first number of the pair is the real part of the complex number; the second number is the imaginary part.

The editing of a complex number involves specifying two successive F, E, D, or G edit descriptors. The edit descriptors need not be the same.

A sample format list description for a complex number is:

```
101     FORMAT (F10.2,E10.2)
```

In this sample, the F10.2 edit descriptor provides editing for the real part of the complex number; the E10.2 edit descriptor provides editing for the imaginary part.

You may include any nonrepeatable edit descriptors between the real and imaginary edit descriptors for a complex number.

O (Octal) Editing

12.4.11.6 O (Octal) Editing - The O (octal) edit descriptor specifies octal editing.

The form of the O edit descriptor is:

```
[r]O[w[.m]]
```

where:

r is an optional, nonzero, unsigned, integer constant specifying the number of successive octal fields being described. The default is one octal field.

w is an optional, nonzero, unsigned, integer constant specifying the total number of digits in the octal field being described.

If w is not specified, for output, the value for a single-precision octal field is 15 (the default), and the value for a double-precision octal field is 25 (the default). For input, the data is scanned until a blank, comma, or character illegal for the O edit descriptor is encountered.

m is an optional, unsigned, integer constant specifying the minimum number of digits to be output to the field. The defaults are 12 for single-precision octal values and 24 for double-precision octal values.

Z
(Hexadecimal)
Editing

12.4.11.7 Z Editing - The Z edit descriptor specifies input and output of hexadecimal values. Hexadecimal is a base 16 number system where the characters 0-9 and A-F (or a-f) represent the numbers 0-9 and 10-15, respectively. (On output, A-F only.)

The form of the Z edit descriptor is:

[r]Z[w[.m]]

where:

r is an optional, unsigned, nonzero, integer constant specifying the number of consecutive hexadecimal fields being specified. The default is one hexadecimal field.

w is an optional, unsigned, nonzero, integer constant specifying the total number of digits in the hexadecimal field being described.

If w is not specified, for output, the value for a single-precision hexadecimal field is 15 (the default), and the value for a double-precision hexadecimal field is 25 (the default). For input, the data is scanned until a blank, comma, or character illegal for the Z edit descriptor is encountered.

m is an optional, unsigned, integer constant specifying the minimum number of digits to be output in the field. The default for single-precision hexadecimal fields is 9 digits; the default for double-precision fields is 18 digits.

L
Editing

12.4.12 L Editing

The L edit descriptor provides editing of logical data.

The form of the L edit descriptor is:

    [r]L[w]

where:

r is an optional, unsigned, nonzero, integer constant specifying the number of consecutive logical fields being described. The default is one logical field.

w is an optional, unsigned, nonzero, integer constant specifying the total number of characters in the logical field being described.

If w is not specified, for output, the value is 15 (the default) For input, the data is scanned until a blank, comma, or character illegal for the L edit descriptor is encountered.

You may transfer logical data under format control in a manner similar to numeric data transfer by use of the field descriptor

    Lw

where:

L is the control character and w is an integer specifying the field width. The data is transmitted as the value of a corresponding logical variable in the associated I/O list.

The input field consists of optional blanks, optionally followed by a period, followed by a T for true or F for false, optionally followed by any series of characters (such as, .TRUE. for true or .FALSE. for false). If the entire input data field is blank or empty, a value of false is stored.

On output, w minus 1 blanks followed by T or F will be output if the value of the logical variable is true or false, respectively.

12.4.13 A Editing

The A edit descriptor specifies the editing of character or Hollerith data. The data are stored left-justified in a word and padded with blanks to the right.

NOTE

> The R edit descriptor performs the same function for Hollerith data, only it stores the data right-justified in a word with leading nulls. The R edit descriptor is not supported for character data. For a description of the R edit descriptor, see Section 12.4.14.

The form of the A edit descriptor is:

```
[r]A[w]
```

where:

r is an optional, unsigned, integer constant specifying the number of consecutive A fields being defined. The default is one A field.

w is an optional, unsigned, integer constant specifying the total number of characters in the field being defined. The default for single-precision values is 5 characters; the default for double-precision and complex values is 10 characters.

Depending on the I/O operation, the A edit descriptor transfers character or Hollerith data into or from a variable in an I/O list. A list variable may be of any type. For example,

```
  READ (6,5) V
5 FORMAT (A4)
```

causes four character or Hollerith characters to be read from unit 6 and stored in the variable V.

The A descriptor deals with variables containing left-justified, blank-filled characters. The following list summarizes the result of character or Hollerith data transfer (both internal and external representations) using the A descriptor. These explanations assume that w represents the field width and m represents the total number of characters possible in the variable. Double-precision and complex variables contain 10 characters (m=10); integer, real, and logical variables contain 5 (m=5).

A Descriptor

1. INPUT, where w > m -- The rightmost m characters of the field are read in and stored in the corresponding variable.

2. INPUT, where w < m -- All w characters are read in and stored left-justified and blank-filled in the corresponding variable.

3. OUTPUT, where w > m -- m characters are output and right-justified in the field. The remainder of the field is blank-filled.

4. OUTPUT, where w < m -- The leftmost w characters of the corresponding variable are output.

R Editing

12.4.14 R Editing

The R edit descriptor specifies the editing of Hollerith data. The Hollerith data are stored right-justified with leading nulls. The R edit descriptor is not supported for character data.

NOTE

> The A edit descriptor, described in Section 12.4.13, performs the same function as the R descriptor except that it left-justifies the data in storage with trailing spaces.

The form of the R edit descriptor is:

```
[r]R[w]
```

where:

r is an optional, unsigned, nonzero, integer constant specifying the number of consecutive R fields being defined. The default is one R field.

w is an optional, unsigned, nonzero, integer constant that specifies the total number of characters in the R field.

If w is not specified, for output, the value for a single-precision field is 5 (the default), and the value for a double-precision or complex field is 10. For input, the data is scanned until a blank, comma, or character illegal for the R edit descriptor is encountered.

The R descriptor deals with variables containing right-justified, zero-filled characters. The following list summarizes the result of Hollerith data transfer (both internal and external representations) using the R descriptor. These explanations assume that w represents the field width and m represents the total number of characters possible in the variable. Double-precision and complex variables contain 10 characters (m=10); integer, real, and logical variables contain 5 (m=5).

NOTE

> When more than five characters are stored, bit zero of the low-order word is skipped. Thus, a double-precision or complex variable filled by an R-format data transfer is of the form:
>
> | | | | |
> |---|---|---|---|
> | 0 | data | 0 | data |

R Descriptor

1. INPUT, where w > m -- The rightmost m characters of the field are read in and stored in the corresponding variable.

2. INPUT, where w < m -- All w characters are read in and stored right-justified and zero-filled in the corresponding variable.

3. OUTPUT, where w > m -- m characters are output and right-justified in the field. The remainder of the field is zero-filled.

4. OUTPUT, where w < m -- The rightmost w characters of the corresponding variable are output.

List-Directed Formatting

12.5 LIST-DIRECTED FORMATTING

The use of an asterisk in a data transfer statement in place of a FORMAT statement label specifies list-directed formatting. For this type of formatting, the type of each transferred data item is specified by the types of respective elements in the I/O list.

List-directed input data transfers are performed without regard for column, card, or line boundaries. List-directed output transfers produce records with a maximum length of 72 characters (the default) or the length specified by the RECL specifier (see Section 11.3.27 for devices other than the terminal). Otherwise, the maximum length of the current terminal width is used.

The following is a sample list-directed data transfer statement:

```
READ (5,*)I,IAB,M,L
```

You may use list-directed transfers to read data from any acceptable input device, including a terminal. However, do not use device-positioning commands in conjunction with list-directed data transfers. If you do, the results are unpredictable.

Data for list-directed transfers should consist of alternate constants and delimiters. The constants used should have the following characteristics:

1. Input constants must be of a form acceptable to FORTRAN.
2. Character constants must be enclosed within single quotes, for example, 'ABLE'. Each apostrophe in a character constant must be represented by two apostrophes.
3. The end of a record is equivalent to a blank except when it occurs in a character constant. In this case, the end of the record is ignored and the character constant is continued with the next record. The first character of the continued record must be blank, which is ignored.
4. If the string of a character constant exceeds the length of the data item, the string is truncated. If the string is shorter than the data item, the string is left-justified and remaining character positions are blank filled.
5. Blanks are used as delimiters in list-directed input. Embedded blanks are, therefore, not permitted in any list-directed data item, with the exception of character constants.
6. Decimal points may be omitted from real constants that do not have a fractional part. In this case, it is assumed that the decimal point follows the rightmost digit of a real constant.
7. Complex constants must be enclosed within parentheses.
8. Octal constants must be preceded with a double quote (").
9. A numeric data item can correspond only to a numeric constant, and a character data item can correspond only to a character constant.

A delimiter in a list-directed list of data items separates one data item from another. Delimiters in data for list-directed input must comply with the following:

1. Delimiters may be commas, blanks, or slashes.
2. Delimiters may be either preceded by or followed by any number of blanks, carriage return/line feed characters, tabs, or line terminators; any such combination is treated as a single delimiter.
3. A null item (the complete absence of a data item) is represented by two consecutive commas that have no intervening constant(s). You may place any number of blanks, tabs, or carriage return/line feed characters between the commas of a null item. Each time you specify a null item in the input data, its corresponding list element is skipped (unchanged).

The following illustrates the effect of the input of a null item:

I/O List	A,B,IAB,N
Data input	101,'A',,20

Resulting contents of I/O list items:

A	101.
B	'A'
IAB	unchanged
N	20

4. Slashes (/) cause the current input operation to terminate even if all the items of the I/O list are not filled. The contents of items of the I/O list that either are skipped (by null items) or have not received an input data item before the transfer is terminated remain unchanged. Once the I/O list of the data transfer statement is satisfied, the use of the / delimiter is optional.

5. Once the I/O list has been satisfied (values have been transferred to each item of the list), any items remaining in the input record are skipped.

Constants or null items in data for list-directed input may be assigned a repeat count so that an item is repeated.

A constant with a repeat count is written as:

```
r*K
```

where:

r is an integer constant that specifies the number of times the constant is repeated, the asterisk delimits the repeat count from the constant, and K represents the constant.

A null item with a repeat count is written as an integer, which specifies the repeat count, followed by an asterisk.

The following are examples of constants and null items:

10*5	represents 5,5,5,5,5,5,5,5,5,5
3*'ABLE'	represents 'ABLE','ABLE','ABLE'
3*	represents null,null,null

NOTE

The asterisk form representing nulls must be delimited by a comma or slash; in this case spaces are ignored and not treated as delimiters.

NAMELIST-Statement Formatting

NAMELIST-STATEMENT FORMATTING

The data transfer statements described in Chapter 10 usually include an I/O list, which is a list of variable, array, or array element names that identify the names of the data being transferred.

An alternative way of creating I/O lists is to use the NAMELIST statement. Using this method, you can specify the I/O list in a NAMELIST statement and then reference the list by name in the appropriate data transfer statement.

When you use NAMELIST-statement formatting, as opposed to FORMAT-statement or list-directed formatting, you need only reference the I/O list by NAMELIST name in a data transfer statement.

NAMELIST Statement

NAMELIST STATEMENT

The form of the NAMELIST statement is:

```
NAMELIST /name/list[/name/list]...
```

where:

name — is the name of the NAMELIST I/O list. This is the name referenced in data transfer statements. Each NAMELIST name must be enclosed in slashes.

list — is the list of items comprising the NAMELIST I/O list. Items within the list may be variable names or array names. Separate multiple list items with commas.

Each list of a NAMELIST statement is identified and referenced by the name immediately preceding the list.

The following is an example of creating two NAMELIST I/O lists having the names TABLE and SUMS.

```
DIMENSION C(2,4),TOTAL(10)
NAMELIST/TABLE/A,B,C/SUMS/TOTAL
```

In this example, the name TABLE identifies the list consisting of the variables A and B and the array C, and the name SUMS identifies the list consisting of the array TOTAL.

Once a list has been defined in a NAMELIST statement, one or more I/O statements may reference its name.

The rules for structuring a NAMELIST statement are:

1. You may use a maximum of six characters for a NAMELIST name.

2. You must begin the list name with an alphabetic character.

3. You must enclose the NAMELIST name in slashes.

4. You should use NAMELIST names that are unique within the program.

5. You may define a NAMELIST name only once, and you must define it by a NAMELIST statement. Once defined, you may use the name only in I/O transfer statements.

6. You must define the NAMELIST name before the data transfer statements in which it is used.

7. You must define any dimensioned variable contained in a NAMELIST statement in an array declaration statement preceding the NAMELIST statement.

12.7.1 NAMELIST-Controlled Data Input Transfer

During data input transfers in which a NAMELIST-defined name is referenced, records are read until a record is found that begins with a blank, then $ (dollar sign), and then the desired NAMELIST name. The dollar sign must be the second character in the record; the first character in the record must be a blank.

NOTE

> You may use "&" instead of "$" in NAMELIST-controlled input.

Data items of records to be input (read) using NAMELIST-defined lists must be separated by commas and may be of the following form:

```
V=K1,K2,...,Kn
```

where:

V — may be a variable, array, or array element name.

K1,...,Kn — are constants. A series of identical constants may be represented as a single constant preceded by a repetition count (5*5 represents 5,5,5,5,5). You can specify more than one constant only if V is an array. If V is a scalar, then you may have only K1.

The input data is always converted to the type of the list variable when there is a conflict of types. A character constant is truncated from the right, or extended on the right with blanks, if necessary, to yield a constant of the same length as the variable, array, or substring.

The input operation continues until another $ symbol is detected. If variables appear in the NAMELIST record that do not appear in the NAMELIST list, an error condition will occur.

A character constant must have delimiting apostrophes. If an apostrophe is part of a character constant, it must be represented by two consecutive apostrophes, which must be contained in the same record (one apostrophe cannot end a record, and the other apostrophe start a record).

For example, assume:

1. A is a 2-dimensional real array
2. B is a 1-dimensional integer array
3. C is an integer variable
4. D is a character variable of length 5.
5. The program contains the NAMELIST declaration:

   ```
   NAMELIST /FRED/ A,B,C,D
   ```

6. The input data is as follows:

   ```
   b̸$FRED A(7,2)=4, B=3,6*2.8, C=3.32, D='RON'$
   ```

A READ statement referring to the NAMELIST-defined name FRED will result in the following:

1. The integer 4 will be converted to floating point and placed in A(7,2).
2. The integer 3 will be placed in B(1).
3. The integer 2 (after being truncated) will be placed in B(2),B(3),...,B(7).
4. The floating point number 3.32 will be converted to the integer 3 and placed in C.
5. The character string 'RONb̸b̸' will be placed in D.

12.7.2 NAMELIST-Controlled Data Output Transfers

When a WRITE statement refers to a NAMELIST-defined name, all variables and arrays and their values belonging to the named list are written out, each according to its type. Character constants are written with delimiting apostrophes. Arrays are written out by columns. Output data is written so that:

1. The fields for the data will be large enough to contain all the significant digits.
2. The output can be read by an input statement referencing a NAMELIST-defined list.

For example, if ARRAY is a 2 X 3 real array, A1 is a real variable, K1 is an integer variable, and D is a character variable containing the five characters AB'CD, the statements:

```
     REAL ARRAY(2,3)
     CHARACTER D*5
     DATA ARRAY,A1,KI,D/-6.75, 0.234E-04, 680.0, -17.8,0.0,00,
     1               73.1, 3, 'AB''CD'/
     NAMELIST/NAM1/ARRAY,A1,K1,D
     WRITE (u,NAM1)
```

generate the following form of output:

```
Column 1
|
↓
b̸$NAM1
 ARRAY=  -6.750000,  0.2340000E-04,  680.0000,  -17.80000,  2*0.0000000,

 A1=   73.10000, K1= 3, D='AB''CD'
 $END
```

NOTE

> Do not use device-positioning commands such as BACKSPACE or SKIPRECORD with NAMELIST-controlled I/O operations. If you do, the results are unpredictable.

CHAPTER 13

FUNCTIONS AND SUBROUTINES

Procedures you use repeatedly in a program can be written once and then referenced each time you need the procedure. Procedures that may be referenced are either contained within the program in which they are referenced, or self-contained executable procedures that can be compiled separately. The kinds of procedures that can be referenced are:

1. Intrinsic functions (FORTRAN-defined functions)
2. Statement functions
3. External functions
4. Subroutines

The first three of these categories are referred to collectively as functions or function procedures; procedures of the last category are referred to as subroutines or subroutine procedures.

Intrinsic functions perform a predefined computation with a specific number and type of arguments. These functions are provided by FORTRAN (see Section 13.1).

Statement functions are user-defined, single statement procedures that resemble assignment statements. The appearance of a statement function reference in an expression causes the user-defined computation to be performed (see Section 13.2).

External functions are separate program units that generally compute a single value using one or more parameters. There are two types of external functions available: user-defined and FORTRAN-supplied. A user-defined external function is defined with a FUNCTION statement. Both types of external functions are invoked by including a function reference in an expression (see Section 13.3).

Subroutines are external program units that are used to perform multiple computations or alter variables. There are two types of subroutines available: user-defined and FORTRAN-supplied. User-defined subroutines are defined with a SUBROUTINE statement (see Section 13.4.2.1). Both types of subroutines are invoked with a CALL statement (see Section 13.4.2.2).

13.1 INTRINSIC FUNCTIONS

Intrinsic functions are supplied with the FORTRAN software. Each intrinsic function performs a predefined computation. There are two types of intrinsic functions: specific and generic.

Specific functions have an implicitly defined data type. Each specific function requires arguments of a particular type and returns results of a predefined type. The IMPLICIT statement cannot be used to change the type of a specific intrinsic function.

The data type of the return value of a generic function is determined by the data type of its arguments. The FORTRAN generic functions are:

ABS
ACOS
AINT
ALOG
ALOG10
AMAX1
AMIN1
ANINT
ASIN
ATAN
ATAN2
CMPLX
COS
COSH
DBLE
DIM
EXP
INT
LOG
LOG10
MAX
MIN
MOD
NINT
REAL
SIGN
SIN
SINH
SQRT
TAN
TANH

NOTE

> Table 13-1 lists all the specific and generic intrinsic functions. For ease of identification, each generic function name in the table is indicated by an asterisk.

13.1.1 Using an Intrinsic Function

An intrinsic function is used in a FORTRAN expression by referencing the name of the function in an expression. For example, the following program contains two intrinsic functions: ABS (returns the absolute value of the argument) and SQRT (returns the square root of the argument).

```
PROGRAM TEST

Y = -64
A = ABS (Y)
TYPE ,A

B = SQRT (A)
TYPE ,B

END
```

When the preceding program is executed, variable Y is assigned the value -64. The ABS function in the second expression calculates the absolute value of -64. Next, in the third expression, the SQRT function calculates the square root of the absolute value of Y, which is A. The square root of A is assigned to B in the third expression. Executing the program yields the following results:

```
EXECUTE TEST.FOR
FORTRAN: TEST
TEST
LINK:   Loading
[LNKXCT TEST execution]
64.00000
8.000000
CPU time 0.1  Elapsed time 4.0
```

The following example contains specific and generic functions. In the example, the generic function SQRT is used to find the square root of the double-precision value 64.0. Next, the specific function DSQRT is used to find the square root of the double-precision value 64.0. If the argument supplied to DSQRT was not a double-precision number, a fatal compilation error would result.

```
      PROGRAM TESFUN

      DOUBLE PRECISION A,B,AR,BR
      REAL C,CR

      A = 64.00D0
      B = 64.00D0
      C = 64.00

C     GENERIC SQRT RETURNS DP
C     SQRT BECAUSE ARG TYPE IS DP

      AR = SQRT(A)

C     SPECIFIC DSQRT PERFORMS THE
C     SAME FUNCTION WHEN GIVEN A DP
C     ARGUMENT

      BR = DSQRT(A)

C     SPECIFIC SQRT RETURNS A REAL
C     VALUE RESULT

      CR = SQRT(C)

      TYPE , AR,BR,CR
      END
```

Executing the program above yields the following results:

```
EXE TESFUN
LINK:   Loading
[LNKXCT TESFUN Execution]
8.0000000000000000, 8.0000000000000000, 8.000000
CPU time 0.1   Elapsed time 0.7
```

Table 13-1 lists the FORTRAN intrinsic functions. This table gives function definitions, argument and function types, and ranges of acceptable values. Each function contains a description of the range for valid arguments(s) and the range within which the function returns valid results. If function arguments do not fall within the specified range, the result of the function is undefined.

For more information on the precision and accuracy of the FORTRAN intrinsic functions, refer to the TOPS-10/TOPS-20 Common Math Library Manual.

Table 13–1: FORTRAN Intrinsic Functions

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Exponential					
EXP*	y = e**x	Real	-89.415 .LE. x .LE. 88.029	Real	y .GT. 0
DEXP**	y = e**x	Double	D–floating: -89.415 .LE. x .LE. 88.029 G–floating: -710.475 .LE. x .LE. 709.089	Double	D–floating: y .GT. 0 G–floating: y .GT. 0
CEXP	w = e**z	Complex	-89.415 .LE. REAL(z) .LE. 88.029 |AIMAG(z)| .LE. 36394.429	Complex	All COMPLEX Numbers
Logarithm (LOG, LOG10=Generic Functions)					
ALOG*	y = log(x) [base e]	Real	x .GT. 0	Real	-89.415 .LE. y .LE. 88.029
DLOG**	y = log(x) [base e]	Double	D–floating: x .GT. 0 G–floating: x .GT. 0	Double	D–floating: -89.415 .LE. y .LE. 88.029 G–floating: -710.475 .LE. y .LE. 709.089
CLOG	w = log(z) [base e]	Complex	z .NE. (0,0)	Complex	-89.415 .LE. REAL(w) .LE. 88.029 -PI .LT. AIMAG(w) .LE. PI
ALOG10*	y = log(x) [base 10]	Real	x .GT. 0	Real	-38.832 .LE. y .LE. 38.230
DLOG10**	y = log(x) [base 10]	Double	D–floating: x .GT. 0 G–floating: x .GT. 0	Double	D–floating: -38.832 .LE. y .LE. 38.320 G–floating: -308.555 .LE. y .LE. 307.953
Square Root					
SQRT*	y = SQRT(x) = x**1/2	Real	x .GE. 0	Real	y .GE. 0
DSQRT**	y = SQRT(x) = x**1/2	Double	D–floating: x .GE. 0 G–floating: x .GE. 0	Double	D–floating: y .GE. 0 G–floating: y .GE. 0
CSQRT	w = SQRT(z) = z**1/2	Complex	Any COMPLEX Number	Complex	All COMPLEX Numbers REAL(w) .GE. 0

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Trigonometric					
SIN*	y = sin(x)	Real	|x| .LE. 210828714	Real	-1 .LE. y .LE. 1
SIND	y = sin(x) (degrees)	Real	|x| .LE. 47185919	Real	-1 .LE. y .LE. 1
DSIN**	y = sin(x)	Double	D–floating: |x| .LE. 6746518852 G–floating: |x| .LE. 1686629713	Double	D–floating: -1 .LE. y .LE. 1 G–floating: -1 .LE. y .LE. 1
CSIN	w = sin(z)	Complex	|REAL(z)| .LE. 210828714 |AIMAG(z)| .LE. 88.895	Complex	All COMPLEX Numbers
COS*	y = cos(x)	Real	|x| .LT. 210828714	Real	1 .LT. y .LE. 1
COSD	y = cos(x) (degrees)	Real	|x| .LT. 47185919	Real	1 .LT. y .LE. 1
DCOS**	y = cos(x)	Double	D–floating: |x| .LT. 6746518852 G–floating: |x| .LT. 1686629713	Double	D–floating: -1 .LE. y .LE. 1 G–floating: -1 .LE. y .LE. 1
CCOS	w = cos(z)	Complex	|REAL(z)| .LE. 210828714 |AIMAG(z)| .LE. 88.895	Complex	All COMPLEX Numbers
TAN*	y = tan(x)	Real	|x| .LE. 36396	Real	All REAL Numbers
DTAN**	y = tan(x)	Real	D–floating: |x| .LE. 3373259426 G–floating: |x| .LE. 843314856	Double	D–floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers
COTAN	y = cot(x)	Real	|x| .LE. 36396	Real	All REAL Numbers
DCOTAN**	y = cot(x)	Double	D–floating: |x| .LE. 3373259426 G–floating: |x| .LE. 843314856	Double	D–floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Inverse Trigonometric					
ASIN*	y = arcsin(x)	Real	-1 .LE. x .LE. 1	Real	-PI/2 .LE. y .LE. PI/2
DASIN**	y = arcsin(x)	Double	D–floating: -1 .LE. x .LE. 1	Double	D–floating: -PI/2 .LE. y .LE. PI/2
			G–floating: -1 .LE. x .LE. 1		G–floating: -PI/2 .LE. y .LE. PI/2
ACOS*	y = arccos(x)	Real	-1 .LE. x .LE. 1	Real	0 .LE. y .LE. PI
DACOS**	y = arccos(x)	Double	D–floating: -1 .LE. x .LE. 1	Double	D–floating: 0 .LE. y .LE. PI
			G–floating: 1 .LE. x .LE. 1		G–floating: 0 .LE. y .LE. PI
ATAN*	y = arctan(x)	Real	x = any REAL Numbers	Real	-PI/2 .LE. y .LE. PI/2
DATAN**	y = arctan(x)	Double	D–floating: x = any D–FLOATING Numbers	Double	D–floating: -PI/2 .LE. y .LE. PI/2
			G–floating: x = any G–FLOATING Numbers		G–floating: -PI/2 .LE. y .LE. PI/2
ATAN2*	y = arctan(arg1/arg2)	Real	arg1,arg2 = any REAL Numbers	Real	-PI .LE. y .LE. PI
DATAN2**	y = arctan(arg1/arg2)	Double	D–floating: arg1,arg2 = any D–FLOATING Numbers	Double	D–floating: -PI .LE. y .LE. PI
			G–floating: arg1,arg2 = any G–FLOATING Numbers		G–floating: -PI .LE. y .LE. PI
Hyperbolic					
SINH*	y = sinh(x)	Real	|x| .LE. 88.722	Real	All REAL Numbers
DSINH**	y = sinh(x)	Double	D–floating: |x| .LE. 88.722	Double	D–floating: All D–FLOATING Numbers
			G–floating: |x| .LE. 709.782		G–floating: All G–FLOATING Numbers
COSH*	y = cosh(x)	Real	|x| .LE. 88.722	Double	y .GE. 1
DCOSH**	y = cosh(x)	Double	D–floating: |x| .LE. 88.722	Double	D–floating: y .GE. 1
			G–floating: |x| .LE. 709.782		G–floating: y .GE. 1
TANH*	y = tanh(x)	Real	Any REAL Numbers	Real	-1 .LE. y .LE. 1
DTANH**	y = tanh(x)	Double	D–floating: Any D–FLOATING Numbers	Double	D–floating: -1 .LE. y .LE. 1
			G–floating: Any G–FLOATING Numbers		G–floating: -1 .LE. y .LE. 1

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Absolute Value					
ABS*	y = |x|	Real	Any REAL Numbers	Real	y .GE. 0
IABS	y = |i|	Integer	Any INTEGER Numbers	Integer	y .GE. 0
DABS**	y = |x|	Double	D-floating: Any D–FLOATING Numbers G-floating: Any G–FLOATING Numbers	Double	D–floating: y .GE. 0 G–floating: y .GE. 0
CABS	y = |z|	Complex	Any COMPLEX Numbers	Real	y .GE. 0
Truncation					
AINT*	Sign of arg * largest integer .LT. |arg|	Real	Any REAL Numbers	Real	All REAL Numbers
INT*	Sign of arg * largest integer .LT. |arg|	Real	Any REAL Numbers	Integer	All INTEGER Numbers
IDINT	Sign of arg * largest integer .LT. |arg|	Double	Any DOUBLE PRECISION Numbers	Integer	All INTEGER Numbers
DINT**	Sign of arg* largest integer .LT. |arg|	Double	D-floating: Any D–FLOATING Numbers G-floating: Any G–FLOATING Numbers	Double	D–floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers
Nearest Whole Number					
ANINT*	y = int(x+.5) if x .GE. 0 else y=int(x–.5)	Real	Any REAL Numbers	Real	All REAL Numbers
DNINT**	y = int(x+.5) if x .GE. 0 else y = int(x–.5)	Double	D-floating: Any D–FLOATING Numbers G-floating: Any G–FLOATING Numbers	Double	D-floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers
Nearest Integer					
NINT*	y = int(x+.5) if x .GE. 0 else y = int(x–.5)	Real	x LE. (2**35)–1 x GE. –(2**35)	Integer	All INTEGER Numbers
IDNINT	y = int(x+.5) if x .GE. 0 else y = int(x–.5)	Double	x LE. (2**35) 1 x GE. ·(2**35)	Integer	All INTEGER Numbers

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Remaindering					
AMOD	Remainder when arg1 is divided by arg2	Real	arg2 .NE. 0	Real	0 .LE. y .LT. arg2
MOD*	Remainder when arg1 is divided by arg2	Integer	arg2 .NE. 0	Integer	0 .LE. y .LT. arg2
DMOD**	Remainder when arg1 is divided by arg2	Double	D–floating: arg2 .NE. 0 G- floating: arg2 .NE. 0	Double	D–floating: 0 .LE. y .LT. arg2 G–floating 0 .LE. y .LT. arg2
Maximum Value (MAX = Generic Function)					
AMAX0	Argument with greatest value	Integer	Any INTEGER Numbers	Real	All REAL Numbers
AMAX1*	Argument with greatest value	Real	Any REAL Numbers	Real	All REAL Numbers
MAX0	Argument with greatest value	Integer	Any INTEGER Numbers	Integer	All INTEGER Numbers
MAX1	Argument with greatest value	Real	Any REAL Numbers	Integer	All INTEGER Numbers
DMAX1**	Argument with greatest value	Double	D- floating: Any D FLOATING Numbers G- floating: Any G–FLOATING Numbers	Double	D–floating: All D–FLOATING: Numbers G–floating: All G–FLOATING Numbers
Minimum Value (MIN = Generic Function)					
AMIN0	Argument with least value	Integer	Any INTEGER Numbers	Real	All REAL Numbers
AMIN1*	Argument with least value	Real	Any REAL Numbers	Real	All REAL Numbers
MIN0	Argument with least value	Integer	Any INTEGER Numbers	Integer	All INTEGER Numbers
MIN1	Argument with least value	Real	Any REAL Numbers	Integer	All INTEGER Numbers
DMIN1**	Argument with least value	Double	D floating: Any D–FLOATING Numbers G–floating: Any G–FLOATING Numbers	Double	D–floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Transfer of Sign					
SIGN*	If arg2 .GE. 0 then |arg1| else -|arg1|	Real	Any REAL Numbers	Real	All REAL Numbers
ISIGN	If arg2 .GE. 0 then |arg1| else -|arg1|	Integer	Any INTEGER Numbers	Real	All INTEGER Numbers
DSIGN**	If arg2 .GE. 0 then |arg1| else -|arg1|	Double	D–floating: Any D–FLOATING Numbers G–floating: Any G–FLOATING Numbers	Double	D–floating: All D–FLOATING Numbers G–floating: All G–FLOATING Numbers
Positive Difference					
DIM*	If arg1 .GT. arg2 then arg1 - arg2 else 0	Real	Any REAL Numbers	Real	y .GE. 0
IDIM	If arg1 .GT. arg2 then arg1 - arg2 else 0	Integer	Any INTEGER Numbers	Integer	y .GE. 0
DDIM**	If arg1 .GT. arg2 then arg1 - arg2 else 0	Double	D–floating: Any D–FLOATING Numbers G–floating: Any G–FLOATING Numbers	Double	D–floating: y .GE. 0 G–floating: y .GE. 0
Double Precision Product					
DPROD	arg1*arg2	Real	Any REAL Numbers	Double	ALL DOUBLE PRECISION Numbers
Conversion Routines					
CONJG	arg = x + i*y, CONJG = x - i*y	Complex	Any COMPLEX Numbers	Complex	All COMPLEX Numbers
REAL*	arg = x + i*y returns x	Complex	Any COMPLEX Numbers	Real	All REAL Numbers
AIMAG	arg = x + i*y returns y	Complex	Any COMPLEX Numbers	Real	All REAL Numbers
CMPLX*	Returns arg1 + i*arg2	Real	Any REAL Numbers	Complex	All COMPLEX Numbers
DFLOAT	Integer to double precision	Integer	Any INTEGER Numbers	Double	|y| .LT. 2**35
DBLE*	Real to double precision	Real	Any REAL Numbers	Double	All DOUBLE PRECISION Numbers
SNGL	Double precision to real	Double	Any DOUBLE PRECISION Numbers	Real	All REAL Numbers
FLOAT	Integer to real	Integer	Any INTEGER Numbers	Real	|y| .LT. 2**35
IFIX	Real to integer	Real	|x| .LT. 2**35	Integer	All INTEGER Numbers
ICHAR	Character to Integer	Character	First character of character value	Integer	0 .LE. y .LE. 127
CHAR	Integer to Character	Integer	0 .LE. y .LE. 127	Character	All Single Character

Table 13–1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
Length					
LEN	Length of character entity	Character	Any CHARACTER Value	Integer	y .GE. 1
Index of a Substring					
INDEX	Return location of arg2 within arg1 if not found return 0	Character	Size of character string arg1 must be larger than or equal to arg2	Integer	y .GE. 0
Character Comparisons					
LGE	arg1 .GE. arg2	Character	Any CHARACTER Value	Logical	.TRUE. or .FALSE.
LGT	arg1 .GT. arg2	Character	Any CHARACTER Value	Logical	.TRUE. or .FALSE.
LLE	arg1 .LE. arg2	Character	Any CHARACTER Value	Logical	.TRUE. or .FALSE.
LLT	arg1 .LT. arg2	Character	Any CHARACTER Value	Logical	.TRUE. or .FALSE.
Bit Manipulation					
IAND	Performs a logical AND on corresponding bits of arg1 and arg2	Integer	arg1.arg2 = any Integer Numbers	Integer	All Integer Numbers
IOR	Performs an inclusive OR on corresponding bits of arg1 and arg2	Integer	arg1.arg2 = any Integer Numbers	Integer	All Integer Numbers
IEOR	Performs an exclusive OR on corresponding bits of arg1 and arg2	Integer	arg1.arg2 = any Integer Numbers	Integer	All Integer Numbers
NOT	Complements each bit of argument	Integer	Any Integer Number	Integer	All Integer Numbers
ISHFT	Logically shifts arg1 left arg2 bits if arg2 is positive; arg1 is logically shifted right arg2 bits if arg2 is negative	Integer	arg1 = any Integer Number 36 .LE. arg2 .LE. 36	Integer	All Integer Numbers
ISHFTC	Circularly shifts (rotates) rightmost arg3 bits of arg1 by arg2 places. If arg2 is positive, the rotation is to the left; if arg2 is negative, the rotation is to the right	Integer	arg1 = any Integer Number 36 .LE. arg2 .LE. 36 1 .LE. arg3 .LE. 36	Integer	All Integer Numbers

Table 13-1: FORTRAN Intrinsic Functions (Cont.)

Name	Definition	Argument Type	Argument Restrictions	Function Type	Result Range
			Bit Manipulation		
IBITS	Extracts bits arg2 through arg2+arg3-1 from arg1***	Integer	arg1 = any Integer Number 0 .LE. arg2 .LE. 35 arg2 + arg3 .LE. 36	Integer	All Integer Numbers
IBSET	Returns the value of arg1 with bit arg2 of arg1 set to 1	Integer	arg1 = any Integer Number 0 .LE. arg2 .LE. 35	Integer	All Integer Numbers
IBCLR	Returns the value of arg1 with bit arg2 of arg1 set to 0	Integer	arg1 = any Integer Number 0 .LE. arg2 .LE. 35	Integer	All Integer Numbers
BTEST	Returns .TRUE. if bit arg2 of arg1 equals 1; returns .FALSE. if bit arg2 of arg1 is 0	Integer	arg1 = any Integer Number 0 .LE. arg2 .LE. 35	Logical	.TRUE. or .FALSE.

Notes:

* = Generic function

** = G-floating double-precision functions (KL model B only) -- are used if /GFLOATING compiler switch is specified (see Section 16.1.3 or 16.2.3).

*** = See also the MVBITS Subroutine, Section 13.4.1.21

(2**35)-1 = 34359738367
-(2**35) = -34359738368

13.1.2 Character Intrinsic Functions

Character intrinsic functions are functions that take character arguments or return character values. Character comparison intrinsic functions are functions that take character arguments and return logical values.

FORTRAN provides four character intrinsic functions:

1. LEN

   The LEN function returns the length of a character expression. The LEN function has the following form:

   ```
   LEN(arg)
   ```

   where:

   arg is a character expression. The value returned indicates how many characters there are in the expression.

   The following example illustrates the LEN function:

   ```
   C    This subroutine reverses an entire character
   C    string.

        SUBROUTINE REVERS(S)
        CHARACTER T, S*(*)

        J = LEN(S)
        DO 10 I=1,J/2
             T = S(I:I)
             S(I:I) = S(J:J)
             S(J:J) = T
             J = J - 1
   10   CONTINUE

        RETURN
        END
   ```

2. INDEX

   The INDEX function searches for a substring (arg2) in a specified character string (arg1), and, if it finds the substring, returns the substring's starting position. If arg2 occurs more than once in arg1, the starting position of the first (leftmost) occurrence is returned. If arg2 does not occur in arg1, the value zero is returned. The INDEX function has the following form:

   ```
   INDEX(arg1,arg2)
   ```

   where:

   arg1 is a character expression specifying the string to be searched for the substring specified by arg2.

   arg2 is a character expression specifying the substring that is searched for.

The following example illustrates the INDEX function:

```
C    This subroutine places the symbol # into the
C    variable MARKS at places corresponding to the
C    beginning of all occurrences of the substring
C    SUB within the string S.

     SUBROUTINE FINSTR(SUB,S)
     CHARACTER*(*) SUB, S
     CHARACTER*132 MARKS
     INTEGER I,J

     I = 1
     MARKS = ' '

10   J = INDEX(S(I:), SUB)
     IF (J .NE. 0) THEN
          I = 1 + J
          MARKS(I-1:I-1) = '#'
          IF (I .LE. LEN(S)) GO TO 10
     END IF

     WRITE (5,91) S, MARKS
91   FORMAT (2(/1X,A))
     END
```

3. ICHAR

The ICHAR function converts a character expression to its equivalent ASCII code and returns the ASCII value. The ICHAR function has the following form:

```
ICHAR(arg)
```

where:

arg   is the character to be converted to an ASCII code. If arg is longer than one character, only the value of the first character is returned; the remaining characters are ignored.

4. CHAR

The CHAR function returns the single character whose ASCII code is the integer or octal argument. The CHAR function has the following form:

```
CHAR(arg)
```

where:

arg is an integer expression.

The following example illustrates the CHAR and ICHAR functions:

```
     CHARACTER C*1
     INTEGER I

C    Convert number between 0 and 9 in I to a character
C    digit

     C = CHAR(I+ICHAR('0'))

     END
```

13.1.3 Character Comparison Functions

The four character comparison functions provided with FORTRAN are:

LLT, where LLT(arg1,arg2) is equivalent to (arg1.LT.arg2)

LLE, where LLE(arg1,arg2) is equivalent to (arg1.LE.arg2)

LGT, where LGT(arg1,arg2) is equivalent to (arg1.GT.arg2)

LGE, where LGE(arg1,arg2) is equivalent to (arg1.GE.arg2)

The comparison functions have the following form:

```
func(arg1,arg2)
```

where:

arg is a character expression.

The character comparison functions defined by the FORTRAN-77 standard are guaranteed to make comparisons according to the ASCII collating sequence, even on non-ASCII processors. On TOPS-10/20 systems, the character comparison functions are identical to the corresponding character relationals.

An example of the use of a character comparison function follows:

```
CHARACTER*10 CH2
IF (LGT(CH2,'SMITH')) STOP
```

The IF statement in this example is equivalent to:

```
IF (CH2.GT.'SMITH') STOP
```

13.1.4 Bit Manipulation Functions

Intrinsic bit manipulation functions are used for manipulation of the bits in the binary patterns that represent integers. Integer data types are represented internally in binary two's complement notation. Bit positions in the binary representation are numbered from right (least significant bit) to left (most significant bit); the rightmost bit position is numbered 0, and the leftmost bit position is numbered 35. A bit in a binary pattern has a value of 0 or 1.

The intrinsic functions IAND, IOR, IEOR, and NOT perform bitwise operations on all the bits of their arguments. Bit 0 of the result is the result of applying the specified logical operation to bit 0 of the arguments. Bit 1 of the result is the result of applying the specified logical operation to bit 1 of the arguments, and so on for all the bits of the result.

The shift functions ISHFT and ISHFTC shift binary patterns. A positive shift count indicates a left shift, while a negative shift count indicates a right shift. A shift count of zero means no shift. ISHFT specifies a logical shift; bits shifted out of one end are lost and zeros are shifted in at the other end. ISHFTC performs a circular shift; bits shifted out at one end are shifted back in at the other end.

The function IBITS and the subroutine MVBITS (see Section 13.4.1.21) operate on bit fields. A bit field is a contiguous group of bits within a binary pattern. Bit fields are specified by a starting bit position and a length. A bit field must be entirely contained in its source operand.

For example, the integer 79 is represented by the following binary pattern:

```
0...0101111

n...6543210 (bit position)
```

where:

n is 35 (the number of bit positions in an integer).

You can refer to the bit field contained in bits 3 through 6 by specifying a starting position of 3 and a length of 4. In the above example, the selected bit pattern would be the following:

```
0...000101
```

Negative integers are represented in two's complement notation. The integer -79 is represented by the following binary pattern:

```
1...1010001

n...6543210 (bit position)
```

where:

n is 35 (the number of bit positions in an integer).

NOTE

> The value of bit position 35 is 1 for a negative number and 0 for a non-negative. Also, all the high-order bits of the pattern to the left of the value up to bit 35 are the same as bit 35.

IBITS and MVBITS operate on bit fields. Both the starting position of a bit field and its length are arguments to these intrinsics. IBSET, IBCLR, and BTEST operate only on one bit. Thus, they do not require a length argument.

13.2 STATEMENT FUNCTIONS

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic, character, or logical assignment statement. The statement function enables you to define a single-line computation once in your program, give it a name, and have that calculation performed each time you reference the statement function in the program. A statement function is classified as a nonexecutable statement.

13.2.1 Defining a Statement Function

Statement functions have the following form:

```
     fun ( [d [,d]...] ) = e
```

where:

fun is the symbolic name for the statement function. The function name follows the rules for forming symbolic names in FORTRAN (see Chapter 4).

d is an optional dummy argument. Separate multiple dummy arguments with commas. (Dummy arguments are described in Section 13.4.5.) The parentheses are still required if no dummy arguments are specified.

e is any type of FORTRAN expression. The expression part of the statement function (to the right of the equal sign) defines the computation performed using the dummy arguments.

The relationship between fun and e must conform to the assignment rules in Sections 8.1, 8.2, and 8.4. Note that the type of the expression may be different from the type of the statement function name.

The following rules govern the formation and use of statement functions:

1. The dummy argument list in the statement function serves only to indicate the order, number, and type of arguments for the statement function.

2. The dummy arguments used in a statement function are local to that statement function. It is valid to use the same names in multiple statement functions in the same program unit. A dummy argument name may also be used elsewhere in the program unit to identify a variable of the same type, including its appearance as a dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement.

3. Each dummy argument in a statement function dummy argument list must be unique; the same dummy argument cannot appear twice in a single list.

Each variable reference in the function can be either a reference to a dummy argument of the statement function, or a reference to a variable that appears within the same program unit as the statement function statement.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression of a statement function statement is a reference to the statement function dummy argument.

13.2.2 Using a Statement Function

Statement functions are used in FORTRAN programs by referencing the name of the statement function in an expression that is in the same program unit as the statement function definition. If a character function is referenced in a program unit, the function length specified in the program unit must be an integer constant expression.

For example, the following program uses a statement function (called PROFIT) to determine the profit for a product. In the statement function definition, PROFIT is defined as the difference between wholesale and retail prices minus .05 sales tax.

```
         PROGRAM STAFUN

         PROFIT (A,B) = ((A - B) - (A * .05))

         WRITE(UNIT=5,FMT=100)
    100  FORMAT(1X,'Enter Wholesale Price and Retail Price')

         ACCEPT *,WHOSAL,RETAIL

    150  C = PROFIT(RETAIL,WHOSAL)

         WRITE(UNIT=5,FMT=101) C
    101  FORMAT(1X,'The Profit (minus sales tax) is: ',F8.2)

         END
```

When the program is executed, the retail and wholesale values are entered at the terminal. Next, the expression at statement number 150 uses the values of RETAIL and WHOSAL to calculate the profit as defined in the PROFIT statement function. A sample execution of this program yields the following results:

```
EXECUTE STATE
LINK:   Loading
[LINKXCT STAFUN execution]
Enter Wholesale Price and Retail Price
31.67   45.95
The profit (minus sales tax) is:    11.98
CPU time 0.2   Elapsed time 18.5
```

When a FORTRAN expression that contains a statement function reference is executed, the following happens:

1. The actual arguments contained in the statement function reference are evaluated.

2. The actual arguments in the statement function are associated with the dummy arguments in the statement function definition.

3. The expression portion of the statement function is evaluated using the actual arguments.

4. If necessary, the value of the expression is converted to the type of the statement function. Finally, the value resulting from the expression evaluation is substituted in the expression containing the statement function reference.

13.2.3 Statement Function Restrictions

The following rules and restrictions must be adhered to when using statement functions:

1. The actual arguments in a function reference must agree in type and number with the corresponding dummy arguments in the statement function dummy argument list.

2. An actual argument in a statement function reference can itself be an expression; all actual arguments must be defined when a statement function reference is evaluated.

3. A statement function can only be referenced in the program unit that contains the statement function statement.

4. A statement function must not contain a reference to another statement function that appears later in the program unit, but can contain a reference to another statement function that appears earlier in the program.

5. The symbolic name used to identify a statement function must not appear as a symbolic name in any specification statement except a type statement (to specify the type of the function) or as the name of a common block in the same program unit.

6. An external function reference (see Section 13.3) in the expression part of a statement function statement must not cause a dummy argument of the statement function to become undefined or redefined.

7. The symbolic name of a statement function may not be an actual argument. It must not appear in an EXTERNAL statement (see Section 7.6).

8. A statement function statement in a function subprogram (see Section 13.3.4.) must not contain a function reference to the name of the function subprogram or an entry name in the function subprogram.

9. An actual argument in a statement function reference can be any expression, including a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses.

10. The length specification of a character statement function must be an integer constant expression.

13.3 EXTERNAL FUNCTIONS

An external function is a procedure that is defined externally to the program unit that references it. FORTRAN offers two types of external functions: FORTRAN-supplied and user-defined. The FORTRAN-supplied external functions are described in Section 13.3.1; the user-defined functions are described in Sections 13.3.2 through 13.3.4.

13.3.1 FORTRAN-Supplied External Functions

FORTRAN-supplied external functions are similar to intrinsic functions. To use a FORTRAN-supplied function, you reference its name in an expression.

The following are the FORTRAN-supplied external functions:

x=DTOG(y) returns a G-floating double-precision number in the range 1.47 x 10**-39 to 1.70 x 10**+38. The argument y is a D-floating double-precision number. (Also, see the DTOGA subroutine, Section 13.4.1.12.)

x=GTOD(y) returns a D-floating double-precision number in the range 1.47 x 10**-39 to 1.70 x 10**+38. The argument y is a G-floating double-precision number. (Also, see the GTODA subroutine, Section 13.4.1.18.)

x=LSNGET(unit) returns the last line number read in a line sequenced file. LSNGET returns a positive integer if the last line has a valid line number; returns zero if the last line is a page mark; or returns -1 if the last line number is invalid (such as, AAAA with bit 35 set). It also returns -1 if the file contains no line number, or was opened with a mode other than LINED (see Section 11.3.20).

x=RAN(0) returns a pseudo random floating-point number in the range of 0.LT.x.LT.1. The argument is a dummy (not used) and may be any number. Refer to the related subroutines SETRAN (see Section 13.4.1.27) and SAVRAN (see Section 13.4.1.26).

x=RANS(0) returns a pseudo random floating-point number in the range of 0.LT.x.LT.1. RANS is a prime-modulus random number generator with shuffling capability. It calls RAN to generate its initial table of random deviates. Refer to related subroutins SETRAN (see Section 12.4.1.27) and SAVRAN (see Section 12.4.1.26).

y=SECNDS(x) returns the system time in seconds as a single-precision, floating-point value, minus the value of its single-precision, floating-point argument. The argument y is set equal to the time in seconds since midnight, minus the user-supplied value of x.

x=TIM2GO(0) returns the number of seconds remaining in the job's run-time limit. The time limit is set by the /TIME switch when submitting the batch job. The argument is a dummy (not used) and may be any number.

You may also specify a time limit for an interactive job by using the SET TIME-LIMIT command on TOPS-20, or the SET TIME command on TOPS-10.

FORTRAN-supplied external functions are treated in the same manner as user-defined functions. Implicit or explicit type declarations affect these functions, and no argument checking (type or number) is performed at compile time.

13.3.2 User-Defined External Functions

An external user-defined function is a procedure that is external to the program unit that references it. The function subprogram enables you to define a multiline function. By referencing the name of that function in an expression, the lines of the function are automatically executed.

The FUNCTION statement is always the first statement in a function subprogram. The form of the FUNCTION statement is:

```
     [type] FUNCTION fun ([arg1 [,arg2]...])
```

where:

type — is an optional type specification for the external function. This may be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER (plus the optional size modifier *len).

For CHARACTER, len is the length specification of the result of the character function. If you specify CHARACTER*(*), the function assumes the length declared for it in the program unit that invokes it. If len is an integer constant, the value of len must agree with the length of the function specified in the program unit that invokes the function. If a length is not specified in a CHARACTER FUNCTION statement, a length of one is assumed.

If you do not specify a type, the type of the function subprogram name determines the data type of the external function.

fun — is the symbolic name of the external function subprogram. Unless the optional data type is specified in the FUNCTION statement, the type of the function name determines the data type of the function subprogram.

arg — is an optional dummy argument. Arg may be a variable name, array name, or dummy procedure name. Separate multiple dummy arguments with commas. The parentheses are optional if no dummy arguments are specified.

You must define the symbolic name assigned a function subprogram as a variable name in the function. During each execution of the function, this variable can be redefined. The value of the variable at the time of execution of any RETURN statement is the value of the function.

NOTE

> The RETURN statement returns control to the statement that referenced the function subprogram (see Section 13.4.4). Additionally, you may desire to have a function start executing at a statement other than the first executable statement in the function subprogram. The ENTRY statement (see Section 13.4.3) enables you to define an alternate entry point in the function subprogram.

13.3.3 Function Subprogram Restrictions

The following rules govern the structuring of a function subprogram:

1. You can not use the symbolic name of a function subprogram in any nonexecutable statement in the subprogram except in the initial FUNCTION statement or a type statement.

2. Dummy argument names cannot appear in any EQUIVALENCE, COMMON, or DATA statement used within the subprogram.

3. The function subprogram can define or redefine one or more of its arguments so as to return results in addition to the value of the function.

4. The function subprogram can contain any FORTRAN statement except BLOCK DATA, SUBROUTINE, PROGRAM, or another FUNCTION statement.

5. The function subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statement signifies a logical conclusion of the computation made by the subprogram, and returns the computed function value and control to the calling program. A subprogram can have more than one RETURN statement.

   The END statement specifies the physical end of the subprogram and implies a return.

6. If the type of a function is specified in a FUNCTION statement, the function name must not appear in a type statement. Note that a name must not have its type explicitly specified more than once in a program unit.

7. A function specified in a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A function subprogram must not reference itself, either directly or indirectly.

8. If the name of a function subprogram is of type character, each entry name in the function subprogram must be of type character. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such items must have a length of (*) declared; otherwise, all such items must have a length specification of the same integer value.

13.3.4 Using a Function Subprogram

After defining a function subprogram, you use it by referencing the name of the function subprogram in an expression. Function subprograms are referenced in expressions using the following form:

```
     fun ([arg1,arg2,...argn])
```

where:

fun
: is the function subprogram name. This is the same name that appears in the corresponding FUNCTION statement. The length of the character function in a character function reference must be the same as the length of the character function in the referenced function.

arg is an optional list of actual arguments. These arguments must agree in type and number with the dummy argument list of the corresponding FUNCTION statement. If the actual and dummy arguments do not agree, no type conversion is done; and the results are unpredictable. The parentheses are required even if no actual arguments are specified.

When an expression that contains a function subprogram reference is executed, the following happens:

1. Evaluation of actual arguments that are expressions

2. Association of actual arguments with the corresponding dummy arguments

3. Execution of the referenced function subprogram

Section 13.3.3 describes all the restrictions that must be adhered to when using function subprograms.

13.4 SUBROUTINES

A subroutine subprogram is a procedure that is external to the program units that reference it. FORTRAN offers two types of subroutines: user-defined and FORTRAN-supplied. FORTRAN-supplied subroutines are described in Sections 13.4.1 - 13.4.1.32; user-defined subroutines are described in Sections 13.4.2 through 13.4.2.4.

NOTE

> The FORTRAN-supplied subroutines are described in this manual in two sections. Section 13.4.1 describes the general FORTRAN subroutines; Appendix F describes the subroutines that enable you to use a plotter.

Program units reference subroutines with the CALL statement (see Section 13.4.2.2). The subroutine reference in the CALL statement contains the unique name of the subroutine, as defined in a SUBROUTINE statement (see Section 13.4.2.1). The SUBROUTINE statement is always the first statement in a subroutine.

When a CALL statement reference is made to a subroutine subprogram, program execution transfers from that CALL statement to the referenced subroutine subprogram.

By including the ENTRY statement (see Section 13.4.3) within the body of a subroutine subprogram, you can enter the subroutine at a point other than the first statement in the subroutine. In this case, the CALL statement used to reference an entry point in a subroutine contains a reference to an entry point name, as opposed to the subroutine name.

Return of program control from the subroutine to the calling program unit occurs when the RETURN statement is executed (see Section 13.4.4). The RETURN statement is always the last statement executed in a subroutine subprogram.

Table 13–2: FORTRAN–Supplied Subroutines

Form	Function	Section
CALL ALCCHR(size)	Allocates space for dynamic character concatenation operations.	13.4.1.1
dpres = CDABS(dparg)	Returns the double-precision absolute value of the specified double-precision complex number.	13.4.1.2
CALL CDCOS(dparg,dpres)	Finds the complex cosine of the specified double-precision complex number.	13.4.1.3
CALL CDEXP(dparg,dpres)	Finds the complex exponential of the specified double-precision complex number.	13.4.1.4
CALL CDLOG(dparg,dpres)	Returns the complex logarithm of a specified double-precision complex number.	13.4.1.5
CALL CDSIN(dparg,dpres)	Returns the complex sine of the double-precision complex number specified.	13.4.1.6
CALL CDSQRT(dparg,dpres)	Returns the complex square root of the double precision complex number specified.	13.4.1.7
CALL CHKDIV(unitvar)	Returns the number of the unit to which error messages are being written.	13.4.1.8
CALL CLRFMT(arrayname)	Discards the FORMAT statement saved by the execution of the SAVFMT subroutine.	13.4.1.9
CALL DATE(name)	Places the current data, left-justified, in a character variable	13.4.1.10
CALL DIVERT(un)	Enables you to redirect error messages from the current device to an open file on a specified device.	13.4.1.11
CALL DTOGA(sname,dname,n)	Converts elements of double-precision arrays from D floating double-precision format to G floating double-precision format.	13.4.1.12
CALL DUMP(LB1,UB1,format1 [,...,LBn,UBn,formatn])	Causes specified portions of memory to be dumped to the line printer (LPT:).	13.4.1.13
CALL ERRSET(n) CALL ERRSET(n,1) CALL ERRSET(n,1,subr)	Controls the output of arithmetic error messages during program execution.	13.4.1.14
CALL ERRSNS(I) CALL ERRSNS(I,J) CALL ERRSNS(I,J,MSG)	Determines the reason for an error trapped by ERR= on an OPEN, CLOSE, or data transfer operation.	13.4.1.15
CALL EXIT	Terminates the program and returns control to the monitor.	13.4.1.16
CALL FFUNIT	Returns the number of the first available FORTRAN logical unit.	13.4.1.17
CALL GTODA(sname,dname,n)	Converts elements of double-precision arrays from G floating double-precision format to D floating double-precision format.	13.4.1.18
CALL ILL	Sets the ILLEG flag.	13.4.1.19
CALL LEGAL	Clears the ILLEG flag.	13.4.1.20
CALL MVBITS(m,i,len,n,j)	Transfers a bit field from one storage location to a second storage location.	13.4.1.21
CALL OVERFL(IANS)	Returns information about overflow, underflow, and divide check.	13.4.1.22
CALL PDUMP(LB1,UB1,format1 [,...,LBn,UBn,formatn])	Functions exactly like DUMP subroutine except that control returns to the calling program after the dump has been executed.	13.4.1.23
QUIETX	Suppresses all summary type out when the program terminates.	13.4.1.24

Table 13-2: FORTRAN-Supplied Subroutines (Cont.)

Form	Function	Section
CALL SAVFMT(name[,arraysize])	Directs FOROTS to encode format specifications contained in the specified character variable or array.	13.4.1.25
CALL SAVRAN(n)	Saves the last internal integer seed value generated by the RAN function.	13.4.1.26
CALL SETRAN(n)	Specifies the internal integer seed value for the RAN function.	13.4.1.27
CALL SORT('sort string')	Sorts one or more files using the SORT program.	13.4.1.28
CALL SRTINI(n)	Directs FOROTS to start allocating memory from top downward to account for large overlay programs and preallocates pages 600:677 for SORT.	13.4.1.29
CALL TIME(x) CALL TIME(x,y)	Returns the current time of day in left-justified ASCII.	13.4.1.30
CALL TOPMEM(n)	Directs FOROTS to start allocating memory from top downward to account for large overlay programs.	13.4.1.31
CALL TRACE	Generates a list of active subprograms on the terminal.	13.4.1.32

13.4.1 FORTRAN-Supplied Subroutines

The FORTRAN software includes a set of predefined subroutines. Th section describes the general FORTRAN subroutines (and a function th is similar) in alphabetical order. (See Appendix F for th FORTRAN-supplied plotter subroutines.)

NOTE

> Sections 13.4.1.2 through 13.4.1.7 describe subroutines (and a function) that are used for calculations on double-precision complex numbers. You must supply your own subroutines for performing addition, subtraction, multiplication, and division of double-precision complex numbers.
>
> In addition, FORTRAN does not support the double-precision complex data type (called COMPLEX*16). These numbers are kept as arrays and cannot be used in expressions or as the arguments of generic routines.

ALCCHR Subroutine

13.4.1.1 ALCCHR Subroutine - The ALCCHR subroutine allocates space for dynamic character concatenation operations. You do not normally need to allocate space for this purpose unless you are doing very large character concatenation operations.

The form of the ALCCHR subroutine is:

```
CALL ALCCHR(size)
```

where:

size — is the integer size in characters for either creating or expanding the character stack.

CDABS
Function

13.4.1.2 CDABS Function - The CDABS function returns the double-precision absolute value of the specified double-precision complex number. (Although CDABS is a function and is not a subroutine, it is included here because it is similar to some subroutines.)

The form of the CDABS function is:

```
dpres = CDABS(dparg)
```

where:

dparg — is a 2-element double-precision array containing the complex value whose absolute value you want calculated. The first element of dparg contains the real part of the double-precision complex number; the second element contains the imaginary part.

dpres — is a double-precision variable that is set to the absolute value of the complex number.

CDCOS
Subroutine

13.4.1.3 CDCOS Subroutine - The CDCOS subroutine finds the complex cosine of the specified double-precision complex number.

The form of the CDCOS subroutine is:

```
CALL CDCOS(dparg,dpres)
```

where:

dparg — is a 2-element double-precision array containing the complex value whose cosine you want calculated. The first element of dparg contains the real part of the double-precision complex number; the second element contains the imaginary part.

dpres   is a 2-element double-precision array in which the subroutine returns the result of the calculation. The first element of dpres contains the real part of the double-precision complex number; the second element contains the imaginary part.

Example:

```
     DOUBLE PRECISION dparg(2),dpres(2)
     dparg(1) = 1D0                !arg is (1,-1)
     dparg(2) = -1D0
     CALL CDCOS(dparg,dpres)
```

CDEXP
Subroutine

13.4.1.4 CDEXP Subroutine - The CDEXP subroutine finds the complex exponential of the specified double-precision complex number.

The form of the CDEXP subroutine is:

```
     CALL CDEXP(dparg,dpres)
```

where:

dparg   is a 2-element double-precision array that contains the complex argument to the subroutine. The first element of dparg contains the real part of the double-precision complex number; the second element contains the imaginary part.

dpres   is a 2-element double-precision array that stores the result of the calculation. The first element of dpres stores the real part of the result; the second element stores the imaginary part.

Example:

```
     DOUBLE PRECISION dparg(2),dpres(2)
     dparg(1) = 0D0
     dparg(2) = 1D0                !arg is (0,1)
     CALL CDEXP(dparg,dpres)
```

CDLOG
Subroutine

13.4.1.5 CDLOG Subroutine - The CDLOG subroutine returns the complex logarithm of a specified double-precision complex number.

The form of the CDLOG subroutine is:

```
CALL CDLOG(dparg,dpres)
```

where:

dparg is a 2-element double-precision array that contains the double-precision complex number whose logarithm you want calculated. The first element of dparg contains the real part of the complex number; the second element contains the imaginary part.

dpres is a 2-element double-precision array that stores the result returned by CDLOG. The first element of dpres contains the real part of the double-precision complex number; the second element contains the imaginary part.

Example:

```
DOUBLE PRECISION dparg(2),dpres(2)
dparg(1) = 1D0                  !arg is (1,0)
dparg(2) = 0D0
CALL CDLOG(dparg,dpres)
```

CDSIN
Subroutine

13.4.1.6 CDSIN Subroutine

The CDSIN subroutine returns the complex sine of the double-precision complex number specified.

The form of the CDSIN subroutine is:

```
CALL CDSIN(dparg,dpres)
```

where:

dparg is a 2-element double-precision array that contains the number whose sine you want calculated. The first element of dparg contains the real part of the double-precision complex number; the second element contains the imaginary part.

dpres is a 2-element double-precision array in which the result of the calculation is returned. The first element of dpres contains the real part of the result; the second element contains the imaginary part.

Example:

```
DOUBLE PRECISION dparg(2),dpres(2)
dparg(1) = -1D0                 !arg is (-1,01)
dparg(2) = 1D0
CALL CDSIN(dparg,dpres)
```

CDSQRT Subroutine

13.4.1.7 CDSQRT Subroutine - The CDSQRT subroutine returns the complex square root of the specified double-precision complex number.

The form of the CDSQRT subroutine is:

```
     CALL CDSQRT(dparg,dpres)
```

where:

dparg is a 2-element double-precision array that contains the double-precision complex number whose square root you want calculated. The first element of dparg contains the real part of the double-precision complex number; the second element contains the imaginary part.

dpres is a 2-element double-precision array that contains the result of the calculation. The first element of dpres contains the real part of the complex square root; the second element contains the imaginary part.

Example:

```
     DOUBLE PRECISION dparg(2),dpres(2)
     dparg(1) = 10D0
     dpres(2) = -10D0                  !arg is (10,-10)
     CALL CDSQRT(dparg,dpres)
```

CHKDIV Subroutine

13.4.1.8 CHKDIV Subroutine - The CHKDIV subroutine returns the number of the unit to which error messages are being written. This subroutine returns the value -1 if the messages are being sent to the terminal.

The form of the CHKDIV subroutine is:

```
     CALL CHKDIV(unitvar)
```

where:

unitvar is the variable in which the unit number is stored.

CLRFMT
Subroutine

13.4.1.9 CLRFMT Subroutine - The CLRFMT subroutine discards the encoded form of the FORMAT statement saved by the execution of the SAVFMT subroutine (see Section 13.4.1.25).

The form of the CLRFMT subroutine is:

```
     CALL CLRFMT(arrayname)
```

where:

arrayname   is the name of the array that contains the encoded form of the FORMAT specifications saved by the SAVFMT subroutine.

DATE
Subroutine

13.4.1.10 DATE Subroutine - The DATE subroutine stores the current date as a left-justified ASCII string in a character variable. The date is in the form:

```
     dd-mmm-yyb
```

where:

dd   is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-letter month abbreviation (such as, Jan,Feb), yy is a 2-letter year, and b is a blank. The data is stored in ASCII, left-justified, and blank filled.

The form of the DATE subroutine is:

```
     CALL DATE(name)
```

where:

name   is the name of a character variable. The date returned by the subroutine is stored in this variable.

NOTE

For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric array name as the argument for the DATE subroutine. The current date is stored as a left-justified ASCII string in a 2-word array or data item (double-precision or complex). The array must have at least two elements.

DIVERT Subroutine

13.4.1.11 DIVERT Subroutine - The DIVERT subroutine enables you to redirect error messages from the current device to an open file on a specified device.

The form of the DIVERT subroutine is:

```
CALL DIVERT(un)
```

where:

un is the logical device number of the file on which the open file resides.

DTOGA Subroutine

13.4.1.12 DTOGA - The DTOGA subroutine converts elements of double-precision arrays from D-floating double-precision format to G-floating double-precision format.

The form of the DTOGA subroutine is:

```
CALL DTOGA (sname,dname,n)
```

where:

sname is the name of the source array.

dname is the name of the destination array.

n is the number of elements to convert.

(See Section 13.4.1.18 for the GTODA subroutine.)

DUMP Subroutine

13.4.1.13 DUMP Subroutine - The DUMP subroutine causes specified portions of memory to be dumped to the line printer (LPT:).

The form of the DUMP subroutine is:

```
     CALL DUMP(LB1,UB1,format1[...,LBn,UBn,formatn])
```

where:

LB1,UB1 — are the integer values of the upper and lower memory addresses to be dumped.

format1 — is an integer that indicates the dump format. The possible specifications are:

0 = octal
1 = real
2 = integer
3 = ASCII

If no arguments are supplied, all of user memory is dumped in octal. If only the bounds arguments are specified, or if the format value is out of range, the dump format is octal. If only the first bound argument is specified, all locations from that address to the end of memory are dumped.

The dump is terminated by a call to EXIT.

ERRSET
Subroutine

13.4.1.14 ERRSET Subroutine - The ERRSET subroutine controls the output of warning messages during program execution.

The ERRSET subroutine has three forms:

1. CALL ERRSET(n)

2. CALL ERRSET(n,i)

3. CALL ERRSET(n,i,subr)

where:

n — is the maximum number of error messages to type.

i — is the code to which error the call applies; one of the following:

-1 any of the following
0 integer overflow
1 integer divide
2 input integer overflow
3 input floating overflow
4 floating overflow
5 floating divide check
6 floating underflow
7 input floating underflow
8 library routine error

9	output field width too small
21	FORLIB warnings
22	nonstandard usage warnings
23	Bounds check warnings

if i is not specified, -1 is assumed

subr is the name of the user-defined error handling routine to be invoked each time any of the above errors occur. The effect is as if

```
CALL SUBR (I,IPC,N2,ITYPE,UNFIXD,FIXED)
```

were placed in the program just after the instruction causing the trap.

I = error number of trap, same as above

IPC = PC of trap instruction (if code 9 is trapped, IPC = PC of the IOLST. call

N2 = second error number (reserved for Digital)

ITYPE = data type of value

UNFIXD = value returned by the hardware

FIXED = value after fixup (SUBR can change this value)

If SUBR is not specified, no routine is called on the APR trap.

ERRSNS Subroutine

13.4.1.15 ERRSNS Subroutine - The ERRSNS subroutine returns integer values that describe the status (success or failure) of the last I/O operation (see Appendix D). This subroutine can be used to determine the reason for an error trapped by ERR= on an OPEN, CLOSE, or data transfer operation. Both return values are always cleared after a successful data transfer operation.

The forms of the ERRSNS subroutine are:

```
CALL ERRSNS (I)
```

or

```
CALL ERRSNS (I,J)
```

or

```
CALL ERRSNS (I,J,MSG)
```

where:

I returns a FORTRAN-supplied number that describes the class of failure that occurred.

J optionally returns a processor-specific number that further describes or qualifies the type of the last error.

MSG If present, is a character variable used to return the ASCII text of the last error message. If the variable for MSG is less than 80 characters, the text is truncated; if the variable is greater than 80 characters, the text is padded to the right with blanks.

NOTE

For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric array as the MSG argument. The numeric array is used as a 16-word array to return the ASCII text of the last error message.

EXIT Subroutine

13.4.1.16 EXIT Subroutine - The EXIT subroutine terminates the program and returns control to the monitor. The EXIT subroutine takes no arguments.

The form of the EXIT subroutine is:

```
CALL EXIT
```

FFUNIT Subroutine

13.4.1.17 FFUNIT Subroutine - The FFUNIT subroutine returns the lowest available FORTRAN logical unit number (see Table 10-3).

The form of the FFUNIT subroutine is:

```
CALL FFUNIT (n)
```

GTODA
Subroutine

13.4.1.18 GTODA Subroutine - The GTODA subroutine converts elements of double-precision arrays from G-floating double-precision format to D-floating double-precision format.

The form of the GTODA subroutine is:

```
     CALL GTODA(sname,dname,n)
```

where:

sname   is the source array name.

dname   is the destination array name.

n   is the number of array elements to convert.

(See Section 13.4.1.12 for the DTOGA subroutine.)

ILL
Subroutine

13.4.1.19 ILL Subroutine - The ILL subroutine sets the ILLEG flag. If this flag is set and an illegal character is encountered in floating-point, double-precision input, the corresponding value is set to zero and no error message is issued. The ILL subroutine takes no arguments. The ILLEG flag is not set initially.

The form of the ILL subroutine is:

```
     CALL ILL
```

LEGAL
Subroutine

13.4.1.20 LEGAL Subroutine - The LEGAL subroutine clears the ILLEG flag set by the ILL subroutine. The LEGAL subroutine takes no arguments.

The form of the LEGAL subroutine is:

```
     CALL LEGAL
```

MVBITS Subroutine

13.4.1.21 **MVBITS Subroutine** - The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination).

The form of the MVBITS subroutine is:

```
CALL MVBITS(m,i,len,n,j)
```

where:

m   is an integer expression that represents the source location from which a bit field is transferred.

i   is an integer expression that identifies the first bit position in the source field transferred from m.

len is an integer expression that identifies the length of the field transferred from m.

n   is an integer variable or array element that is the destination location to which a bit field is transferred.

j   is an integer expression that identifies the first bit position in the destination field transferred from m.

The MVBITS subroutine transfers len bits from position i through i+len-1 of the source location (m) to positions j through j+len-1 of the destination location (n). Other bits of the destination location and all of the bits of the source location remain unchanged. The values of i+len and j+len must not be greater than 36.

(See Section 13.1.4 for information on bit manipulation functions.)

OVERFL Subroutine

13.4.1.22 OVERFL Subroutine - The OVERFL subroutine returns information about overflow, underflow, and divide check.

The form of the OVERFL subroutine is:

```
CALL OVERFL(IANS)
```

where:

IANS is an integer variable whose value specifies whether an overflow, underflow, or divide check has occurred since the last call to OVERFL. The values returned are:

1 = at least one overflow, underflow, or divide check occurred.

2 = none occurred.

PDUMP Subroutine

13.4.1.23 **PDUMP Subroutine** - The PDUMP subroutine functions exactly like the DUMP subroutine (see Section 13.4.1.13) except that control returns to the calling program after the dump has been executed.

The form of the PDUMP subroutine is:

```
     CALL PDUMP(LB1,UB1,format1[...,LBn,UBn,formatn])
```

where:

LB1,UB1 are the integer values of the upper and lower memory addresses to be dumped.

format1 is an integer that indicates the dump format. The possible specifications are:

0 = octal
1 = real
2 = integer
3 = ASCII

If no arguments are supplied, all of user memory is dumped in octal. If only the bounds arguments are specified, or if the format value is out of range, the dump format is octal. If only the first bound argument is specified, all locations from that address to the end of memory are dumped.

QUIETX Subroutine

13.4.1.24 **QUIETX Subroutine** - The QUIETX subroutine suppresses all summary typeout when the program terminates, including library error summaries and CPU times. The QUIETX subroutine takes no arguments.

The form of the QUIETX subroutine is:

```
     CALL QUIETX
```

SAVFMT
Subroutine

13.4.1.25 **SAVFMT Subroutine** -- The SAVFMT subroutine directs FOROTS to encode the FORMAT specifications contained in the specified character variable or array, and to save the encoded form. This action improves the performance of any future I/O statements which that character variable or array is the format identifier.

FOROTS saves the encoded form of the format until a call to CLRFMT (see Section 13.4.1.9) is executed for that variable or array, or until another call to SAVFMT is executed for that variable or array.

NOTE

> After a call to SAVFMT, you must not change the value of the variable or array. If the value is changed, the new value may be ignored. A call to SAVFMT with a variable or array whose address is identical to that in a previous call, does an implied call to CLRFMT.

The form of the SAVFMT subroutine is:

```
     CALL SAVFMT(name[,arraysize])
```

where:

name	is the name of the character variable or array that contains the FORMAT descriptors that you want encoded.
arraysize	is the number of array elements if an array is specified.

NOTE

> For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric array name as the argument for the SAVFMT subroutine.

SAVRAN
Subroutine

13.4.1.26 **SAVRAN Subroutine** -- The SAVRAN subroutine saves the last internal integer seed value generated by the RAN function. The RAN function, described in Section 13.3.1, returns a random number each time it is called. This value can be used at a later time in a call to SETRAN to reestablish the same random number sequence.

The form of the SAVRAN subroutine is:

```
CALL SAVRAN(n)
```

where:

n is an integer variable into which the SAVRAN subroutine stores the last internal integer seed value generated.

SETRAN Subroutine

13.4.1.27 SETRAN Subroutine - The SETRAN subroutine specifies the internal integer seed value for the RAN function. If the SETRAN argument is zero, RAN uses its own default starting value.

The form of the SETRAN subroutine is:

```
CALL SETRAN(n)
```

where:

n is a nonnegative integer constant or variable (less than 2**31).

SORT Subroutine

13.4.1.28 SORT Subroutine - The SORT subroutine sorts one or more files using the SORT program. You can successfully use this subroutine only if the SORT software has been installed (see the *FORTRAN Installation Guide*). The SORT software is sold as a separate product and may not be available at your installation.

The form of the SORT subroutine is:

```
CALL SORT('sort string')
```

where:

sort string is a command line containing file specifications and SORT switches. For specific information about the SORT command line, see the *SORT/MERGE User's Guide*.

NOTE

> The sort string must be compatible with the current version of the standalone SORT. Therefore, the string is not the same for TOPS-10 and TOPS-20 (see the SORT/MERGE User's Guide).

SRTINI Subroutine

13.4.1.29 SRTINI Subroutine

- The SRTINI subroutine directs FOROTS to start allocating memory from top downward to account for large overlay programs and preallocates pages 600:677 (octal) for SORT.

The form of the SRTINI routine is:

```
     CALL SRTINI(n)
```

where:

n is top page number to use when allocating memory.

Note that FOROTS will not allocate pages (including DDT pages) that have been marked as unavailable at memory initialization.

TIME Subroutine

13.4.1.30 TIME Subroutine

- The TIME subroutine returns the current time of day in left-justified ASCII.

The TIME subroutine has two forms:

```
     CALL TIME(x)

     or

     CALL TIME(x,y)
```

where:

x is a character variable. In the one argument form, TIME returns the time in x in the form: hh:mm, where hh is the hour (24-hour time) and mm is the minutes.

y is a character variable. When the two argument form of the TIME subroutine is used, the forms of the time returned in x is the same as the one argument form, and the value returned in y has the form: bss.t, where b is a blank, ss is the current seconds, and t is the current tenths of seconds.

NOTE

> For compatibility with previous versions of FORTRAN-10/20, you can specify a numeric variable or array element as an argument of the TIME subroutine.

The following example demonstrates using the one and the two argument forms of the TIME subroutine in a program.

```
          PROGRAM TIMTST
          CHARACTER*10 X,Y

          CALL TIME(X,Y)

          WRITE(UNIT=5,FMT=101)X,Y

          CALL TIME(X)

          WRITE (UNIT=5, FMT=102)X
    102   FORMAT(1X,'THE ONE ARGUMENT TIME RETURNS: ',A)

          END
```

```
    EXECUTE TIMTST
    LINK:   Loading
    [LNKXCT TIMTST execution]
    THE TWO ARGUMENT TIME RETURNS: 09:00 20.9
    THE ONE ARGUMENT TIME RETURNS: 09:00
    CPU time 0.1   Elapsed time 0.2
```

TOPMEM Subroutine

13.4.1.31 **TOPMEM Subroutine** - The TOPMEM subroutine directs FOROTS to start allocating memory from a specified page number downward to account for large overlay programs.

The form of the TOPMEM subroutine is:

```
    CALL TOPMEM(n)
```

where:

n   is the top page number to use in allocating memory.

Note that FOROTS will not allocate pages (including DDT pages) that have been marked as unavailable at memory initialization.

13.4.1.32 TRACE Subroutine - The TRACE subroutine generates a list of active subprograms on the terminal. An active subprogram is one that has been called but has not yet returned. The main program is always active. The trace listing starts at the currently active routine (the one containing the call to TRACE) and proceeds back to the main program.

The form of the TRACE subroutine is:

```
CALL TRACE
```

The information produced by the TRACE routine consists of, for each subprogram:

1. The name of the routine

2. The address of the routine (in octal)

3. The address of the subprogram call (expressed as routine-name + offset)

4. The address of the subprogram call (expressed as label + offset with the calling routine)

5. The number of arguments passed to the called routine

6. The types of the arguments passed including:

   C - Character string descriptor
   D - D-floating double precision
   F - Real
   G - G-floating double precision
   I - Integer or double integer
   K - Literal string
   L - Logical
   O - Octal or double octal
   S - Statement label
   U - Unknown argument type
   X - Complex

If there are too many arguments to display, the 'types' column will contain '...'.

If local symbols are loaded with the program, the label+offset information will be much more informative. A label of the form 12345P refers to FORTRAN statement number 12345; a label of the form 56789L refers to line number 56789 in the compiler listing. Line number labels only appear if the program was compiled with /DEBUG:LABELS (see Chapter 16).

The traceback listing is sent to the error-message unit, which is normally the terminal. You can use the DIVERT subroutine (Section 13.4.1.11) to change where the listing is sent.

Example:

```
              PROGRAM MAIN
              TYPE 10
       10     FORMAT (/' Calling SUB1:')
              CALL SUB1
              TYPE 20
       20     FORMAT (/' Calling SUB2:')
              CALL SUB2 (A,B)
              END

              SUBROUTINE SUB1
              Y = F(X)
              END

              SUBROUTINE SUB2 (G,H)
              CALL SUB3 (G,H,I)
              END

              SUBROUTINE SUB3 (A1,A2,A3)
              CALL TRACE
              END

              FUNCTION F(Y)
              CALL SUB2 (Y,0)
              F=2.
              END

EXECUTE TRC.FOR
LINK:  Loading
[LNKXCT TRC execution]

Calling SUB1:

Name    (Loc)     <<---  Caller        (Loc)          Args    Types
TRACE.  (426373)  <<---  SUB3+2        (SUB3+2)        0
SUB3    (256)     <<---  SUB2+6        (SUB2+6)        3       FFI
SUB2    (232)     <<---  F+20          (F+20)          2       FI
F       (307)     <<---  SUB1+2        (SUB1+2)        1       F
SUB1    (214)     <<---  MAIN.+7       (MAIN.+7)       0

Calling SUB2:

Name    (Loc)     <<---  Caller        (Loc)          Args    Types
TRACE.  (426373)  <<---  SUB3+2        (SUB3+2)        0
SUB3    (256)     <<---  SUB2+6        (SUB2+6)        3       FFI
SUB2    (232)     <<---  MAIN.+14      (MAIN.+14)      2       FF

CPU time 0.4  Elapsed time 3.1
```

13.4.2 User-Defined Subroutines

A subroutine subprogram is a separate program unit. The FORTRAN CALL statement is used in a program unit to call a subroutine subprogram. The CALL statement contains the name of the subroutine to which control passes when the CALL statement is executed. The CALL statement can also optionally contain actual arguments that are passed to the called subroutine. The CALL statement is described in Section 13.4.2.2.

The SUBROUTINE statement is always the first statement in a subroutine subprogram. The SUBROUTINE statement defines the name and, optionally, any dummy arguments used by the subroutine. The SUBROUTINE statement is described in Section 13.4.2.1.

The ENTRY statement enables you to enter a subroutine subprogram at a statement other than the first statement of the subroutine. The ENTRY statement is described in Section 13.4.3.

The last logical statement of a subroutine subprogram is always the RETURN statement. By default, the RETURN statement always returns control to the first executable statement in the calling program that immediately follows the CALL statement. Optionally, you may use the alternate return form of the RETURN statement to return control to a statement other than the default. Both forms of the RETURN statement are described in Section 13.4.4.

13.4.2.1 **SUBROUTINE Statement** - The SUBROUTINE statement is always the first statement in a subroutine subprogram. It is used to define the name of the subroutine and, optionally, to define dummy arguments that are used by the subroutine.

The form of the SUBROUTINE statement is:

```
SUBROUTINE sub [([d1[,d2]...])]
```

where:

sub is the symbolic name of the subroutine or dummy procedure.

d is an optional dummy argument for the subroutine subprogram. This argument can be a variable name, an array name, a dummy procedure name, or any combination of these separated by commas. The parentheses are optional if no dummy arguments are specified.

The following rules control the structuring of a subroutine subprogram:

1. You can not use the symbolic name of the subprogram in any statement within the defined subprogram except the SUBROUTINE statement itself. The symbolic name of a subroutine is a global name and must not be the same as any other global name or any local name in the program unit.

2. The symbolic name of a dummy argument is local to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, COMMON, or DATA statement except as a common block name.

3. The subroutine subprogram may define or redefine one or more of its dummy arguments so as to return results.

4. If the actual argument is a constant or expression, the subroutine must not change the value of that argument.

5. The subroutine subprogram may contain any FORTRAN statement except BLOCK DATA, FUNCTION, PROGRAM, another SUBROUTINE statement, or any statement that either directly or indirectly references the subroutine being defined or any of the subprograms in the chain of subprogram references leading to this subroutine.

6. Dummy arguments that represent statement labels may be either an *, $, or &.

7. The subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statement indicates a logical end of the routine; the END statement signifies the physical end of the subroutine. If no RETURN precedes the END statement, then the RETURN statement is implicit.

8. Subroutine subprograms can have as many entry points as desired (see description of ENTRY statement given in Section 13.4.3).

9. A character dummy argument whose length specification is an asterisk in parentheses may appear as an operand for concatenation.

13.4.2.2 **CALL Statement** - The CALL statement is used in a program unit to reference a subroutine. Execution of the CALL statement causes a transfer of program control to the subroutine referenced in the CALL statement.

The CALL statement can also contain a list of arguments that is used by the computation performed in the referenced subroutine.

The form of the CALL statement is:

```
     CALL sub [([a1[,a2]...])]
```

where:

sub is the symbolic name of a subroutine or dummy procedure.

a is an optional actual argument that is used by the subroutine. The actual arguments in the CALL statement must agree in position and type with the dummy arguments in the referenced SUBROUTINE statement. The parentheses are optional if no actual arguments are specified.

13.4.2.3 **Execution of a CALL Statement** - When a CALL statement is executed, the following results occur:

1. Any actual arguments in the CALL statement argument list that are expressions are evaluated.

2. The actual arguments are then associated with the dummy arguments in the referenced SUBROUTINE statement.

3. Control passes to the subroutine and it is executed.

4. Return of control from the referenced subroutine to the calling program completes the execution of the CALL statement.

A subroutine can be referenced within any other procedure or the main program of the executable program. A subprogram cannot, however, reference itself, either directly or indirectly.

13.4.2.4 Actual Arguments for a Subroutine - Actual arguments can appear in the CALL statement argument list. Actual arguments must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The exception to the agreement rules between actual and dummy argument lists is the use of a subroutine name or an alternate return specifier as an actual argument.

Actual arguments in CALL statements can be any of the following:

1. Any expression, including a character expression whose length specification is an asterisk in parentheses.

2. An array name

3. An intrinsic function name

4. An external procedure name

5. A dummy procedure name

6. An alternate return label

An actual argument in a subroutine reference may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument must not be used as an actual argument in a subprogram reference.

13.4.3 ENTRY Statement

The ENTRY statement provides you with a method for entering a function or subroutine subprogram at any executable statement. The ENTRY statement can appear anywhere within a function subprogram after the FUNCTION statement, or within a subroutine subprogram after the SUBROUTINE statement. An ENTRY statement cannot appear between a DO statement and the terminal statement of its DO-loop or inside a block-IF statement.

A subprogram may optionally have one or more ENTRY statements. An ENTRY statement is classified as a nonexecutable statement.

The form of the ENTRY statement is:

```
     ENTRY en [(dl [,d2...])]
```

where:

en is the symbolic name of an entry in a function or subroutine subprogram. This symbolic name is called an entry name. If the entry name appears in a subroutine subprogram, it is referred to as a subroutine name; if the entry name appears in a function subprogram, it is called an external function name.

d is a variable name, array name, dummy procedure name, or the symbols: *, $, or & (these symbols can represent a dummy argument which is an alternate return label). The symbol references (asterisk, ampersand, and dollar sign) are permitted only when the ENTRY statement appears in a subroutine subprogram.

If you do not specify any parentheses after the entry name, you need not specify any dummy arguments. If, however, you include the parentheses, you must specify at least one dummy argument. The rules for the use of an ENTRY statement follow:

1. The ENTRY statement allows entry into a subprogram at a place other than that defined by the SUBROUTINE or FUNCTION statement. You may include more than one ENTRY statement in an external subprogram.

2. Execution begins at the first executable statement following the ENTRY statement.

3. Appearance of an ENTRY statement in a subprogram does not negate the rule that statement functions in subprograms must precede the first executable statement.

4. ENTRY statements are nonexecutable and do not affect the execution flow of a subprogram.

5. You can not use an ENTRY statement in a main program or have a subprogram reference itself through its entry points.

6. You can not use an ENTRY statement in the range of a DO statement construction.

7. The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in SUBROUTINE or FUNCTION statements or any other ENTRY statement in the subprogram. However, the arguments for each call or function reference must agree with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that is referenced.

8. Entry into a subprogram initializes only the dummy arguments of the referenced ENTRY statement.

9. You can not reference a dummy argument unless it appears in the dummy list of the ENTRY, SUBROUTINE, or FUNCTION statement by which the subprogram is entered.

10. The source subprogram must be ordered such that references to dummy arguments in executable statements follow the appearance of the dummy argument in the dummy list of a SUBROUTINE, FUNCTION, or ENTRY statement.

11. Dummy arguments that were defined for a subprogram by some previous reference to the subprogram are undefined for subsequent entry into the subprogram.

12. The value of a function must be returned by use of the current entry name.

13. 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. If the name of the function subprogram or any entry in the subprogram has a length of (*) declared, all such items must have a length specification of the same integer value.

13.4.4 RETURN Statement

The RETURN statement returns control to the referencing program unit and may appear only in a function subprogram or a subroutine subprogram.

The RETURN statement has two possible forms, depending on whether it is the last statement in a function subprogram or a subroutine subprogram.

The form of the RETURN statement in a function subprogram is:

```
RETURN
```

The form of the RETURN statement in a subroutine subprogram is:

```
RETURN [e]
```

where:

e is an integer constant, variable, or expression. This form of the RETURN statement is called an alternate return. The alternate return form enables you to select any labeled statement in the calling program unit as a return point after execution of the program unit in which the alternate RETURN statement appears.

The value of e should be a positive integer that is equal to or less than the number of statement labels given in the argument list of the calling statement. If e is less than 1 or is larger than the number of available statement labels, a standard return is performed. (A standard return transfers control back to the first executable statement immediately following the calling statement in the calling program unit).

NOTE

A dummy argument for a statement label must be either an asterisk (*), a dollar sign ($), or an ampersand (&).

You may use more than one RETURN (standard return) statement in any subprogram. The use of the alternate return form of the RETURN statement is restricted to subroutine subprograms.

For example, assume the following statement sequence in a main program:

```
          CALL EXAMP(1,*10,K,*15,M,*20)
          GO TO 101
          .
          .
          .
     10   alternate return #1
          .
          .
          .
     15   alternate return #2
          .
          .
          .
     20   alternate return #3
          .
          .
          .
          END

          SUBROUTINE EXAMP (L, *,M, *,N,*)
          .
          .
          .
          RETURN
          .
          .
          .
          RETURN
          .
          .
          .
          RETURN(I+J)
          .
          .
          .
          END
```

Each occurrence of RETURN returns control to the statement GO TO 101 in the calling program.

If, on the execution of the RETURN(I+J) statement, the value of (I+J) is:

Value	The following is performed:
less than one	a standard return to the GO TO 101 statement is made
1	the return is made to statement 10
2	the return is made to statement 15
3	the return is made to statement 20
Greater than 3	a standard return to the GO TO 101 statement is made.

13.4.5 Dummy and Actual Arguments

Since you may reference subprograms at more than one point throughout a program, many of the values used by the subprogram may be changed each time it is used. Dummy arguments in subprograms represent the actual values to be used, which are passed to the subprogram when it is called.

For example, shown below is a subroutine (TEST) being called from the main program by a CALL statement. In this example, the variables in the CALL statement, A, B, and C(2), represent actual values in the main program. They are therefore called actual arguments.

On the other hand, the variables in the SUBROUTINE statement, R, X, and Z, do not represent any values until they have values passed to them from the CALL statement. They are therefore called dummy arguments.

(The CALL, SUBROUTINE, and RETURN statements are described in Sections 13.4.2.2, 13.4.2.1, and 13.4.4, respectively.)

```
CALL TEST (A,B,C(2))
    .
    .
    .
    END
SUBROUTINE TEST (R,X,Z)
    .
    .
    .
RETURN
END
```

Functions and subroutines use dummy arguments to indicate the type of the actual arguments they represent and whether the actual arguments are variables, arrays, subroutine names, or the names of external functions. Each dummy argument must be used within a function or subroutine as if it were a variable, array, subroutine, or external function identifier.

Dummy arguments are given in an argument list associated with the identifier assigned to the subprogram; actual arguments are normally given in an argument list associated with a call made to the desired subprogram.

The position, number, and type of each dummy argument in a subprogram list must agree with the position, number, and type of each argument list of the subprogram reference.

NOTE

> If the /DEBUG:ARGUMENTS compiler switch is specified (see Section 16.3), optional type checking is performed at load time on dummy and actual arguments.

Dummy arguments may be:

1. variables

2. array names

3. subroutine identifiers

4. function identifiers

5. the symbols *, $, or & that are used to denote the position of alternate return labels

When you reference a subprogram, its dummy arguments are replaced by the corresponding actual arguments supplied in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and character constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the results of the subprogram computations will be unpredictable.

Argument association may be carried through more than one level of subprogram reference if a valid association is maintained through each level. The dummy/actual argument associations established when a subprogram is referenced are terminated when the desired subprogram operations are completed.

The following rules govern the use and form of dummy arguments:

1. The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time the procedure is referenced.

2. Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.

3. A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding argument.

4. An array dummy argument should have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. Each element of a dummy array is associated directly with the corresponding elements of the actual array.

5. A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.

6. A dummy argument representing an external function must have an external function as its actual argument.

7. A dummy argument may be defined or redefined in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then elements of the array may be redefined.

Additional information regarding the use of dummy and actual arguments is given in the description of how subprograms are defined and referenced.

13.4.5.1 **Length of Character Dummy and Actual Arguments** - The length of a dummy argument of type character must not be greater than the length of its associated actual argument. Note that if the character dummy argument's length is specified as *(*), the length used is exactly the length of the associated actual argument. This is known as a passed length character argument.

The length of the dummy argument is determined each time control transfers to the function. The length of the actual argument can be the length of a character variable, array element, substring, or expression.

A character array dummy argument can have passed length. The length of each element in the dummy argument is the length of the elements in the actual argument. The passed length and the array declaractor together determine the size of the passed length character array.

The following example of a function subprogram uses a passed length character argument. The function finds the position of the character with the highest ASCII code value; it uses the length of the passed length character argument to control the iteration. (Note that the intrinsic function LEN is used to determine the length of the argument. See Table 13-1 for a description of the LEN function.)

```
          INTEGER FUNCTION ICMAX(CVAR)
          CHARACTER*(*) CVAR
          ICMAX = 1
          DO 10 I = 2, LEN(CVAR)
     10   IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX =I
          RETURN
          END
```

Each of the following function references specifies a different length for the dummy argument:

```
     CHARACTER VAR*10, CARRAY(3,5)*20
          .
          .
          .

     I1 = ICMAX(VAR)
     I2 = ICMAX(CARRAY(2,2))
     I3 = ICMAX(VAR(3:8))
     I4 = ICMAX(CARRAY(1,3)(5:15))
     I5 = ICMAX (VAR(3:4)//CARRAY(3,5))
```

13.4.5.2 **Character and Hollerith Constants as Actual Arguments** - Actual arguments and their corresponding dummy arguments must agree in data type. If the actual argument is a Hollerith constant (for example, 4HABCD), the dummy argument must be of numeric data type.

In FORTRAN-10/20, if an actual argument is a character constant (for example 'ABCD'), the corresponding dummy argument can have either numeric or character data type. If the dummy argument has a numeric data type, the character constant 'ABCD' is, in effect, converted to a Hollerith constant by the FORTRAN compiler and the loader.

An exception to this occurs when the function or subroutine name is itself a dummy argument. It is not possible to determine at compile time or load time whether a character constant or Hollerith constant is required. In this case, a character constant actual argument can correspond only to a character dummy argument.

CHAPTER 14

BLOCK DATA SUBPROGRAMS

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

A block data subprogram must start with the BLOCK DATA statement. The only valid statements within a block data subprogram are the specification and DATA statements (COMMON, DIMENSION, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, type statements, and DATA statements). The last statement of a block data subprogram must be an END statement.

You can enter initial values into more than one labeled common block in a single subprogram of this type.

An executable program can contain more than one block data subprogram.

14.1 BLOCK DATA STATEMENT

The form of the BLOCK DATA statement is:

```
     BLOCK DATA [sub]
```

where:

sub is the optional symbolic name of a block data subprogram in which the BLOCK DATA statement appears.

The name sub is a global name and must therefore be a unique symbolic name within the executable program.

The following is an example of a block data subprogram:

```
     BLOCK DATA TEST
     COMMON /SQUARE/ CIRCLE,RECTAN,PI
     DATA CIRCLE,RECTAN,PI/1.,2.,3.14159/
     END
```

This example initializes the COMMON variables CIRCLE, RECTAN, and PI to 1., 2., and 3.14159 respectively.

CHAPTER 15

WRITING USER PROGRAMS

This chapter is a guide for writing effective programs with FORTRAN. It contains techniques for optimization, interaction with non-FORTRAN programs, and other useful programming hints.

15.1 GENERAL PROGRAMMING CONSIDERATIONS

The following paragraphs describe programming considerations you should observe when preparing a FORTRAN program to be compiled by FORTRAN.

15.1.1 Accuracy and Range of Double-Precision Numbers

Floating-point and real numbers may consist of up to 16 digits in a double-precision mode. Their range is specified in Chapter 3, Section 3.2. You must be careful when testing the value of a number within the specified range, since, although numbers up to 10**38 may be represented, FORTRAN can only test numbers of up to eight significant digits (REAL precision) and 16 significant digits (DOUBLE precision).

NOTE

> For KL model B systems, if the /GFLOATING compiler switch is specified (see Section 16.1.3 or 16.2.3), double-precision numbers up to 10**307 can be represented.

You must also be careful when testing floating-point computations for a result of 0. In most cases the anticipated result, that is, 0 will be obtained; however, in some cases the result may be a very small number that approximates 0. Such an approximation of 0 will cause tests for equality to 0 to fail.

To increase the accuracy of its compile-time arithmetic, the compiler does all floating point arithmetic in double precision (it converts back to single precision when necessary). For KL model B systems, if the /GFLOATING compiler switch is specified (see Section 16.1.3 or 16.2.3), the compiler does its compile-time arithmetic in G-floating double-precision. If the /DFLOATING compiler switch is specified (the default), the compiler does its compile-time arithmetic in D-floating double-precision.

Compile-time arithmetic done in G-floating double-precision may not overflow or underflow as it might with D-floating double-precision arithmetic. G-floating double-precision has a greater range than D-floating double-precision (see Section 3.4).

15.1.2 Writing FORTRAN Programs for Use on Other Computers

If you prepare a program to run on both TOPS-10 or TOPS-20 and another manufacturer's computer, you should:

1. Avoid using any blue print language extensions in this manual. (All information in blue print represents aspects of FORTRAN that are extensions to the FORTRAN-77 Standard.)

2. Consider the accuracy and size of the numbers that another manufacturer's computer is capable of handling.

You can use the /FLAG compiler switch to invoke the compatibility flagger. This feature provides warning messages for language elements that are extensions to the FORTRAN-77 standard or that are incompatible with VAX FORTRAN (see Section 16.6).

15.1.3 Using Floating-Point DO Loops

FORTRAN permits you to use noninteger, single- or double- precision numbers as the parameter variables in a DO statement. This lets you generate a wider range of values for the DO loop index variables, which may, in turn, be used inside the loop for computations.

WARNING

> If a noninteger index is used, accumulation of rounding errors may lead to unexpected values for the loop variable.

15.1.4 Computation of DO Loop Iterations

The number of times through a DO loop is computed outside the loop and is not affected by any changes to the DO index parameters within the loop. The formula for the number of times a DO loop is executed is:

```
DO 10 I=M1,M2,M3

Number of cycles=MAX (INT((M2-M1+M3)/M3),0)
```

The values of the parameters M1, M2, M3 can be of any type except complex. If the iteration count is less than or equal to zero, the body of the loop is not executed. The index variable retains its assigned value (M1). (See Section 9.3.)

NOTE

> The interpretation of the iteration count and the index variable described above is different from that of earlier versions of FORTRAN-10/20. If the /NOF77 compiler switch is specified (see Section 16.1.3 or 16.2.3), and the iteration count is less than or equal to zero, the body of the loop is executed once. In addition, the final value of the index variable of the DO statement is undefined after a normal exit.

15.1.5 Subroutines - Programming Considerations

Consider the following items when preparing and executing subroutines:

1. During execution, no check is made to see if the proper number of arguments is passed (unless the /DEBUG:ARGUMENTS compiler switch is specified, see Section 16.3).

2. If the number of actual arguments passed to a subroutine is less than the number of dummy arguments specified, the values of the unspecified arguments are undefined (unless the /DEBUG:ARGUMENTS compiler switch is specified, see Section 16.3).

3. If the number of actual arguments passed to a subroutine is greater than the number of dummy arguments given, the excess arguments are ignored (unless the /DEBUG:ARGUMENTS compiler switch is specified, see Section 16.3).

4. If an actual argument is a constant and its corresponding dummy argument is set to another value (an illegal usage), all references made to the constant in the calling program may be changed to the new value of the dummy argument.

5. No check is made to see if the arguments passed are of the same type as the dummy arguments (unless the /DEBUG:ARGUMENTS compiler switch is specified, see Section 16.3).

NOTE

An exception is that a check is always made for G-floating and D-floating type mismatches regardless of the /DEBUG:ARGUMENTS switch being specified.

In addition, when you pass character data to a subroutine or function that is compiled with the /EXTEND switch, the calling program must also be compiled with the /EXTEND switch.

If an actual parameter is a constant and the corresponding dummy is of type real, be sure to include the decimal point in the constant. If the dummy is double-precision, be sure to specify the constant with a "D".

NOTE

You are given no warning if any of the situations described in items 1,2,3,4, or 5 occur (unless the /DEBUG:ARGUMENTS compiler switch is specified, see Section 16.3).

Examples:

If a function F has a single dummy argument of type real, and that function is called with:

```
F(2)
```

F interprets the integer 2 as an unnormalized floating-point number. In this instance, F(A) should be called with:

```
     F(2.0)
```

Similarly, if the function F1(D) is called with:

```
     F1(2.5)
```

and D is double-precision, F1 assumes that its parameters have been specified with two words of precision and picks up whatever follows the constant 2.5 in memory. The proper method is to use:

```
     F1(2.5D0)
```

15.1.6 Reordering of Computations

Computations that are not enclosed within parentheses may be reordered by the compiler. Sometimes it is necessary to use parentheses to ensure proper results from a specific computation.

For example, assuming that:

1. RL1 represents a large number, such that RL1*RL2 will cause an overflow condition, and

2. RS1 is a very small number, that is, less than 1, the program sequence:

   ```
          .
          .
          .
   A=RS1*RL1*RL2
   B=RS2*RL2*RL1
          .
          .
          .
   ```

   will not produce an overflow when evaluated left to right, since the first computation in each expression (that is, RS1*RL1 and RS2*RL2) will produce an interim result that is smaller than either large number (RL1 or RL2).

However, the compiler may recognize RL1*RL2 as a common subexpression (see Section 15.2.1.1) and generate the following sequence:

```
     temp = RL1*RL2
     A    = RS1*temp
     B    = RS2*temp
```

The computation of temp will cause an overflow.

You should write the program as follows to ensure that the desired results are obtained:

```
          .
          .
     A=(RS1*RL1)*RL2
     B=(RS2*RL2)*RL1
```

Computations may be reordered even when global optimization is not selected.

15.1.7 Dimensioning of Dummy Arrays

When you specify an array as a dummy argument to a subprogram unit, you must indicate to the compiler that the parameter is an array by dimensioning the array in a specification statement. This is the only way the compiler is able to distinguish a reference to such an array from a function reference. A dummy array can be dimensioned the following ways:

1. Assumed size

2. Adjustable dimensioned

3. Fixed dimension bound

Dimensioning the array with a size of 1 is a common, although dangerous, practice. The alternative to this practice is to use assumed-size arrays (see Section 7.1.2).

Example:

```
SUBROUTINE SUB1(A,B)
DIMENSION A(1)
```

There are disadvantages to using the above technique because it may prevent the compiler from diagnosing illegal programs, specifically:

1. Reading or writing the array by name

   ```
   DIMENSION ARRAY (10)
   READ (1) ARRAY
   ```

   The above is a binary read that will read ten words into ARRAY.

   ```
   SUBROUTINE SUB1(A)
   DIMENSION A(1)
   READ(1)A
   ```

   This binary read will cause one word to be read into A.

2. Using the array as a format

   ```
   SUBROUTINE SUB2(FMT)
   DIMENSION FMT(1)
   READ (1,FMT)
   ```

   Only the first word of the format specification contained in FMT is used.

3. Using the /DEBUG:BOUNDS compilation switch (see Section 16.3), the dimension information used is that which is specified in the array declaration

   ```
   SUBROUTINE SUB3(A)
   DIMENSION A(1)
   A(2)=0
   ```

   The reference to A(2) will cause the out-of-bounds warning message to be generated.

15.2 FORTRAN GLOBAL OPTIMIZATION

You have the option of invoking the global optimizer during compilation. The optimizer treats groups of statements in the source program as a single entity. The purpose of the global optimizer is to prepare a more efficient object program that produces the same results as the original unoptimized program, but takes significantly less execution time.

The output of the lexical and syntactic analysis phase of the compiler is developed into an optimized source program equivalent (in results) to the original. The optimized program is then processed by the standard compiler code generation phase.

15.2.1 Optimization Techniques

15.2.1.1 Elimination of Redundant Computations - Often the same subexpression will appear in more than one computation throughout a program. If the values of the operands of such a common expression are not changed between computations, the subexpression may be written as a separate arithmetic expression. Also, the variable representing its resultant may then be substituted where the subexpression appears. This eliminates unnecessary recomputation of the subexpression. For example, the instruction sequence:

```
     A=B*C+E*F
         .
         .
         .
     H=A+G-B*C
         .
         .
         .
     IF((B*C)-H) 10,20,30
```

contains the subexpression B*C three times when it really needs to be computed only once. Rewriting the preceding sequence as:

```
     T=B*C
     A=T+E*F
         .
         .
     H=A+G-T
         .
         .
     IF(T-H) 10,20,30
```

eliminates two computations of the subexpression B*C from the overall sequence.

Decreasing the number of arithmetic operations performed in a source program by the elimination of common subexpressions shortens the execution time of the resulting object program.

15.2.1.2 Reduction of Operator Strength - The time required to execute arithmetic operations varies according to the operator(s) involved. The hierarchy of arithmetic operations according to the amount of execution time required is:

```
MOST TIME     OPERATOR
                 **
                 /
                 *
LEAST TIME       +,-
```

During program optimization, the global optimizer replaces, where possible [1] some arithmetic operations that require the most time with operations that require less time. For example, consider the following DO loop that is used to create a table for the conversion of from 1 to 20 miles to their equivalents in feet:

```
     DO 10 MILES=1,20
10   IFEET(MILES)=5280*MILES
```

The execution time of the loop would be shorter if the time-consuming multiply operation, that is, 5280*MILES, could be replaced by a faster operation. Since you increment MILES on each pass, you can replace the multiply operation by an add and total operation.

In its optimized form, the loop would be replaced by a sequence equivalent to:

```
     K=5280
     DO 10 MILES=1,20
     IFEET(MILES)=K
10   K=K+5280
```

In the optimized form of the loop, the value of K is set to 5280 for the first iteration of the loop, and is increased by 5280 for each succeeding iteration of the loop.

This situation occurs frequently in subscript calculations that implicitly contain multiplications.

15.2.1.3 Removal of Constant Computation from Loops - The speed with which a given algorithm may be executed can be increased if instructions and/or computations are moved out of frequently traversed program sequences into less frequently traversed program sequences.

Movement of code is possible only if none of the arguments in the items to be moved are redefined within the code sequences from which they are to be taken. Computations within a loop consisting of variables or constants that are not changed in value within the loop may be moved outside the loop. Decreasing the number of computations made within a loop greatly decreases the execution time required by the loop.

For example, in the sequence:

```
     DO 10 I=1,100
10   F=2.0*Q*A(I)+F
```

[1] Numerical analysis considerations severely limit the number of cases where this is possible.

the value of the computation 2.0*Q, once calculated on the first iteration, will remain unchanged during the remaining 99 iterations of the loop. Reforming the preceding sequence to:

```
         QQ=2.0*Q
         DO 10 I=1,100
     10  F=QQ*A(I)+F
```

moves the calculation 2.0*Q outside the scope of the loop. This movement of code eliminates 99 multiply operations.

In addition, it is possible to remove entire assignment statements from loops. This action can be easily detected from the macro expanded listings. The internal sequence number remains with the statement and appears out of order in the leftmost column of the macro expanded listing (LINE).

15.2.1.4 Constant Folding and Propagation - In this method of optimization, expressions containing determinate constant values are detected and the constants are replaced, at compile time, by their defined or calculated value. For example, assume that the constant PI is defined and used in the following manner:

```
         .
         .
         .
     PI=3.14159
         .
         .
         .
     X=2*PI*Y
         .
         .
         .
```

At compile time, the optimizer will have used the defined value of PI to calculate the value of the subexpression 2*PI. The optimized sequence would then be:

```
         .
         .
     PI=3.14159
         .
         .
         .
     X=6.28318*Y
         .
         .
```

thereby eliminating a multiply operation from the object code program.

The evaluation of constant expressions at compile time is called "folding"; the replacement of variables with their constant values is called "constant propagation".

NOTE

> For KL model B systems, use of the /GFLOATING compiler switch (see Section 16.1.3 or 16.2.3) may affect compile-time arithmetic.

15.2.1.5 Removal of Inaccessible Code - The optimizer detects and eliminates any code within the source program that cannot be accessed. In general, this will not happen since programmers do not normally include such code in their programs; however, inaccessible code may appear in a program during the debugging process. The removal of inaccessible code by the optimizer reduces the size of the object program.

A warning message is generated for each inaccessible line removed.

15.2.1.6 Global Register Allocation - During the compilation of a source program, the optimizer controls the allocation of registers to minimize computation time in the optimized object program. The allocation process is designed to minimize the number of MOVE and MOVEM machine instructions that will appear in the most frequently executed portions of the code.

15.2.1.7 I/O Optimization - Every effort is made to minimize the number of required calls to the FOROTS system. This is done primarily through extensive analysis of implied DO loop constructs on I/O data transfer statements. The formats of these special blocks are described in Chapter 18. These optimizations reduce the size of the program (argument code plus argument block size is reduced) and greatly improve the performance of programs that use implied DO loop I/O statements.

15.2.1.8 Uninitialized Variable Detection - A warning message may be generated when a scalar variable is referenced before it has received a value (only when optimizing).

15.2.1.9 Test Replacement - If the only use of a DO loop index is to reduce operator strength (see Section 15.2.1.2) and the loop does not contain exits (GO TOs out of the loop), the DO loop index is not needed and can be replaced by the reduced variable.

For example:

```
         DO 10 I=1,10
         K=K+7*I
     10  CONTINUE
```

Reduction of operator strength and test replacement together transform this loop into:

```
         DO 10 I=7,70,7
         K=K+I
     10  CONTINUE
```

This situation occurs frequently in subscript computation. After execution of these statements, I=11.

15.2.2 Programming Techniques for Effective Optimization

Observe the following recommendations during the coding of a FORTRAN source program. They will improve the effectiveness of the optimizer:

1. Do not use DO loops with an extended range.
2. Specify label lists when using assigned GO TOs.
3. Nest loops so that the innermost index is the one with the largest range of values.
4. Avoid the use of associated I/O variables.
5. Avoid unnecessary use of COMMON and EQUIVALENCE.

15.3 FUNCTION SIDE EFFECTS

Unpredictable results can occur if a statement includes calls to different functions that modify the same variables.

Consider the following example:

```
COMMON A
A=5.
P=F(1.)+Q(2.)
END

FUNCTION F(X)
COMMON A
A=0.
F=X+1
END

FUNCTION Q(Y)
COMMON A
Q=A
END
```

In the preceding sequence, if P is evaluated by calling F before Q, the value of P will be 2. If P is evaluated by calling Q before F, the value of P will be 7.

15.4 INTERACTING WITH NON-FORTRAN PROGRAMS AND FILES

15.4.1 Using The Sharable High-Segment FOROTS

If your program does not contain a FORTRAN main program module, and you wish to have the sharable FOROTS GETSEGed at run time, you must do the following:

1. Force the loading of the FOROTS initialization routine RESET. by declaring it as an external.
2. Define the symbol FOROT% as a global with a positive, nonzero value before FORLIB.REL is searched.

3. Initialize FOROTS by the appropriate initialization call:

```
JSP 16, RESET.
EXP 0
```

15.4.2 Calling Sequences

The following paragraphs describe the standard procedures for writing subroutine calls.

1. Procedure

   a. The calling program must load the accumulator (AC) 16 with the address of the first argument in the argument list.

   b. The subroutine is then called by a PUSHJ instruction using AC 17.

   c. The return will be made to the instruction immediately after the PUSHJ 17 instruction.

   d. The FOROTS trace facility requires the calling sequence to be:

   ```
   XMOVEI 16,AP
   PUSHJ 17,F
   ```

   where AP is the pointer to the argument list and F is the subprogram name. The word preceding the first word of an entry point should have its name in SIXBIT.

2. Restrictions

   a. Skip returns are not permitted.

   b. The contents of the pushdown stack located before the address specified by AC 17 belong to the calling program; they cannot be read by the called subprogram.

   c. FOROTS assumes that it has control of the stack; therefore, you must not create your own stack. The FOROTS stack is initialized by the call to RESET. (See Section 15.4.1).

15.4.3 Accumulator Usage

The specific functions performed by accumulators (AC) 17,16,0, and 1 are:

1. Pushdown Pointer - AC 17 is always maintained as a pushdown pointer. In section zero, its right half points to the last location in use on the stack, and its left half contains the negative of the number of words allocated to the unused remainder of the stack.

   In non-zero sections, the pushdown pointer contains the global address of the last location in use on the stack.

2. Argument List Pointer - AC 16 is used as the argument pointer. The called subprogram does not need to preserve its contents. The calling program cannot depend on getting back the address of the argument list passed to the called subprogram. AC 16 cannot point to the ACs or to the stack.

3. Temporary and Value Return Registers - AC 0 and 1 are used as temporary registers and for returning values. The called subprogram does not need to preserve the contents of AC 0 or 1 (even if not returning a value). The calling program must never depend on getting back the original contents of the data passed to the called subprogram.

4. Returning Values - A subroutine subprogram may pass back results by modifying arguments.

   A numeric function subprogram always returns the value of the function in AC 0 (or ACs 0-1 if the value is double precision or complex). A function subprogram may also pass back additional results by modifying the arguments. (See Section 15.4.4 for a description of character functions.)

5. Preserved ACs - FORTRAN function subprograms preserve ACs 2 through 15; subroutine subprograms do not.

The design of the called subprogram cannot depend on the contents of any of the ACs being set up by the calling subprogram, except for ACs 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either FORTRAN or COBOL.

15.4.4 Argument Lists

Since the FORTRAN compiler uses the indirect bits on argument lists (note that this permits shared, pure code argument lists), it is essential for code that accesses parameters to take this into account. Specifically, sequences that obtained the values of parameters through use of operations such as:

```
     HRRZ R,1(16)
```

to pick up the address of the second argument should be changed to

```
     XMOVEI R,@1(16)
```

The latter operation will work when interfacing with FORTRAN.

The format of the argument list is as follows:

```
                     Arg count word
     Arg list addr.---First arg entry
                     Second arg entry
                         .
                         .
                         .
                     Last arg entry
```

The format of the arg count word is:

bits 0-17 These contain -n, where n is the number of arg entries.
bits 18-35 Reserved for future DIGITAL development, and must be 0.

The format of an arg entry is as follows (each entry is a single word):

bit 0	IFIW (Instruction Format Indirect Word) flag, must be 1.
bits 1-8	Reserved for future DIGITAL development, must be 0.
bits 9-12	Arg type code.
bit 13	Indirect bit if desired.
bits 14-17	Index field, must be 0.
bits 18-35	Address of the argument.

For character functions, the first argument points to the return value, which is a character string descriptor (see Section 15.4.6). The actual arguments to the function start in the second argument entry.

The following restrictions should be observed:

1. Neither the argument list nor the arguments themselves can be on the stack. The same restriction applies to any indirect argument pointers.

2. The called program may not modify the argument list itself. The argument list may be in a write-protected segment.

   Note that the arg count word is at position -1 with respect to the contents of AC 16. This word is always required even if the subroutine does not handle a variable number of arguments. A subroutine that has no arguments must still provide an argument list consisting of two words, that is, the argument count word with a 0 in it and a zero argument word.

Example:

```
          XMOVEI 16,AP    ;SET UP ARG POINTER
          PUSHJ 17,SUB    ;CALL SUBROUTINE
          ...             ;RETURN HERE
              .
              .
              .
          ;ARGUMENT LIST
          -3,,0
     AP:  IFIW 4,A
          IFIW 4,B
          IFIW 4,C

     ;SUBROUTINE TO SET THIRD ARG TO SUM OF FIRST TWO ARGS

     SUB:      MOVE       T,@0(16)       ;GET FIRST ARG
               ADD        T,@1(16)       ;ADD SECOND ARG
               MOVEM      T,@2(16)       ;SET THIRD ARG
               POPJ       17,            ;RETURN TO CALLER
```

15.4.5 Argument Types

Table 15-1: Argument Types and Types Codes

Type Code (Octal)	Description	
	FORTRAN Use	COBOL Use
0	Unspecified	Unspecified
1	FORTRAN Logical	Not applicable
2	Integer	1-word COMP
3	Reserved	Reserved
4	Real	COMP-1
5	Reserved	Reserved
6	Octal	Reserved
7	Label	Procedure address
10	Double real (D-floating)	Not applicable
11	Not applicable	2-word COMP
12	Double octal	Not applicable
13	Double real (G-floating)	Not applicable
14	Complex	Not applicable
15	Character	Byte string descriptor
16	Reserved	Reserved
17	Hollerith	Not applicable

Literal arguments are permitted, but they must reside in a writable segment. This is because the FORTRAN compiler makes a local copy of all non array elements and may copy dummy arguments back to the actual arguments. All unused type codes are reserved for future DIGITAL development.

15.4.6 Description of Arguments

The types of the arguments that may be passed are:

1. Type 0 - Unspecified

   The calling program has not specified the type. The called subprograms should assume that the argument is of the correct type if it is checking types. If several types are possible, the called subprogram should assume a default as part of its specification. If none of the above conditions is true, the called subprogram should handle the argument as an integer (type 2).

2. Type 1 - FORTRAN logical

   A 36-bit binary value containing 0 or positive to specify .FALSE. and negative to specify .TRUE..

3. Type 2 - Integer and 1-word-COMP

   A 36-bit 2's complement signed binary integer.

4. Type 4 - Real and COMP-1

   A 36-bit floating-point number.

bit 0	sign
bits 1-8	excess 128 exponent
bits 9-35	mantissa

5. Type 6 - Octal

   A 36-bit unsigned binary value.

6. Type 7 - Label and procedure address

   The address of the parameter is the address of an alternate return label or a subprogram.

7. Type 10 - Double real (D-floating)

   A double-precision floating-point number represented in D-floating form. (See Section 3.4.)

8. Type 11 - 2-word COMP

   A 2-word (72-bit) 2's complement signed binary integer.

word 1, bit 0	sign
word 1, bits 1-35	high order
word 2, bit 0	same as word 1, bit 0
word 2, bits 1-35	low order

9. Type 12 - Double octal

   A 72-bit unsigned binary value.

10. Type 13 - Double real (G-floating) (KL model B only)

    A double-precision floating-point number represented in G-floating form. (See Section 3.4.)

11. Type 14 - Complex

    A complex number represented as an ordered pair of 36-bit floating-point numbers. The first represents the real part, and the second represents the imaginary part.

12. Type 15 - Character string descriptor

    The format of the character string descriptor is:

    word 1: ILDB-type pointer, that is, aimed at the character preceding the first character of the string
    word 2: EXP character count

    The character descriptor may not be modified by the called program. The character string itself must consist of a string of contiguous 7-bit ASCII characters.

13. Type 17 - Hollerith

    A string of contiguous 7-bit ASCII characters left justified on the word boundary of the first word and terminated by a null character in the last word.

    The FORTRAN compiler emits constants that are padded with spaces to a word boundary, followed by a full-word containing zero.

15.4.7 Interaction with COBOL

FORTRAN programs can call COBOL programs as subprograms, and, conversely, the COBOL programs can call FORTRAN programs as subprograms.

Note that I/O operations can be performed only in subprograms that are written in the same language as the main program. Also note that APR trap handling will be done in a manner consistent with the language used in the main program.

15.4.7.1 Calling FORTRAN Subprograms from COBOL Programs - COBOL programmers may write subprograms in FORTRAN to use the conveniences and facilities provided by this language. The COBOL verb ENTER is used to call FORTRAN subroutines. The form of ENTER is as follows:

```
                            [         {identifier-1      }   [   {identifier-2}  ]
ENTER FORTRAN program name  [ USING   {literal-1         }   [ , {literal-2...}  ]
                            [         {procedure-name-1  }   [   {procedure-2 }  ]
```

The USING clause names the data within the COBOL program that is to be passed to the called FORTRAN subprogram. The passed data must be in a form acceptable to FORTRAN (see Table 14-1).

NOTE

> G-floating double-precision does not exist as a data type in COBOL.

The calling sequence used by COBOL in calling a FORTRAN subprogram is:

```
MOVEI 16, address of first entry in argument list
PUSHJ 17, subprogram address
```

If the USING clause appears in the ENTER statement, the compiler creates an argument list that contains an entry for each identifier or literal in the order of appearance in the USING clause. It is preceded by a word containing, in its left half, the negative number of the number of entries in the list. If no USING clause is present, the argument list contains an empty word, and the preceding word is set to 0. Each entry in the list is one 36-bit word of the form:

0-8	9-12	13-35
0	type	address

Bits 0-8 are reserved for DIGITAL.

Bits 9-12 contain a type code that indicates the USAGE of the argument.

Bits 13-35 contain the address of the argument of the first word of the argument; the address can be indexed or indirect.

Following is a description of the types generated by COBOL, their codes, how the codes appear in the argument list, and the locations specified by the addresses.

1. For 1-word COMPUTATIONAL items

CODE:	2
IN ARGUMENT LIST:	XWD 100, address
ADDRESS:	that of the argument itself
FORTRAN TYPE:	INTEGER

2. For 2-word COMPUTATIONAL items

CODE:	11
IN ARGUMENT LIST:	XWD 440, address
ADDRESS:	that of the high-order word of the argument
FORTRAN TYPE:	Not allowed

3. For COMPUTATIONAL-1 items

CODE:	4
IN ARGUMENT LIST:	XWD 200, address
ADDRESS:	that of the argument itself
FORTRAN TYPE:	REAL

4. For DISPLAY-6 and DISPLAY-7 items

CODE:	15
IN ARGUMENT LIST:	XWD 640, address
ADDRESS:	that of a 2-word descriptor for the argument
WORD1:	a byte pointer to the identifier or literal
WORD2:	bit 0 is 1 if the item is numeric bit 1 is 1 if the item is signed bit 2 is 1 if the item is a figurative constant (including ALL) bit 3 is 1 if the item is a literal bits 4 through 11 are reserved for expansion bit 12 is 1 if the item has a PICTURE with one or more Ps just before the decimal point, that is, 99PPV. bits 13 through 17 are the number of decimal places. If bit 12 is 1, this is the number of Ps. bits 18 through 35 contain the size of the item in bytes.
FORTRAN TYPE:	If FORTRAN is called, the string must be DISPLAY-7, nonnumeric, and either a figurative constant or literal. The bits 0-17 of words must be zero. The FORTRAN type is then character.

5. For procedure names (which cannot be used for calls to COBOL subprograms)

CODE:	7
IN ARGUMENT LIST:	XWD 340, address
ADDRESS:	that of the procedure
FORTRAN TYPE:	External subprogram name

The return from a subprogram (through POPJ 17,) is to the statement after the call.

15.4.7.2 Calling COBOL Subroutines from FORTRAN Programs - To call COBOL subprograms use the CALL statement:

```
CALL sub (args...)
```

where sub is a COBOL subprogram.

15.4.8 Interaction with BLISS-36

FORTRAN programs can call BLISS-36 routines as subprograms, and, conversely, BLISS-36 programs can call subprograms written in FORTRAN.

BLISS routines called by FORTRAN programs must be able to coexist compatibly with FOROTS. For instance, they must use FUNCT. functions (see Section 18.6) for dynamic memory management within the section that FOROTS is in, since FOROTS assumes that it has control of that section.

One problem that the BLISS routines can encounter is stack overflow. The size of the program stack as set up by FOROTS may be too small for BLISS routines which have several STACKLOCAL variables. The only supported way to increase the size of the stack is to use a copy of FORLIB that has been assembled with a larger stack.

15.4.8.1 Calling FORTRAN Subprograms From BLISS-36 Programs - To call a FORTRAN subprogram from a BLISS-36 program, the FORTRAN subprogram must be declared an EXTERNAL ROUTINE in any module that contains a call to the subprogram. In addition, if the FORTRAN subprogram is a subroutine, it must be declared with a linkage type of FORTRAN_SUB. If the FORTRAN subprogram is a function, then it must be declared with a linkage of FORTRAN_FUNC. For example:

```
EXTERNAL ROUTINE FOO:  FORTRAN_SUB,
                 BAR:  FORTRAN_FUNC;
```

declares FOO to be the name of a FORTRAN subroutine which will be called in this module, and declares BAR to be the name of a FORTRAN function. After the FORTRAN subprogram has been declared appropriately, it can be called just like any function written in BLISS.

15.4.8.2 Calling BLISS-36 Routines From FORTRAN - A BLISS-36 routine that is to be called from a FORTRAN program must have either the FORTRAN_SUB linkage attribute (if the routine is to be used as a subroutine) or the FORTRAN_FUNC linkage attribute (if the routine is to be used as a function).

The programmer that wishes to write a BLISS-36 routine to be called from FORTRAN must be familiar with the calling sequence used by FORTRAN to call subprograms (see Section 15.4.2), FORTRAN argument lists (see Section 15.4.4), and FORTRAN argument descriptors (see Sections 15.4.5 and 15.4.6). This knowledge is necessary because the values of the formal arguments of the BLISS-36 routine are the FORTRAN argument list entries that correspond to actual arguments of the BLISS routine.

In general, the BLISS routines must be compiled with ADDRESSING_MODE(INDIRECT), or MACHOP calls must be used to generate any instruction that references formals since all FORTRAN arguments must be accessed through indirect addressing. This must be done because the FORTRAN compiler frequently sets the indirect bit in argument lists (see Section 15.4.4).

See the BLISS-36 Language Guide for more information.

NOTE

> ADDRESSING_MODE(INDIRECT) can have far reaching effects on your program, which you should understand fully before using.

15.4.9 LINK Overlay Facilities

LINK provides several routines that are accessible directly from a FORTRAN program. These routines are presented here briefly, together with the FORTRAN specification of their parameters. In general, LINK performs these functions automatically. These routines are available only for your convenience. Full details of the use of the overlay facilities can be found in the LINK Reference Manual.

NOTE

> Overlays are not allowed when TOPS-20 extended addressing is used.

The following terms are used to describe the parameters to LINK overlay routines.

File spec	A character expression consisting of 'dev:file.ext[directory]' (TOPS-10), or 'dev:<directory>file.typ.gen' (TOPS-20).
Name	A quoted string giving a link name, or an integer constant or variable giving a link number.
List of link names	A sequence of name items separated by commas.

The routines available are:

CLROVL	Specifies a non-writable overlay.
GETOVL(List of link names)	Used to change the overlay structure in core.
INIOVL(File spec)	Used to specify the overlay file to be found if the load time specification is to be overridden.
LOGOVL(File spec)	Used to specify where the log file is to be written. If no arguments are given, the log file is closed.
REMOVL(List of link names)	Removes the specified links from core.
RUNOVL(Name)	Loads the specified link and transfers to that LINK.
SAOVL	Specifies a writable overlay.

For a full description of these routines, refer to the *LINK Reference Manual*.

NOTE

> The SAVE statement retains the values stored in a variable, array, or common block after execution of a RETURN or END statement in a subprogram. When overlays are used, the SAVE statement must be used to ensure retention of values. When the SAVE statement is used, it is not necessary to specify the LINK switch /OVERLAY:WRITABLE when loading a program (see Section 7.10).

15.4.10 FOROTS and Overlay Memory Management

For sharable FOROTS, the FOROTS static data area is several pages located at the top of FOROTS. FOROTS dynamic memory is allocated at runtime below FOROTS and in a downward direction (toward the user's program).

For nonsharable FOROTS (FOROTS loaded from FORLIB), the FOROTS data area is located in the low segment, so that it will be linked with variables used by the user's program. FOROTS dynamic memory is allocated at runtime starting at the page designated by the symbol STARTP in FORPRM.MAC, downward toward the user program. The distributed value for this page number is 577. If the user's program has two segments, FOROTS allocates memory down to the user's high segment, skips over the high segment, and begins allocating memory below the user's high segment toward the user's low segment.

For both sharable and nonsharable FOROTS, when FOROTS can no longer allocate memory downward toward the user's low segment, it allocates memory starting at the top of memory downward. When FOROTS can no longer allocate any memory, it reports:

```
?Memory full
```

and returns to the monitor after attempting to close all files.

Figure 15-1 illustrates the run-time memory layout.

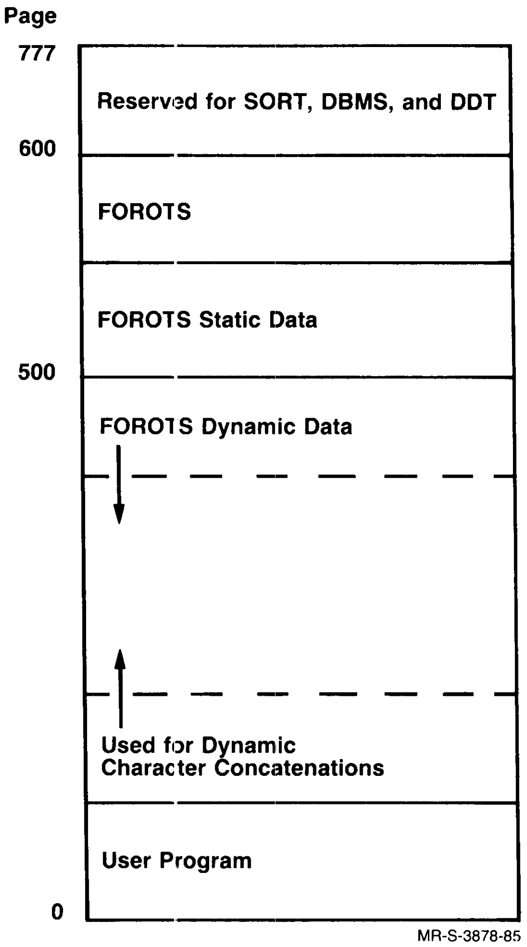


Figure 15-1: Run-time Memory Layout for Section Zero

FOROTS has a separate memory manager specifically designed for use by OVRLAY. This memory manager allocates memory at the top of the users's low segment. In general, user programs that use overlays should not use the FUNCT. calls GAD, COR, and RAD. OVRLAY expects to be able to use memory beginning at the top of the user's low segment, allowing for a special FOROTS scratch space allocated by the /SPACE switch.

Under certain circumstances, notably when concatenating character expressions whose length is not known until runtime, FOROTS dynamically allocates a special scratch area using the FUNCT. COR function (see Section 18.6). This area is allocated the first time a dynamic concatenation is performed.

The /SPACE switch to LINK determines the maximum size for such "dynamic concatenations" if overlays are used. The default value given by LINK for this value is 4000 (octal). (See the LINK Reference Manual.)

You can allocate space for dynamic character concatenation using the ALCCHR subroutine (see Section 13.4.1.1).

15.4.11 Extended Addressing Memory Layout (TOPS-20 only)

The FORTRAN compiler must determine which psect every word of data or code should reside in. For non-extended addressing compilations, the data and code reside in either the low segment or the high segment. For extended addressing compilations, there are three segments (psects) in which the data and code can be allocated:

1. The code psect corresponds to the non-extended high segment. The default name is .CODE..
2. The data psect corresponds to the non-extended low segment. The default name is .DATA..
3. The .LARG. psect is where large data resides. (Note that the name of this psect cannot be changed.)

A data item can be placed in the .LARG. psect by either of the following ways:

1. The data item is an array or any character scalar whose size is greater or equal to the value of the /EXTEND:DATA switch (default 10,000 words). (See Section 16.5).
2. The data item is placed in a COMMON block or EQUIVALENCE group that is in the .LARG. psect.

Table 15-2 describes the various memory allocations for extended and non-extended compilations.

Table 15-2: Memory Allocations for /EXTEND and /NOEXTEND

Item	/NOEXTEND	/EXTEND
User subprogram	Hiseg	Code
FORLIB	Hiseg	.CODE.
Argument blocks	Hiseg	Code
Compile-time constant character descriptors	Hiseg	Code
Array dimension information	Hiseg	Code
EFIWS	N/A	Code
Symbol tables (from LINK)	Lowseg or Hiseg	.DATA. (by default)
FORMAT statements	Lowseg	Data
Constants	Lowseg	Data
Small arrays and scalars	Lowseg	Data
Large arrays	Lowseg	.LARG.
COMMON variables	Lowseg	.LARG. (by default)
Variables EQUIVALENCED to large arrays	Lowseg	.LARG.
PDV	N/A	.DATA.

NOTE

When the sharable FORCTS is used, LINK loads the /NOEXTEND high-segment into the low-segment. This is done so that the sharable FOROTS can be used as the high-segment (see Section 16.9)

CHAPTER 16

USING THE FORTRAN COMPILER

This chapter explains how to use the FORTRAN compiler. Section 16.1 describes using the FORTRAN-10 compiler and Section 16.2 describes using the FORTRAN-20 compiler.

16.1 USING THE FORTRAN-10 COMPILER

This section describes how use the FORTRAN-10 compiler. You should be familiar with the TOPS-10 operating system. The TOPS-10 Operating System provides commands that enable you to compile, execute, and debug FORTRAN programs. These commands are known as the COMPILE-Class commands.

16.1.1 TOPS-10 COMPILE-Class Commands

You can invoke the FORTRAN-10 compiler by using TOPS-10 COMPILE-Class commands. These commands enable you to compile, execute, and debug a program by giving the commands at TOPS-10 command level.

The COMPILE-Class commands are:

```
COMPILE
LOAD
EXECUTE
DEBUG
```

Example:

```
.EXECUTE ROTOR.FOR
```

The compiler switches OPTIMIZE, CREF, and DEBUG may be specified directly in COMPILE-Class commands and may be used globally or locally. (See Section 16.1.3 for a description of FORTRAN-10 switches.)

Example:

```
.EXECUTE/CREF P1.FOR,P2.FOR/DEBUG
```

The other compiler switches must be passed in parentheses for each specific source file.

Example:

```
     .EXECUTE P1.FOR(INCLUDE)
```

Refer to the TOPS-10 Operating System Commands Manual for further information about the COMPILE-Class commands.

16.1.2 RUNNING THE FORTRAN-10 COMPILER

On TOPS-10, the command to run the FORTRAN compiler directly is:

```
     .R FORTRA
```

The compiler responds with an asterisk (*), and is then ready to accept a command string. The form of the FORTRAN compiler command string is:

```
     object filespec, listing filespec=source filespec(s)
```

You are given the following options:

1. File specifications consist of an optional device name, a one to six character filename, an optional one to three character file extension, and an optional directory path specification. The path may include SFDs.

2. You may specify more than one source file in the compilation command string. These files will be logically concatenated by the compiler and treated as one source file.

3. More than one program unit may be contained in a single source file.

4. A program unit may consist of more than one source file.

5. If no object file is specified, no relocatable binary file is generated.

6. If no listing file is specified, no listing is generated.

7. If no extension is given, the defaults are the following for the respective files:

   .LST (listing)

   .REL (relocatable binary)

   .FOR (source)

   .CRF (cross reference) if the /CROSSREF switch is specified (see Section 16.1.3)

16.1.3 TOPS-10 Compiler Command Switches

Switches to the FORTRAN-10 compiler are accepted anywhere in the command string. They are totally position and file independent. Table 16-1 lists the switches.

Table 16-1: FORTRAN-10 Compiler Switches

Switch	Meaning	Defaults
/CROSSREF	Generates a file with extension .CRF that can be input to the CREF program.	OFF
/DEBUG	Includes debugging information in your program (see Section 16.3).	NONE
/DFLOATING	Indicates that double-precision numbers are stored in D-floating format. (See Section 3.4.)	ON
/EXPAND	Includes the octal-formatted version of the object file in the listing.	OFF
/F66	The FORTRAN-66 standard rules apply for DO loops and EXTERNAL statements. (Same function as the /NOF77 switch.)	OFF
/F77	The FORTRAN-77 standard rules apply for DO loops and EXTERNAL statements.	ON
/FLAG	Invokes the compatibility flagger (see Section 16.6).	OFF
/GFLOATING	Indicates that double-precision numbers are stored in G-floating format. (See Section 3.4.)	OFF
/INCLUDE	Compiles a D in card column 1 as space.	OFF
/LNMAP	Produces a line number/octal location map in the listing only if /MACROCODE was not specified.	OFF
/MACROCODE	Adds the mnemonic translation of the object code to the listing file.	OFF
/NOFLAG	Indicates that no compatibility flagging will be done (see Section 16.6).	ON
/NOF77	The FORTRAN-66 standard rules apply for DO loops and EXTERNAL statements. (Same function as the /F66 switch.)	OFF
/NOERRORS	Does not print error messages on the terminal.	OFF
/NOWARN	Suppresses warning messages (see Section 16.4).	NONE
/OPTIMIZE	Performs global optimization.	OFF
/SYNTAX	Performs syntax check only.	OFF

Each switch must be preceded by a slash (/). Switch names need only contain those letters that are required to make the switch name unique. You are encouraged to use at least three letters to prevent conflict with switches in future implementations.

Example:

```
.R FORTRA
*OFILE,LFILE=SFILE/MAC,S2FILE
```

The /MAC switch will cause the MACRO code generated for SFILE and S2FILE to appear in LFILE.LST.

All switches, used or implied, are printed at the top of each listing page.

16.2 USING THE FORTRAN-20 COMPILER

This section describes how to use the FORTRAN-20 compiler. You should be familiar with the TOPS-20 Operating System. The TOPS-20 Operating System provides commands that enable you to compile, execute, and debug FORTRAN program. These commands are known as the COMPILE-Class commands.

16.2.1 TOPS-20 COMPILE-Class Commands

The TOPS-20 COMPILE-Class commands enable you to initiate compilation, execution, and debugging of FORTRAN programs from TOPS-20 command level.

The TOPS-20 COMPILE-Class commands are:

```
COMPILE
LOAD
EXECUTE
DEBUG
```

Example:

```
@EXECUTE ROTOR.FOR
```

The following FORTRAN compiler switches (see Section 16.2.3) can be specified directly in a COMPILE-Class command:

```
/ABORT
/BINARY
/CROSS-REFERENCE
/DEBUG
/LIST
/MACHINE-CODE
/NOWARNINGS
/OPTIMIZE
/WARNINGS
```

NOTE

> When you specify the switches /BINARY, /DEBUG, /LIST, and /NOWARNINGS directly in a COMPILE-Class command, the switches cannot have arguments as they can when running the compiler.

All other switches must be specified by using Language-switches, as shown in the following example:

```
@COMPILE TEST.FOR/OPT/LANG:"/NOWARNINGS"
```

Refer to the <u>TOPS-20 Commands Reference Manual</u> for more information about the COMPILE-Class commands.

NOTE

> You cannot use long TOPS-20 filenames with the COMPILE-Class commands.

16.2.2 RUNNING THE FORTRAN-20 COMPILER

On TOPS-20, the command to run the FORTRAN compiler directly is:

```
@FORTRA
```

The compiler responds with the following prompt:

```
FORTRAN>
```

and is then ready to accept a command string.

You can use the question mark to list the commands beginning with a specific letter or letters. Type the letter or letters followed by a question mark. (Refer to the TOPS-20 User's Guide.)

You can type commands to the system by using either full input, recognition input, abbreviated input, or a combination of these methods.

To give a command using full input, type the complete command name, arguments, or switches (if any), using a space to separate the fields.

To give a command using recognition input, type a portion of the switch or filename and press the ESC key. You must type enough of the switch or filename to make it unique. Continue typing and pressing the ESC key until the switch or filename is complete. Recognition input requires less typing than full input, so you are less likely to make a mistake. You can use recognition in typing switches, switches arguments, and file specifications. When typing file specifications, you can also use CTRL/F to complete the rest of a partial file specification.

To give a command using abbreviated input, type only enough of the switch or filename to distinguish one switch or filename from another. (Refer to the TOPS-20 User's Guide.)

You should enter a command string in one of the following forms:

1. [COMPILE]<source-file-spec> [switches]

   If no switches are specified, the compiler produces a .REL file, with the same filename as the source file. The user must use a /LISTING switch to get a listing file (see Section 16.2.3 for a description of FORTRAN-20 compiler switches). COMPILE is optional if the command begins with a switch or begins with a file specification that cannot be confused with the words EXIT, HELP, TAKE, or COMPILE.

2. [COMPILE]<source-file-spec>+<source-file-spec>+... [switches]

   The source files are treated as if they were concatenated together prior to the beginning of compilation.

   If no switches are specified, the compiler produces a .REL file, with the same filename as the last source file in the list. The user must use a /LISTING switch to get a listing (see Section 16.2.3 for a description of FORTRAN-20 compiler switches). COMPILE is optional if the command begins with a switch or begins with a file specification that cannot be confused with the words EXIT, HELP, TAKE, or COMPILE.

3. TAKE <file-spec> [/ECHO]

   The compiler reads the file specified as the command input stream. The TAKE command is legal within 'take' files. The maximum nesting depth is 10 'take' files.

   The /ECHO switch optionally causes commands to be displayed on TTY: as they are executed. The optional /NOECHO switch can be used on a nested take command to cancel the affect of the /ECHO switch while processing that nested command file.

4. RUN <file-spec> [/OFFSET:<integer>]

   This command runs another program (for example, LINK). It causes an exit from the FORTRAN compiler and the start of execution of the program indicated by the file specification, with the additional option of starting that program at an OFFSET relative to the normal starting address.

5. HELP

   This command prints information on the user's terminal about how to use the FORTRAN compiler.

6. EXIT

   This command exits from FORTRA.

You are given the following options:

1. Filename specifications consist of the following:

   An optional device name (the default device is DSK:)

   An optional directory name

   An up to 39 alphanumeric character filename

   An optional up to 39 alphanumeric character file type

   An optional generation number that identifies the version of the file

   An optional file attribute to specify distinctive characteristics of a file specification

   (Refer to the *TOPS-20 User's Guide*)

NOTE

> LINK is restricted to 6-character filenames and 3-character extensions.

2. You may specify more than one source file in the compilation command string. These files will be concatenated by the compiler and treated as one source file. The name of the last source file is used as the default name of the object and listing files. If the last source file does not have a name (such as, TTY:), FORTRAN-OUTPUT is used as the default filename.

3. More than one program unit may be contained in a single source file.

4. A program unit may consist of more than one file.

5. If no /LISTING switch is specified (see Section 16.2.3), no listing is generated.

6. If no extension is given, the defaults are the following for the respective files:

   .LST (listing) if the /CROSSREF switch is not specified

   .CRF (cross reference) if the /CROSSREF switch is specified (see Section 16.2.3)

   .REL (relocatable binary)

   .FOR (source)

16.2.3 TOPS-20 Compiler Commands Switches

Switches to the FORTRAN-20 compiler are accepted anywhere in the command string. They are totally position and file independent. Table 16-2 lists the switches.

Table 16-2: FORTRAN-20 Compiler Switches

Switch	Meaning	Defaults
/ABORT	Causes the compiler to exit at the end of a compilation that contains errors.	OFF
/BINARY[:relfile]	Indicates that a relocatable binary file is generated. You can optionally specify the file specification.	ON
/CROSS-REFERENCE	Generates a file with extension .CRF that can be input to the CREF program.	OFF
/DEBUG[keys:]	Includes debugging information in your program (see Section 16.3).	NONE
/DFLOATING	Indicates that double-precision numbers are stored in D-floating format. (See Section 3.4.)	ON
/ECHO-OPTION	Echo switches selected from the SWITCH.INI file.	OFF
/EXPAND	Includes the octal-formatted version of the object file in the listing.	OFF
/EXTEND[keys:]	Indicates extended addressing. Programs can have up to 30 sections of code and data (see Section 16.5).	OFF
/F66	The FORTRAN-66 standard rules apply for DO loops and EXTERNAL statements. (Same function as the /NOF77 switch.)	OFF
/F77	The FORTRAN-77 standard rules apply for DO loops and EXTERNAL statements.	ON
/FLAG-NON-STANDARD	Invokes the compatibility flagger (see Section 16.6).	OFF
/GFLOATING	Indicates that double-precision numbers are stored in G-floating format. (See Section 3.4.)	OFF
/INCLUDE	Compiles a D in card column 1 as space.	OFF
/LISTING[:listfile]	Indicates a list file will be generated. You can optionally specify the file specification.	OFF

Table 16-2: FORTRAN-20 Compiler Switches (Cont.)

Switch	Meaning	Defaults
/LNMAP	Produces a line number/octal location map in the listing only if /MACHINE-CODE was not specified.	OFF
/MACHINE-CODE	Adds the mnemonic translation of the object code to the listing file. This command will cause a default /LISTING.	OFF
/NOBINARY	Indicates that no relocatable binary file is generated.	OFF
/NOF77	The FORTRAN-66 standard rules apply for DO loops and EXTERNAL statements. (Same function as the /F66 switch.)	OFF
/NOFLAG-NON-STANDARD	Indicates that no compatibility flagging will be done (see Section 16.6).	ON
/NOERRORS	Does not print error messages on the terminal.	OFF
/NOEXTEND	Indicates extended addressing is not in effect (see Section 16.5).	ON
/NOWARN	Suppresses warning messages (see Section 16.4).	NONE
/OPTIMIZE	Performs global optimization.	OFF
/OPTION[:option]	Only read lines from the SWITCH.INI file that start with FORTRA:option.	OFF
/SYNTAX	Performs syntax check only.	OFF

Each switch must be preceded by a slash (/). Switch names need only contain those letters that are required to make the switch name unique. You are encouraged to use at least three letters to prevent conflict with switches in future implementations.

NOTE

> When using switches in control files, you are encouraged to type the full name of the switch.

Example:

```
@FORTRA
FORTRAN>SFILE+S2FILE/MAC/LIST:LFILE
```

The /MAC switch will cause the MACRO code generated for SFILE and S2FILE to appear in LFILE.LST. An relocatable binary file will be created with the name S2FILE.REL.

All switches, used or implied, are printed at the top of each listing page.

16.3 THE /DEBUG SWITCH

The /DEBUG switch tells FORTRAN to compile a series of debugging features into your program. Several of these features are specifically designed to be used with the FORTRAN debugging program (FORDDT). Refer to Chapter 17 for more information. By using the DEBUG switch arguments listed in Table 16-3, you can include specific debugging features.

The form of the /DEBUG switch is:

```
/DEBUG:arg
```

or

```
/DEBUG:(arg1,arg2,...)
```

Table 16-3: Arguments to /DEBUG Switch

Arguments	Meaning
DIMENSIONS	Includes dimension information in .REL file for FORDDT.
TRACE	Generates references to FORDDT required for its trace features (automatically activates LABELS).
LABELS	Generates a label for each statement of the form <line-number>L. (This option can be used without FORDDT.)
INDEX	Forces DO LOOP indexes to be stored at the beginning of each iteration rather than held in a register for the duration of the loop. In addition, this switch forces all function values to be stored in memory prior to return from the function. If this switch is specified, you can set a FORDDT pause on the RETURN statement (see Section 13.4.4) and then examine the value to be returned.
BOUNDS	Generates the bounds checking code for all array references and substring references. Bounds violations will produce run-time error messages. Note that the technique of specifying dimensions of 1 for subroutine arrays will cause bounds check errors. (You may use this option without FORDDT.)
ARGUMENTS	Generates type checking information at load time for actual argument types and associated dummy argument types. Type violations will produce non-fatal load-time error messages. This switch also performs type checking at compile-time for statement functions.
NONE	Do not include any debug features.
ALL	Enable all debugging aids.

Options available with the /DEBUG arguments are:

1. No debug features - Either do not specify the /DEBUG switch or include /DEBUG:NONE.

2. All debug features - Either /DEBUG or /DEBUG:ALL.

3. Selected features - Either a series of modified switches, that is:

   ```
   /DEBUG:BOU/DEBUG:LAB
   ```

   or a list of modifiers

   ```
   /DEBUG:(BOU,LAB,...)
   ```

4. Exclusion of features - If you wish all but one or two modifiers and do not wish to list them all, you can use the prefix "NO" before the switch you wish to exclude. The exclusion of one or more features implicitly includes all the others, that is, /DEBUG:NOBOU is the same as /DEBUG:(DIM,TRA,LAB,IND,ARG).

If you include more than one statement on a single line, only the first statement will receive a label (/DEBUG:LABELS) or FORDDT reference (/DEBUG:TRACE).

NOTE

> If a source file contains line sequence numbers that occur more than once in the same subprogram, the /DEBUG option cannot be used. Also, the /DEBUG option and the /OPTIMIZE option cannot be used at the same time.

The following formulas may be used to determine the increases in program size that will occur as a result of the addition of various /DEBUG options.

DIMENSIONS
For each array, 3+3*N words where N is the number of dimensions, and up to three constants for each dimension.

TRACE
One instruction per executable statement.

LABELS
No increase.

INDEX
One instruction per inner loop plus one instruction for some of the references to the index of the loop. Also one instruction per subprogram.

BOUNDS
For each array, the formula is the same as DIMENSIONS.

For each reference to an array element, 5+N words additional words are generated, where N is the number of dimensions in the array. If you do not specify BOUNDS, approximately 1+3*(N-1) words will be used. For each reference to a substring, add 5 words.

ARGUMENTS
No increase.

If the /DEBUG:ARGUMENTS switch argument is specified, type checking is performed at LINK time for calls to external programs and at compile time for calls to statement functions. Non-fatal error messages are issued at LINK time for the following cases:

1. If the number of arguments in the called subprogram and the calling program unit are not equal.

2. If the length of an array or character scalar actual argument is less than that of the corresponding dummy argument. (This is checked only if the length of the actual is known at compile time.)

3. If the associations 'of actual argument data types with dummy argument data types are other than those indicated as legal in Table 16-4.

4. If a non-routine name is passed where a routine name is expected, or a routine name is passed where a non-routine name is expected.

Non-fatal error messages are issued at compile time for the following cases (only for statement functions):

1. If a non-routine name is passed where a routine name is expected, or a routine name is passed where a non-routine name is expected.

2. If the length of the actual character expression being passed to a statement function is less than that of the corresponding character expression dummy argument.

Table 16–4: Legal Dummy and Actual Argument Associations

Actual Argument Type

Dummy Argument Type	Alternate Return Label	Logical	Integer	Real	D-floating	G-floating	Complex	Character	Octal	Hollerith	Double Octal
Alternate Return Label	X										
Logical		X							X	X	
Integer			X						X	X	
Real				X					X	X	
D-floating					X					X	X
G-floating						X				X	X
Complex							X			X	X
Character								X			

X indicates legal associations. All others will cause a warning to be issued if DEBUG:ARGUMENTS is specified.

16.4 THE /NOWARN SWITCH

The /NOWARN switch is used to suppress compiler warning messages. If this switch is used with no arguments, all warning messages are suppressed. The /NOWARN switch may also be used with arguments as shown in Table 16-5.

Table 16-5: Arguments to /NOWARN Switch

Arguments	Meaning
ALL	Suppress all warning messages.
NONE	Do not suppress warning messages.
xxx	Where xxx is the three character error mnemonic for the error message to be suppressed. This is the three letters that follow %FTN, for example %FTNABD.

For example,

```
/NOWARN:LID
```

will suppress all warnings of an identifier having more than six characters.

A list of arguments is also allowed. For example,

```
/NOWARN:(LID,DIM)
```

would suppress both LID and DIM types of warning messages.

(See Appendix C for a list of all compiler error mnemonics.)

16.5 THE /EXTEND SWITCH (TOPS-20 ONLY)

Support for extended addressing is almost completely transparent; a program compiled with the /EXTEND switch will use extended addressing without requiring changes to the FORTRAN source program.

When /EXTEND is specified, arrays and COMMON blocks can extend across multiple sections. Executable code can also reside in multiple sections, with the restriction that a single subprogram must not cross a section boundary.

The /EXTEND switch can be specified without arguments to use the default extended address space layout. This is suitable for most applications in which the executable code fits within a single section, but which may employ data structures that require more than a section of memory. When such an application is compiled /EXTEND without further arguments, a default memory layout is used that depends on the default size settings for arrays and strings:

Table 16-6: /EXTEND Default Memory Layout

Section	Pages	Contents
1	0-477	Executable code Argument blocks Literals Non-COMMON scalars Non-COMMON arrays smaller than 10,000 words Non-COMMON strings smaller than 10,000 words
1	500-577	FOROTS
1	600-777	Reserved
2-31	0-777	COMMON blocks Arrays larger than 10,000 words Strings larger than 10,000 words

Additional arguments to /EXTEND (COMMON:, DATA:, PSECT:, and CODE), in conjunction with the /SET switch to LINK, can be supplied to override these defaults and direct specific placement of COMMON blocks, non-COMMON arrays and scalars, and executable code. For example, a decimal argument can be specified for the DATA: keyword, which overrides the 10,000 word default minimum size for large arrays and character scalars. /EXTEND:COMMON can be used to direct specific named COMMON blocks to non-default sections of extended memory. (Section 16.5.3 describes the /EXTEND arguments.)

16.5.1 /EXTEND and Applications with Large Arrays

The simplest usage of extended addressing is for applications in which the executable code fits in one section. These applications may have very large arrays or common blocks. In many cases, these applications can be compiled with the /EXTEND switch with no arguments, using the default memory layout in Table 16-6.

In some cases, you may need to use the DATA or COMMON arguments to /EXTEND to redirect the placement of variables in areas when default placements cannot be used. Specifying a smaller /EXTEND:DATA size may be necessary if the total size of non-COMMON scalars and arrays causes them to overlap FOROTS pages. If such overlap occurs, program execution will terminate with the error messages:

```
"?Can't get FORO10.EXE"
```

and

```
"?Can't overlay existing pages"
```

A smaller DATA size will force these data structures into a non-FOROTS section.

16.5.2 /EXTEND and Applications with Large Executable Code

A more complex use of extended addressing is one where the application's executable code is larger than a section. In order to build an application that has more than one section of executable code, you have to specify which program units will be linked together in each section.

When /EXTEND is specified, the extended code will be in three PSECTS, one analogous to the present hiseg, one analogous to the present lowseg, and one containing the large variables. If the user does not specify the extended addressing switch, FORTRAN will generate a TWOSEG REL file as it always has. The three PSECTS generated under /EXTEND are:

1. The "large data area" psect (.LARG.) has a maximum size of 30 sections. It will consist of user-specified COMMON blocks, arrays and character scalars that are larger than a user-specified size (or default).

   There is no restriction on the size of an individual array or COMMON block, beyond the restriction on the total size of the large data area.

2. The "small data area" psect (default name .DATA.) of a program consists of user-specified COMMON blocks, and scalars and arrays that are smaller than a user-specified size (or default). If FORDDT is used, it will reside in the .DATA. psect.

3. The "executable code and sharable data area" psect (default name .CODE.) of a program consists of all executable code, argument blocks and literals. Library functions and subroutines used by the program are placed in the .CODE. psect. This does not include space used by SORT, which occupies its own section.

For each "executable code area" (code psect) there will be a corresponding "small data area" (data psect). The combined size of these two areas must not exceed 256K words. The default psect names .DATA. and .CODE. can be changed at compile time by the /EXTEND:PSECT command switch (see Section 16.5.3).

16.5.2.1 /EXTEND PSECT Placement - The three psects will be set up according to the table below:

Psect	Default Origin	Attributes
data	1001000	Single section, Non-zero section, Concatenated, Writable
code	1300000	Single section, Non-zero section, Concatenated, Read-only
.LARG.	2000000	Non-zero section, Concatenated, Writable

You can alter the default psect origins when loading programs by giving a /SET switch to LINK (see the LINK Reference Manual). This may be necessary if a program has too much code or local data to fit in the regions allocated for them by the default psect origins. This would cause the psects to overlap, and would be indicated by a LINK warning message (%LNKPOV). Altering the defaults is also useful if the user wants to reserve one or more sections for some purpose (such as telling the monitor where dynamic libraries should be loaded).

The first page (locations 000-777) of any section that contains code is reserved for use by FORDDT. The LINK /SET switch should not specify a psect origin less than 1000 for any such section.

16.5.2.2 **Building Large-Code Applications** - When compiling large-code applications, the following considerations apply:

1. The code and small data psects for a single program unit must always reside together in the same section, since the PC will not advance across section boundaries, and the small data area is assumed to be in the same section.

2. You will need to use the /EXTEND:CODE keyword. This specifies that the object code emitted by the compiler assumes that all subprograms that it calls may be in a separate section.

3. You must also use the PSECT argument to /EXTEND to specify the psect names for code and small data. If you wish to link the program units in several different compilations (source files) together in the same section, you should use the same psect names for those program units.

At link-time, you must specify the starting address for each psect using the /SET switch, according to the following rules:

1. The code and small data psects for a program unit must be linked in the same section with each other.

2. If a section contains any executable code, page 0 of that section is reserved for FORDDT and FOROTS.

3. You must always allocate space for the .CODE. and .DATA. psects, since FORLIB routines will be linked in these psects.

4. Pages 500-577 of the section that contains .CODE. and .DATA. are reserved for FOROTS.

5. Pages 600-777 are reserved for FOROTS I/O buffers and DDT.

6. You must always allocate space for the .LARG. psect.

16.5.3 Arguments to /EXTEND

By using the /EXTEND switch arguments listed in Table 16-7, you can include specific extended addressing features.

The form of the /EXTEND switch is:

```
/EXTEND:arg

or

/EXTEND:(arg1,arg2,...)
```

Table 16-7: Arguments to /EXTEND Switch

Arguments	Meaning
CODE	Specifies that the object code produced by the compiler has to assume that any subprogram that it calls could be in a separate section. NOCODE is the default.
COMMON[:name] or COMMON:(name,...)	Without a common block name specified, causes all common blocks that have not already been allocated by /EXTEND:[NO]COMMON to be allocated in the .LARG. psect. This is the default. Individual common blocks can be placed explicitly in .LARG. by putting their names in a list after COMMON:. When you explicitly place an individual common block in .LARG., any common blocks that have not already been allocated by /EXTEND:[NO]COMMON are placed in the small data psect.
DATA[:decimal number]	Specifies a decimal argument that is the minimum size (in words) for non-common arrays and character scalars, which will be allocated to the .LARG. psect. The default is 10,000.
NOCODE	Allows the compiler to assume that all of the code will be in the same section. A program compiled with the NOCODE argument cannot call any subprograms compiled with the CODE argument. This is the default.
NOCOMMON[:name] or NOCOMMON:(name,...)	Without a common block name specified, causes all common blocks to be allocated in the data psect. Individual common blocks can be placed explicitly in the data psect by putting their names in a list after COMMON:. COMMON is the default.
NODATA	Specifies that all non-common variables will reside in .DATA. This is equivalent to DATA:1073741823, which excludes all variables from .LARG.
PSECT[:[data psect] [:[code psect]]]	Allows users to set the code and data psect names explicitly (the large data psect is always called .LARG.) If PSECT is specified with one argument, that argument becomes the name of the small data area psect. Any second argument becomes the name of the code psect. This allows separate program units to be put in separate psects, then the psect can be placed in different sections with the /SET switch at LINK time (see the LINK Reference Manual).

NOTE

When using the PSECT argument, the small data psect and code psect for any given program unit must be loaded into the same memory section.

16.5.4 Linking With TWOSEG REL Files

If a main program unit compiled with /EXTEND is linked with subprogram units (FORTRAN or not) that were compiled for non-extended use, then LINK will automatically place the LOWSEG of non-extended units in the .DATA. psect, and the HISEG of such units into the .CODE. psect.

A program compiled with /EXTEND can call a subprogram that is not compiled with /EXTEND; however, it is illegal for a subprogram that is not compiled with /EXTEND to call a subprogram that is.

Programs that were compiled by old versions of FORTRAN-10/20 (prior to Version 7) will not work if loaded in a non-zero section.

Most MACRO routines written for non-extended use will require conversion to run in non-zero sections. Data structures accessed with 18-bit address fields, indexed and indirect words, stack pointers and some monitor calls may need modification to perform correctly in extended sections. See the TOPS-20 Monitor Call User's Guide for more information concerning extended MACRO programs.

16.6 THE /FLAG (/FLAG-NON-STANDARD) SWITCH

The /FLAG switch invokes the compatibility flagger.

NOTE

> For TOPS-20 systems, the full switch names are /FLAG-NON-STANDARD and /NOFLAG-NON-STANDARD; however /FLAG and /NOFLAG work.

This feature provides warning messages for language elements used that are the following:

- Extensions to the ANSI FORTRAN-77 standard
- Features not found in VAX FORTRAN
- Features that could cause logically different results when used on the VAX FORTRAN system

NOTE

> VAX FORTRAN is used on the VAX/VMS operating system.

This allows the flagging of any element that could cause conversion problems for programs written on the TOPS-10/20 system that might be compiled and executed on a VAX/VMS system or an ANSI-compatible system. This includes problems that could occur at object time, as well as compilation incompatibilities.

By using the FLAG switch arguments listed in Table 16-8, you can specify which features to flag.

The form of the /FLAG switch is:

```
/FLAG[:arg]

or

/FLAG[:(arg1,arg2,...)]
```

Table 16-8: Arguments to /FLAG Switch

Arguments	Meaning
ALL	Gives warning messages for language elements incompatible with both FORTRAN-77 and VAX FORTRAN.
ANSI	Gives warning messages whenever a language element is an extension to the FORTRAN-77 standard.
NOANSI	Does not flag FORTRAN-77 extensions.
NONE	Does not flag.
NOVMS	Does not flag VAX incompatibilities.
VMS	Gives warning messages whenever a language element is incompatible with VAX FORTRAN.

If no /FLAG switch is specified, no flagging is done. If no arguments are given with the /FLAG switch, then flagging is done for both FORTRAN-77 and VAX incompatibilities.

The /NOFLAG switch indicates that no flagging will be done.

16.7 READING A FORTRAN COMPILER LISTING

When you request a listing from the FORTRAN compiler, it may contain the following information, depending on the switches used at compilation time:

1. A printout of the source program plus an internal sequence number assigned to each line by the compiler. This internal sequence number is referenced in any error or warning messages generated during the compilation. If the input file is line-sequenced, the number from the file is used. If code is added by means of the INCLUDE statement, all INCLUDEd lines will have an asterisk (*) appended to their line-sequence number.

2. A summary of the names and relative program locations (in octal) of scalars and arrays (including unreferenced character scalars and arrays) in the source program plus compiler-generated variables.

3. All COMMON blocks and the relative locations (in octal) of the variables in each COMMON block.

4. A listing of all equivalenced variables or arrays and their relative locations. Note that all equivalenced variables that are also in COMMON are listed only as being in COMMON.

5. A listing of the subprograms referenced (both user-defined and FORTRAN-defined library functions).

6. A summary of temporary locations generated by the compiler.

7. A heading on each page of the listing containing the program unit name (MAIN., .BLOCK, program, subroutine or function, principal entry), the input filename, the list of compiler switches, and the date and time of compilation.

8. If you used the /MACRO switch, a mnemonic printout of the generated code (in a format similar to MACRO) is appended to the listing. This section has four fields:

   LINE: This column contains the internal sequence number of the line corresponding to the mnemonic code. It appears on the first line of the code sequence associated with that internal sequence number. An asterisk indicates a compiler inserted line.

   LOC: The relative location in the object program of the instruction.

   LABEL: Any program or compiler generated label. Program labels have the letter "P" appended. Labels generated by the compiler are followed by the letter "M". Labels generated by the compiler and associated with the /DEBUG:LABELS switch consist of the internal sequence number followed by an "L".

   GENERATED CODE: The MACRO mnemonic code.

   If you use the /LNMAP switch and do NOT use the /MACRO switch, a line number/octal location map is appended to the listing. This section lists the line numbers in increments of 10 on subsequent lines and each number from 0 through 9 for each line in adjacent columns. The numbers appearing inside the matrix are the relative octal locations of the statements in the FORTRAN program unit.

   For example, to find the relative octal location of line number 001043, find the row marked 001040 and then column 3 on that line. The number in that place is the desired relative location. This listing can be very large and sparse for line-numbered files with large increments, such as those produced by the editor SOS on TOPS-10 (or the editor EDIT on TOPS-20).

NOTE

> A single FORTRAN line can produce multiple machine instructions. In this case the line number map lists only the first location.

9. A list of all argument blocks generated by the compiler. A zero argument appears first followed by argument blocks for subroutine calls and function references (in order of their appearance in the program). Argument blocks for all I/O operations follow this.

10. FORMAT statement listings.

11. A summary of errors detected or warning messages issued during compilations.

16.7.1 Compiler-Generated Variables

In certain situations the compiler generates internal variables. Knowing what these variables represent can help you read the macro expansion. The variables are of the form:

.letter digit digit digit digit

The function of these variables can be determined by the first letter of the variable name as described below:

Letter	Function of Variable
A	Register save area.
D	Compile-time constant character descriptor
F	Arithmetic statement function formal parameters.
I	Result of a DO LOOP initial value expression or parameter of an adjustable dimensioned array.
O	Result of a common subexpression or constant computation.
Q	Temporary storage for expression values.
R	Result of reduced operator strength expression.
S	Result of the DO LOOP step size expression of computed iteration count for a loop.

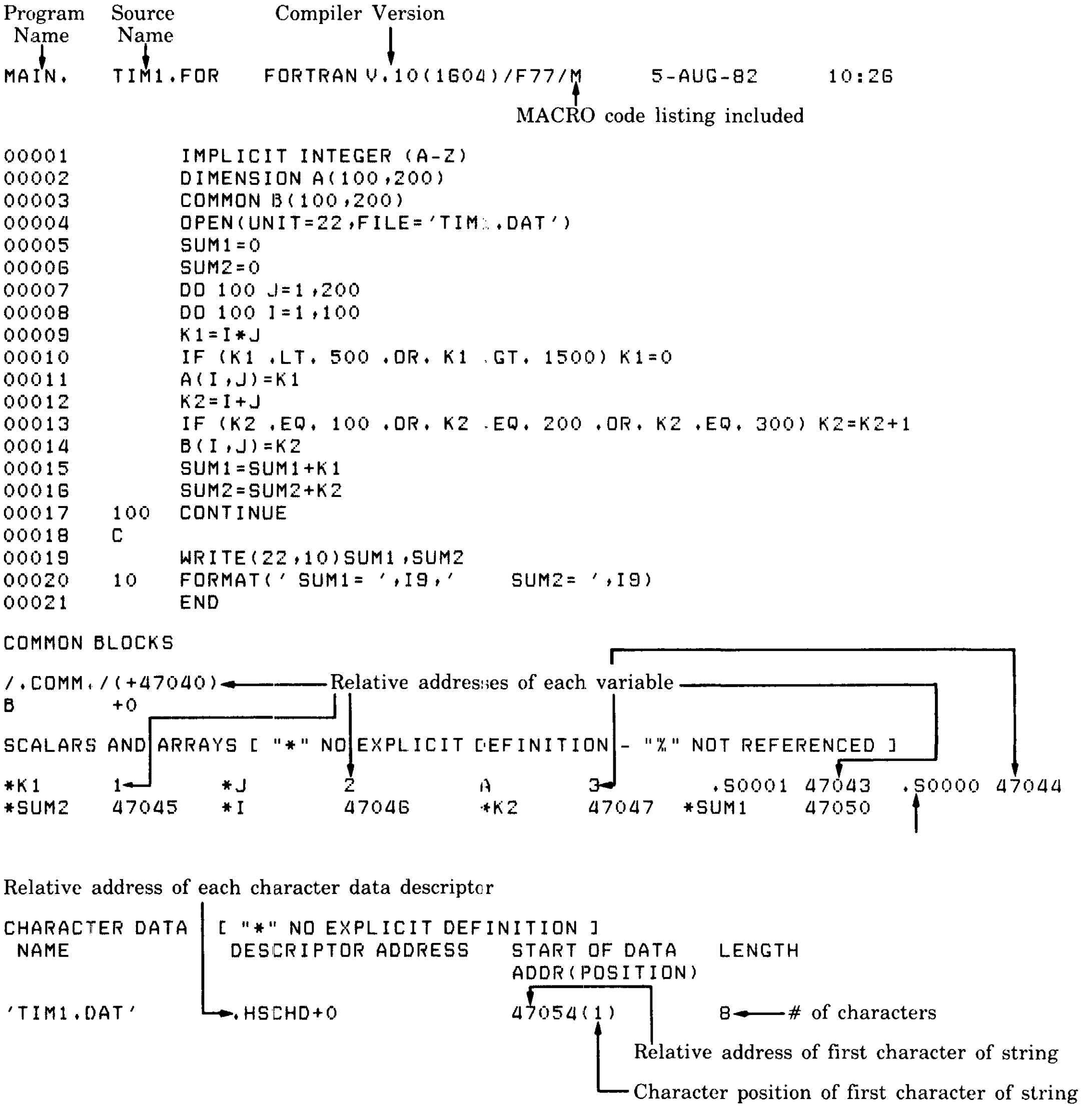
For example:

```
.S0001
```

You may find these variables on the listing under SCALARS and ARRAYS.

The following examples show listings where all of these features are pointed out.

Example 1:



Program Name

Source Name

Compiler Version

```
MAIN.     TIM1.FOR    FORTRAN V.10(1604)/F77/M      5-AUG-82       10:26
```

MACRO code listing included

```
00001          IMPLICIT INTEGER (A-Z)
00002          DIMENSION A(100,200)
00003          COMMON B(100,200)
00004          OPEN(UNIT=22,FILE='TIM1.DAT')
00005          SUM1=0
00006          SUM2=0
00007          DO 100 J=1,200
00008          DO 100 I=1,100
00009          K1=I*J
00010          IF (K1 .LT. 500 .OR. K1 .GT. 1500) K1=0
00011          A(I,J)=K1
00012          K2=I+J
00013          IF (K2 .EQ. 100 .OR. K2 .EQ. 200 .OR. K2 .EQ. 300) K2=K2+1
00014          B(I,J)=K2
00015          SUM1=SUM1+K1
00016          SUM2=SUM2+K2
00017     100  CONTINUE
00018     C
00019          WRITE(22,10)SUM1,SUM2
00020     10   FORMAT(' SUM1= ',I9,'     SUM2= ',I9)
00021          END

COMMON BLOCKS

/.COMM./(+47040)
B       +0

SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*K1       1       *J        2        A        3        .S0001  47043   .S0000  47044
*SUM2     47045   *I        47046    *K2      47047    *SUM1   47050
```

Relative addresses of each variable

Relative address of each character data descriptor

```
CHARACTER DATA   [ "*" NO EXPLICIT DEFINITION ]
 NAME              DESCRIPTOR ADDRESS    START OF DATA     LENGTH
                                         ADDR(POSITION)

'TIM1.DAT'         .HSCHD+0              47054(1)          8
```

# of characters

Relative address of first character of string

Character position of first character of string

Internal sequence number on first instruction for this source line

Octal displacement of instruction

```
LINE     LOC      LABEL    GENERATED CODE

         0                 JFCL     0,0
         1                 JSP      16,RESET.
         2                          0,0
4        3                 XMOVEI   16,2M
         4                 PUSHJ    17,OPEN.
5        5                 SETZB    2,SUM1
6        6                 MOVEM    2,SUM2
7        7                 MOVE     2,[777470000001]
         10                HLREM    2,.S0000
         11       3M:
                           HRRZM    2,J
8        12       4M:
                           MOVE     2,[777634000001]
9        13       5M:
                           MOVE     3,J
         14                IMULI    3,0(2)
         15                MOVEM    3,K1
10       16                CAIL     3,764
         17                CAILE    3,2734
         20                JRST     0,7M
         21                JRST     0,6M
10       22       7M:
                           SETZB    4,K1
11       23       6M: <------------------------- Compiler generated label
                           MOVEI    3,144
         24                IMUL     3,J
         25                ADDI     3,0(2)
         26                MOVE     4,K1
         27                MOVEM    4,A-145(3)
12       30                MOVE     3,J
         31                ADDI     3,0(2)
         32                MOVEM    3,K2
13       33                MOVE     5,K2
         34                CAIE     5,144
         35                CAIN     5,310
         36                JRST     0,9M
         37       10M:
                           CAIN     5,454
13       40       9M:
                           AOS      3,K2
14       41       8M:
                           MOVEI    3,144
         42                IMUL     3,J
         43                ADDI     3,0(2)
         44                MOVE     5,K2
         45                MOVEM    5,B-145(3)
15       46                ADDM     4,SUM1
16       47                ADDM     5,SUM2
17       50       100P: <----------------------- Program label
                           AOBJN    2,5M
         51                HRRZM    2,I
         52                AOS      2,J
         53                AOSGE    0,.S0000
         54                JRST     0,4M
19       55                XMOVEI   16,11M
         56                PUSHJ    17,OUT.
         57                XMOVEI   16,12M
         60                PUSHJ    17,IOLST.
21       61                XMOVEI   16,1M
         62                PUSHJ    17,EXIT.
```

```
ARGUMENT BLOCKS:←——— Function, subroutine, and FOROTS argument blocks

        63                     0,,0
        64        1M:          0,,0
        65                     777776,,0
        66        2M:          436000,,26
        67                     406640,,.HSCHD+0
        70                     777776,,0
        71        11M:         401000,,26
        72                     402340,,10P
        73                     777775,,0
        74        12M:         401100,,SUM1
        75                     401100,,SUM2
        76                     4000,,0


FORMAT STATEMENTS (IN LOW SEGMENT):

20      47056               6
        47057    10P:       (' SUM1= ',I9,'    SUM2= ',I9)

MAIN.   [ No errors detected ]←——————Summary of errors
```

Example 2:

```
MAIN.   TIM1.FOR    FORTRAN V.10(1604)/F77/L       5-AUG-82      10:28

00001         IMPLICIT INTEGER (A-Z)
00002         DIMENSION A(100,200)
00003         COMMON B(100,200)
00004         OPEN(UNIT=22,FILE='TIM1.DAT')
00005         SUM1=0
00006         SUM2=0
00007         DO 100 J=1,200
00008         DO 100 I=1,100
00009         K1=I*J
00010         IF (K1 .LT. 500 .OR. K1  GT. 1500) K1=0
00011         A(I,J)=K1
00012         K2=I+J
00013         IF (K2 .EQ. 100 .OR. K2  EQ. 200 .OR. K2 .EQ. 300) K2=K2+1
00014         B(I,J)=K2
00015         SUM1=SUM1+K1
00016         SUM2=SUM2+K2
00017   100   CONTINUE
00018   C
00019         WRITE(22,10)SUM1,SUM2
00020   10    FORMAT(' SUM1= ',I9,'    SUM2= ',I9)
00021         END

COMMON BLOCKS

/.COMM./(+47040)
B         +0

SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*K1     1         *J         2         A       3          .S0001 47043    .S0000    47044
*SUM2   47045     *I         47046     *K2     47047      *SUM1  47050

LINE NUMBER/OCTAL LOCATION MAP←——— Requested with /LNMAP

      : 0       1       2       3       4       5       6       7       8       9
------:-----------------------------------------------------------------------------
      :
00000 :                                 3       5       6       7      12      13
00010 : 16      23      30      33      41      46      47      50              55
00020 :         61
MAIN.   [ No errors detected ]
```

Line #11 starts at octal offset 23 (from the previous example, note that line 11 uses locations 23 through 27, but only the first location is shown here).

Example 3:

```
MAIN.    TIM1.FOR    FORTRAN V.10(1604)/F77/OPT/M  5-AUG-82      10:30

00001           IMPLICIT INTEGER (A-Z)
00002           DIMENSION A(100,200)
00003           COMMON B(100,200)
00004           OPEN(UNIT=22,FILE='TIM1.DAT')
00005           SUM1=0
00006           SUM2=0
00007           DO 100 J=1,200
00008           DO 100 I=1,100
00009           K1=I*J
00010           IF (K1 .LT. 500 .OR. K1 .GT. 1500) K1=0
00011           A(I,J)=K1
00012           K2=I+J
00013           IF (K2 .EQ. 100 .OR. K2 .EQ. 200 .OR. K2 .EQ. 300) K2=K2+1
00014           B(I,J)=K2
00015           SUM1=SUM1+K1
00016           SUM2=SUM2+K2
00017     100   CONTINUE
00018     C
00019           WRITE(22,10)SUM1,SUM2
00020     10    FORMAT(' SUM1= ',I9,'    SUM2= ',I9)
00021           END

COMMON BLOCKS

/.COMM./(+47040)
B        +0

SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

                         Optimizer created variables
*K1      1          .R0001   2          .R0000  3       *J       4       A           5
 .S0001  47045      .S0001   47046      *SUM2   47047   *I       47050   .00001      47051
*K2      47052      *SUM1    47053

CHARACTER DATA [ "*" NO EXPLICIT DEFINITION ]
 NAME            DESCRIPTOR ADDRESS      START OF DATA    LENGTH
                                         ADDR(POSITION)

'TIM1.DAT'      .HSCHD+0                 47060(1)         8
```

```
LINE     LOC      LABEL     GENERATED CODE

         0                  JFCL       0,0
         1                  JSP        16,RESET.
         2                             0,0
4        3                  XMOVEI     16,4M
         4                  PUSHJ      17,OPEN.
6        5                  SETZB      10,11
*        6                  MOVEI      12,144
         7                  MOVEM      12,,R0001
7        10                 MOVNI      12,310
         11                 MOVEI      7,1
         12                 MOVEM      12,,S0000
*        13       5M:
                            MOVE       6,7
8        14                 MOVE       2,[777634000001]
*        15       6M:
                            MOVEI      4,0(2)
         16                 ADD        4,,R0001
9        17                 MOVE       5,6
10       20                 CAIL       5,764
         21                 CAILE      5,2734
         22                 JRST       0,8M
         23                 JRST       0,7M
10       24       8M:
                            MOVEI      5,0
11       25       7M:
                            MOVEM      5,A-145(4)
12       26                 MOVE       3,7
         27                 ADDI       3,0(2)
13       30                 CAIE       3,144
         31                 CAIN       3,310
         32                 JRST       0,10M
         33       11M:
                            CAIN       3,454
13       34       10M:
                            ADDI       3,1
14       35       9M:
                            MOVEM      3,B-145(4)
15       36                 ADD        11,5
16       37                 ADD        10,3
*        40                 ADD        6,7
17       41       100F:
                            AOBJN      2,6M
         42                 HRRZM      12,144
*        43                 MOVEI      12,144
         44                 ADDM       12,,R0001
*        45       1M:
                            ADDI       7,1
         46                 AOSGE      0,,S0000
         47                 JRST       0,5M
         50                 MOVEM      7,J
*        51                 MOVEM      11,SUM1
*        52                 MOVEM      10,SUM2
*        53                 MOVEM      5,K1
*        54                 MOVEM      3,K2
19       55                 XMOVEI     16,12M
         56                 PUSHJ      17,OUT.
*        57                 XMOVEI     16,13M
         60                 PUSHJ      17,IOLST.
21       61       2M:
                            XMOVEI     16,3M
         62                 PUSHJ      17,EXIT.
```

Asterisks indicate optimizer generated statements

```
ARGUMENT BLOCKS:

        63                        0,,0
        64      3M:               0,,0
        65                        777776,,0
        66      4M:               436000,,26
        67                        406640,,.HSCHD+0
        70                        777776,,0
        71      12M:              401000,,26
        72                        402340,,10P
        73                        777775,,0
        74      13M:              401100,,SUM1
        75                        401100,,SUM2
        76                        4000,,0

FORMAT STATEMENTS (IN LOW SEGMENT):

20      47062                     6
        47063   10P:              (' SUM1= ',I9,'   SUM2= ',I9)

MAIN.   [ No errors detected ]
```

Example 4:

```
MAIN.     TIM1.FOR     FORTRAN V.10(2530)/F77/M/EXT       24-APR-85      16:27

00001            IMPLICIT INTEGER (A-Z)
00002            DIMENSICN A(100,200)
00003            COMMON B(100,200)
00004            OPEN(UNIT=22,FILE='TIM1.DAT')
00005            SUM1=0
00006            SUM2=0
00007            DO 100 J=1,200
00008            DO 100 I=1,100
00009            K1=I*J
00010            IF (K1 .LT. 500 .OR. K1 .GT. 1500) K1=0
00011            A(I,J)=K1
00012            K2=I+J
00013            IF (K2 .EQ. 100 .OR. K2 .EQ. 200 .OR. K2 .EQ. 300) K2=K2+1
00014            B(I,J)=K2
00015            SUM1=SUM1+K1
00016            SUM2=SUM2+K2
00017    100     CONTINUE
00018    C
00019            WRITE(22,10)SUM1,SUM2
00020    10      FORMAT(' SUM1= ',I9,'      SUM2= ',I9)
00021            END

COMMON BLOCKS   [ "!" STORED IN .LARG. ]

/.COMM./(+47040!)
B        +0                  Large common block

SCALARS  AND ARRAYS [  "*"  NO EXPLICIT DEFINITION - "%" NOT REFERENCED  ]
                    [  "!" VARIABLE STORED IN .LARG.  ]
                                                          Large variable
*K        1              *J        2                !A        0
 .S0001 3                 .S0000 4                  *SUM2     5
*I        6              *K2       7                *SUM1     10


CHARACTER DATA [  "*"  NO EXPLICIT DEFINITION - "!" VARIABLE STORED IN .LARG.  ]
 NAME            DESCRIPTOR ADDRESS        START OF DATA    LENGTH
                                           ADDR(POSITION)

'TIM1.DAT'      .HSCHD+0                   14(1)            8


LINE     LOC      LABEL    GENERATED CODE

         0                 JFCL     0,0
         1                 JSP      16,RESET.
         2                          0,0
4        3        XMOVEI   16,2M
         4                 PUSHJ    17,OPEN.
5        5                 SETZB    2,SUM1
6        6                 MOVEM    2,SUM2
7        7                 MOVE     2,[777470000001]
         10                HLREM    2,.S0000
         11       3M:
                           HRRZM    2,J
8        12       4M:
                           MOVE     2,[777634000001]
9        13       5M:
                           MOVE     3,J
         14                IMULI    3,0(2)
         15                MOVEM    3,K1
10       16                CAIL     3,764
         17                CAILE    3,2734
         20                JRST     0,7M
         21                JRST     0,6M
10       22       7M:
                           SETZB    4,K1
```

```
11      23      6M:
                        MOVEI   3,144
        24              IMUL    3,J
        25              ADDI    3,0(2)
        26              MOVE    4,K1
        27              MOVEM   4,@[.EFIW A-145(3)]
12      30              MOVE    3,J
        31              ADDI    3,0(2)
        32              MOVEM   3,K2
13      33              MOVE    5,K2
        34              CAIE    5,144
        35              CAIN    5,310
        36              JRST    0,9M
        37      10M:
                        CAIN    5,454
13      40      9M:
                        AOS     3,K2
14      41      8M:
                        MOVEI   3,144
        42              IMUL    3,J
        43              ADDI    3,0(2)
        44              MOVE    5,K2
        45              MOVEM   5,@[.EFIW B-145(3)]
15      46              ADDM    4,SUM1
16      47              ADDM    5,SUM2
17      50      100P:
                        AOBJN   2,5M
        51              HRRZM   2,I
        52              AOS     2,J
        53              AOSGE   0,.S0000
        54              JRST    0,4M
19      55              XMOVEI  16,11M
        56              PUSHJ   17,OUT.
        57              XMOVEI  16,12M
        60              PUSHJ   17,IOLST.
21      61              XMOVEI  16,1M
        62              PUSHJ   17,EXIT.

ARGUMENT BLOCKS:

        63              0,,0
        64      1M:     0,,0
        65              777776,,0
        66      2M:     436100,,[000000000026]
        67              406640,,.HSCHD+0
        70              777776,,0
        71      11M:    401000,,26
        72              402340,,10P
        73              777775,,0
        74      12M:    401100,,SUM1
        75              401100,,SUM2
        76              4000,,0

FORMAT STATEMENTS (IN LOW SEGMENT):

20      17              6
        20      10P:    (' SUM1= ',I9,'    SUM2= ',I9)

MAIN.   [ No errors detected ]




00001
00002
00003
```

NOTE

Note that in the scalars and arrays list, 'NO EXPLICIT DEFINITION' indicates that the variable was never explicitly defined, as in a TYPE or DIMENSION statement. Also, 'NOT REFERENCED' indicates that the variable was declared, but never used, and therefore was never allocated any storage in the program.

Character variables that are declared, but not referenced, appear under the scalars and arrays section of the listing. No storage is allocated for either the character descriptor or the character data.

16.8 ERROR REPORTING

If an error occurs during the initial pass of the compiler (while the actual source code is being read and processed), an error message is printed on the listing immediately following the line in which the error occurred. When pertinent and possible, the error references the internal sequence number of the incorrect line. The error messages along with the statement in error are output to the user terminal.

Example:

```
TYPE DAY.FOR
01000            I=10
01100            IMPLICIT INTEGER (X)
01200            J=I**4
01300            K1
01400            X=I+J+K1
01500     100    CONTNUE
01600       C
01700            TYPE 200,X
01800     200    FORMAT(1X,I8)
01900            END


COMPILE DAY.FOR
FORTRAN:DAY
01100            IMPLICIT INTEGER (X)
%FTNSOD LINE:01100  IMPLICIT statement out of order
01300            K1
?FTNNRC LINE:01300 Statement not recognized
01500   100      CONTNUE
?FTNMSP LINE:01500 Statement name misspelled
01600     ?
?FTNICL LINE:01600 Illegal character C in label field

?FTNFTL   MAIN.        3 fatal errors and 1 warning
```

If errors are detected after the initial pass of the compiler, they appear in the list file after the end of the source listing. They are output to your terminal without the statement in error, but they may reference its internal sequence number.

16.8.1 Fatal Errors and Warning Messages

There are two levels of messages, warning and fatal error. Warning messages are preceded by "%" and indicate a possible problem. The compilation will continue, and the object program may be correct. Fatal errors are preceded by a "?". If a fatal error is encountered in any pass of the compiler, the remaining passes will not be called, and no relocatable binary file will be generated.

Additional errors that would be detected in later compiler passes may not become apparent until the first errors are corrected. It is not possible to generate a correct object program for a source program containing a fatal error.

The format of messages is:

```
?FTNxxx Line:n text
```

or

```
%FTNxxx Line:n text
```

where:

?	indicates a fatal message
%	indicates a warning message
FTN	is the FORTRAN mnemonic
xxx	is the 3-letter mnemonic for the error message
Line:n	is the optional line number where error occurred
text	is the explanation of error

The printing of fatal errors and warning messages on your terminal can be suppressed by the use of the /NOERRORS switch; however, messages will still appear on the listing. The /NOWARN switch will suppress warning messages on both the user terminal and in the listing. Specific warnings can be suppressed by using options to the /NOWARN switch (see Section 16.4).

16.8.2 Message Summary

At the end of the listing file and on the terminal, a message summary is printed after each program unit is compiled. This message has two forms:

1. When one or more messages were issued

```
?FTNFTL  name 1 fatal error and no warnings
         name 2 fatal errors and no warnings
%FTNWRN  name no fatal errors and 1 warning
         name no fatal errors and 2 warnings
```

or

2. When no messages were issued

```
name [No errors detected]
```

where name is the program or subprogram name. Appendix C contains a complete list of fatal errors and warning messages.

16.9 CREATING A SHARABLE HIGH SEGMENT FOR A FORTRAN PROGRAM

For non-extended addressing programs, the FORTRAN compiler always generates two segment code for a program unit. However, by default, the linking loader loads all FORTRAN code into the low segment to allow the sharable run-time system to be bound to the program at run-time rather than at load-time.

This default action of the loader can be overridden by using the LINK switch /OTS:NONSHAR when loading the program. If this switch is given, the linking loader loads the impure code (the data areas) in the low segment, loads the pure code (the machine instructions) in the high segment, and binds a private copy of the run-time system to the program at load-time.

A program loaded with the /OTS:NONSHAR switch can be saved in order to produce an executable file with a sharable high segment using the TOPS-10 SSAVE command or the TOPS-20 EXEC SAVE command. (The LINK switches /SSAVE (TOPS-10) or /SAVE (TOPS-20) can also be used to produce the sharable executable file.) This is an advantage if a large application program is to be run by several users simultaneously. The code unique to the program and the copy of the FORTRAN run-time system that is private to the program is shared between all the program's users.

The possible benefits gained by the users of the application program sharing the high segment of their program containing both user and FOROTS code must be weighed against the loss of not sharing the common copy of FOROTS as all other users.

(See the LINK Reference Manual.)

CHAPTER 17

USING THE FORTRAN INTERACTIVE DEBUGGER (FORDDT)

FORDDT is an interactive program used to debug FORTRAN programs and control their execution. By using the symbols created by the FORTRAN compiler, FORDDT allows you to examine and modify the data and FORMAT statements in your program, set breakpoints at any executable statement or routine, trace your program statement-by-statement, and make use of many other debugging techniques described in this chapter.

Table 17-1 lists all the commands available to the user of FORDDT.

Table 17-1: FORDDT Commands

Command	Purpose
Control Commands	
CONTINUE	Continues execution after a FORDDT breakpoint.
DDT	Enters DDT.
GOTO	Transfers control to some program statement within the open program unit.
NEXT	Traces execution of the program.
START	Begins execution of the FORTRAN program.
STOP	Terminates the program and returns to monitor mode.
Data Access Commands	
ACCEPT	Modifies variables or FORMAT statements.
TYPE	Displays variables. Declarative Commands
CHARACTER	Defines dimensions of character arrays for FORDDT references. (This command is unnecessary if /DEBUG is specified at compile time. See Table 16-3.)

Table 17-1: FORDDT Commands (Cont.)

Command	Purpose
DIMENSION	Defines dimensions of real and integer arrays for FORDDT references. (This command is unnecessary if /DEBUG is specified at compile time. See Table 16-3.)
DOUBLE	Defines dimensions of double-precision and complex arrays for FORDDT references. (This command is unnecessary if /DEBUG is specified at compile time. See Table 16-3.)
GROUP	Defines indirect lists for TYPE statements.
MODE	Specifies format of typeout.
OPEN	Accesses program unit symbol table.
PAUSE	Sets FORDDT breakpoints.
PAUSE ON ERROR	Sets FORDDT breakpoints (for errors such as arithmetic overflows).
REMOVE	Clears FORDDT breakpoints.
REMOVE ON ERROR	Clears PAUSE ON ERROR breakpoints.
Other Commands	
LOCATE	Lists program unit names in which a given symbol is defined.
STRACE	Displays routine traceback of current program status.
WHAT	Displays current DIMENSION, GROUP, and FORDDT breakpoint information.

The FORDDT commands are described in detail in Section 17.5.

17.1 INPUT FORMAT

FORDDT commands consist of alphabetic FORTRAN-like identifiers and need consist of only those characters required to make the command unique. If you wish to specify parameters, a space is required following the command name. Comments may be appended to command lines by preceding the comment with an exclamation point (!).

17.1.1 Variables and Arrays

FORDDT allows you to access and modify the data in your program using standard FORTRAN symbolic names. Variables are specified simply by name. For example:

```
     name
```

where:

name is a variable name.

Array elements are specified in the following formats:

```
name
name (s1,...,sn)
```

where:

name is the name of the array

(s1,...,sn) are the subscripts of a particular array. The subscripts must be integer constants or variables.

You may reference an entire array simply by typing the array name (without subscripts). You may specify a range of array elements by typing the first and last element in the chosen range, separated by a dash (-).

The following examples show the various ways of specifying variables and arrays to FORDDT:

```
ALPHA
ALPHA(7)
ALPHA(PI)
ALPHA(2)-ALPHA(5)
```

17.1.2 Constant Conventions

FORDDT accepts optionally signed numeric data in the standard FORTRAN input formats:

1. INTEGER - A string of decimal digits.

2. REAL - A string of decimal digits optionally including a decimal point. Standard engineering and double-precision exponent formats are also accepted.

3. OCTAL - A string of octal digits optionally preceded by a double quote (").

4. COMPLEX - An ordered pair of integer or real constants separated by a comma and enclosed in parentheses.

5. LOGICAL - A Boolean argument, either .TRUE. or .FALSE.

6. CHARACTER - A string of printable ASCII characters enclosed by apostrophes.

7. HOLLERITH - A string of alphanumeric and/or special characters delimited by any alphanumeric or special character, excluding the space character, which does not occur with the string itself. Such as, # 12AB#, where # is the delimiting character.

17.1.3 Statement Labels and Source Line Numbers

FORTRAN statement labels are input and output by straightforward numeric reference, such as 1234. However, source line numbers must be input to FORDDT with a number sign (#) preceding them. This mandatory sign distinguishes statement labels from source line numbers.

```
PAUSE #3     !This causes a pause at source line number 3.

PAUSE 3      !This causes a pause at the statement labeled 3.
```

17.2 FORDDT AND THE FORTRAN /DEBUG SWITCH

Most facilities of FORDDT are available without the FORTRAN /DEBUG features. However, if you do not use the /DEBUG switch when compiling a FORTRAN program, the trace features (NEXT command) will not be available, and several of the other commands will be restricted.

Using the /DEBUG switch tells FORTRAN to compile extra information for debugging. (See Chapter 16 for more information.) These features are:

1. /DEBUG:DIMENSIONS, which generates dimension information in the .REL file for all arrays dimensioned in the subprogram. The dimension information is automatically available to FORDDT if you wish to reference an array in a TYPE or ACCEPT command. This feature eliminates the need to specify dimension information for FORDDT by using the DIMENSION command.

2. /DEBUG:LABELS, which generates labels for every executable source line in the form <line-number>L. If these labels are generated, they may be used as arguments with the FORDDT commands PAUSE and GOTO.

   This switch also generates labels at the last location allocated for a FORMAT statement so that FORDDT can detect the end of the statement. These labels have the form <format-label>F. If they are generated, you can display and modify FORMAT statements by means of the TYPE and ACCEPT commands.

   Note that the :LABELS switch is automatically activated with the :TRACE switch, since labels are needed to accomplish the trace features.

3. /DEBUG:TRACE, which generates a reference to FORDDT before each executable statement. This switch is required for the trace command NEXT to function.

   Note that if more than one FORTRAN statement is placed on a single input line, only the first statement has a FORDDT reference and line-number label associated with it. This also applies to the :LABELS switch.

4. /DEBUG:INDEX, which forces the compiler to store in its respective data location, as well as a register, the index variable of all DO loops at the beginning of each loop iteration. You will then be able to examine DO loops by using FORDDT. If you modify a DO loop index using FORDDT, it will not affect the number of loop iterations because a separate loop count is used.

In addition, this switch forces all function values to be stored in memory prior to return from the function. If this switch is specified, you can set a FORDDT pause on the RETURN statement (see Section 13.4.4) and then examine the value to be returned.

5. /DEBUG:BOUNDS, which generates the bounds checking code for all array references. Bounds violations produce run-time error messages. Note that the technique of specifying dimensions of 1 for subroutine arrays causes bounds check errors. (You can use this option without FORDDT.)

6. /DEBUG:ARGUMENTS, which performs type checking at load time for actual argument types and associated dummy argument types. Type violations produce non-fatal, load-time error messages. This switch also performs type checking at compile-time for statement functions.

17.3 LOADING AND STARTING FORDDT

1. On TOPS-10, the simplest method of debugging with FORDDT is:

```
.DEBUG filespec(DEBUG)
```

On TOPS-20, the corresponding command is:

```
@DEBUG filespec /DEBUG
```

On both systems, FORDDT responds with:

```
STARTING FORTRAN DDT

Program name:
```

When FORDDT prompts you for a program name, type the same name specified in the PROGRAM statement of the program being debugged. If the PROGRAM statement is not used in the program being debugged, FORDDT uses MAIN., and will not prompt for a program name.

FORDDT next prints its command prompt:

```
>>
```

The angle brackets indicate that FORDDT is ready to receive a command.

2. If you are on TOPS-20, you can type a question mark to the prompt to get a list of all FORDDT commands, as follows:

```
>>? One of the following:
ACCEPT  CHARACTER    CONTINUE  DDT         DIMENSION DOUBLE
GOTO    GROUP        HELP      LOCATE      MODE      NEXT
OPEN    PAUSE        REMOVE    START       STOP      STRACE
TYPE    WHAT
```

Also on TOPS-20, you can use the ESCape key for recognition of FORDDT commands. For example:

```
>>CON<ESC>TINUE
```

On both systems, you need only type the unique abbreviation of a specific FORDDT command.

3. You may wish to load your compiled program and FORDDT directly with the linking loader. (Loading with LINK is accomplished implicitly in the DEBUG command string.) The command sequence is as follows:

   On TOPS-10, to start LINK, type:

   ```
   .R LINK
   ```

   On TOPS-20, type:

   ```
   @LINK
   ```

   On both systems, when LINK prompts you with an asterisk, you can type a command string in any of the following forms:

   ```
   *filespec /DEB/G                         (loads DDT)

   *filespec /DEB:{FORDDT}/G                (loads FORDDT)
                  {FORTRA}

   *filespec /DEB:(DDT,{FORDDT})/G          (loads DDT
                       {FORTRA}             and FORDDT)

   *filespec /DEB:({FORDDT},DDT)/G          (loads FORDDT
                   {FORTRA}                 and DDT)
   ```

   In the last two command forms shown, the first debugging program specified (FORDDT or DDT) in the command string is the one you communicate with after the LINK command string is executed.

   See Section 17.9 for information on loading extended addressing programs.

17.4 SCOPE OF NAME AND LABEL REFERENCES

Each program unit has its own symbol table. When you initially enter FORDDT, you automatically open the symbol table of the main program. All references to names or labels through FORDDT must be made with respect to the currently open symbol table.

If you have given the main program a name other than MAIN. by using the PROGRAM statement (see Section 6.4.1), FORDDT asks for the defined program name. After you enter the program name, FORDDT opens the appropriate symbol table. At this point, symbol tables in programs other than the main program can be opened by using the OPEN command.

References to statement labels, line numbers, FORMAT statements, variables, and arrays must have labels that are defined in the currently open symbol table. However, FORDDT will accept variable and array references outside the currently open symbol table, providing the name is unique with respect to all program units in the given load module.

17.5 FORDDT COMMANDS

This section gives a detailed description of all commands in FORDDT. The commands are given in alphabetical order:

ACCEPT

Allows you to change the contents of a FORTRAN variable, array, array element, array element range, or FORMAT statement. The command format is:

```
ACCEPT name[/mode] value
```

where:

name — is the variable, array, array element, array element range, or FORMAT statement to be modified.

mode — is the format of the data value to be entered. The mode keyword must be preceded by a slash (/) and immediately follows the name. Intervening blanks are not allowed. (Note that /mode does not apply to FORMAT modification.)

value — is the new value to be assigned. The format of the input value must correspond to the specified mode.

DATA LOCATION MODIFICATION

Data Modes

The following data modes are accepted:

Mode	Meaning	Example
A	ASCII (left-justified)	/FOO/
C	CHARACTER	'ABC'
D	DOUBLE-PRECISION	123.4567890
F	REAL	123.45678
I	INTEGER	1234567890
O	OCTAL	7654321
L	LOGICAL	.TRUE. or .FALSE.
R	RASCII (right-justified)	\BAR\
S	SYMBOLIC	PSI(2,4)
X	COMPLEX	(1.25,-78.E+9)

If not specified, the default mode is REAL (F).

1. Two-Word Values

   For the data modes ASCII (A), OCTAL (O), RASCII (R), and SYMBOLIC (S), FORDDT will accept a "/BIG" modifier on the mode switch. This modifier indicates that the variable and the value are to be interpreted as two words long.

Example:

```
ACCEPT VAR/RASCII/BIG '1234567890'
```

assumes that VAR is two words long and stores the given 10-character literal into it.

The /BIG modifier can also be used to display more than the first 256 characters of long character strings.

2. Character Variables

   A character variable can be initialized by using an ACCEPT command of the following form:

   ```
   ACCEPT VAR/C 'string'
   ```

   Note that length fo the variable is that which is specified in the source program. If the string is longer than the variable, the rightmost characters are truncated. If the string is shorter than the variable, it is stored left-justified and padded on the right with blanks.

3. Initialization of Arrays

   If the name field of an ACCEPT contains an unsubscripted array name or a range of array elements, all elements of the array or the specified range are set to the given value.

   Example:

   ```
   ACCEPT ARRAY/F 1.0

   or

   ACCEPT ARRAY(5)-ARRAY(10)/F 1.0
   ```

   Note that this applies only to modes other than ASCII and RASCII.

   For character arrays, if the value has fewer characters than the length of the array element, the rightmost character positions of the element are initialized with spaces. If the value has more characters than the length of the array element, the value is truncated to the right.

4. Long Literals

   When the value field of an ACCEPT contains an unsubscripted array name or range of array elements, and the specified data mode is ASCII, the value field is expected to contain a long literal string. ACCEPT stores the string linearly into the array or array range. If the array is not filled, the remainder of the array or range is filled with zeroes. If the literal is too long, the remaining characters are ignored.

Example:

```
     ACCEPT ARRAY/ASCII
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

5. FORMAT Statement Modification

   When the name field of an ACCEPT contains a label, FORDDT expects this label to be a FORMAT statement label and that the value field contains a new format specification.

   Example:

   ```
   ACCEPT 10 (1H0,F10.2,3(I2))
   ```

   The new specification cannot be longer than the space originally allocated to the FORMAT by the compiler. The remainder of the area is cleared if the new specification is shorter.

   Note that FOROTS performs some encoding of FORMAT statements when it processes them for the first time. If any I/O statement referencing the given FORMAT has been executed, the FORTRAN program has to be restarted (re-initializing FOROTS).

CHARACTER

Defines the dimensions of a character array. The result of this command is the same as for the DIMENSION command except that the array so dimensioned is understood by FORDDT to be a character array. The command format is:

```
CHARACTER arrayname ([L1:]U1[,[L2:]U2,...])
```

NOTE

> This declarator cannot be used to specify element length. The length specified in the user program will be used.

CONTINUE

Allows the program to resume execution after a FORDDT pause. After a CONTINUE is executed, the program either runs to completion or until another pause is encountered. The command format is:

```
CONTINUE [n]
```

where the n is optional and, if omitted, is assumed to be one. If a value is provided, it can be a numeric constant or program variable, but it is treated as an integer. When the value n is specified, the program continues execution until the nth occurrence of this pause. For example,

```
CONTINUE 20
```

continues execution until the 20th occurrence of the pause, or until a different pause is encountered.

DDT

Transfers control of the program to DDT, the standard system debugging program. Any files currently opened by FOROTS are unaffected, and a return to FORDDT is possible so that program execution may be resumed.

%FDDT is the global symbol used to return control to FORDDT. The command format is:

```
%FDDT<ESC>G
```

Your program will be in the same condition as before unless you have modified your core image with DDT.

DIMENSION

Sets, displays, or removes the user-defined dimensions of an array for FORDDT access purposes. These dimensions need not agree with those declared to the compiler in the source code. FORDDT allows you to redimension an array to have a larger scope than that of the source program. If this is done, a warning is given.

NOTE

> The DIMENSION command cannot be used to declare double-precision, complex, or character arrays (see the CHARACTER and DOUBLE commands).

The command format is:

```
DIMENSION name ([L1:]U1[,[L2:]U2,...])
```

where:

name	is the name of the array
([L1:]U1...)	specifies the bounds of the array, where L is the lower bound and U is the upper bound. The default value of L is 1. The bounds must be integer constants or variables.

For example:

```
DIMENSION ALPHA(7,5:6,10)
```

FORDDT remembers the dimensions of the array until they are redefined or removed.

The command:

```
DIMENSION
```

gives a full list of all the user-defined dimensions for all arrays.

```
DIMENSION ALPHA
```

displays the current information for the array ALPHA only.

```
DIMENSION ALPHA/REMOVE
```

removes any user-defined array information for the array ALPHA.

DOUBLE

Defines the dimensions of a double-precision or complex array. The result of this command is the same as for the DIMENSION command except that the array so dimensioned is understood by FORDDT to be an array with two-word elements.

The command format is:

```
DOUBLE arrayname ([L1:]U1[,[L2:]U2,...])
```

GOTO

Allows you to continue your program from a point other than the one at which it last paused. The GOTO allows you to continue at a statement label or source line number provided that the /DEBUG:LABELS switch has been used or the contents of a symbol previously ASSIGNed during the program execution has been used.

Note that the program must be STARTed before this command can be used, and also note that a GOTO is not allowed after the <CTRL/C>, REENTER sequence (see Section 17.6).

The command format is:

```
GOTO n
```

GROUP

Sets up a string of text for input to a TYPE command. You can store TYPE statements as a list of variables identified by the numbers 1 through 8. This feature eliminates the need to retype the same list of variables each time you wish to examine the same group. Refer to the TYPE command for the proper format of the list.

The command format is:

```
GROUP [n list]
```

where:

n — is the group number 1-8

list — is a list of group numbers preceded by a slash (/) and/or variable names to be typed when you give the command: TYPE /n, where n is the group number. The validity of the list is not checked.

```
GROUP
```

with no arguments causes FORDDT to type out the current contents of all the groups.

```
GROUP n
```

types out the contents of the particular group requested.

Note that one group may refer to another.

For example:

```
GROUP 2 VAR2,VAR3
GROUP 3 /1,/2
```

LOCATE

Lists the program unit names in which a given symbol is defined. This is useful when the variable you wish to locate is not in the currently open program unit and is defined in more than one program unit. The command format is:

```
LOCATE n
```

where n may be any FORTRAN variable, array, label, line number, or FORMAT statement number.

MODE

Defines the display format for succeeding FORDDT TYPE commands. You need type only the first character of the mode to identify it to FORDDT because all characters after the first are ignored. The modes are:

Mode	Meaning
C	CHARACTER
F	REAL
D	DOUBLE-PRECISION
X	COMPLEX
I	INTEGER
O	OCTAL
A	ASCII (left-justified)
R	RASCII (right-justified)
L	LOGICAL

Unless the MODE command is given, the default typeout mode is the REAL (F) format.

The command format is:

```
MODE list
```

where list contains one or more of the mode identifiers separated by commas. The current setting can be changed by issuing another MODE command. If more than one mode is given, the values are typed out in the order: C, F, D, X, I, O, A, R, L.

A typical command string might be:

```
MODE A,I,OCTAL
```

NEXT

Allows you to cause FORDDT to trace source lines, statement labels, and entry point names during execution of your program. This command only provides trace facilities if the program is compiled with the FORTRAN /DEBUG switch. If this switch is not used, the NEXT command acts as a CONTINUE command. The command format is:

```
NEXT [n][/sw]
```

where:

n is a program variable or integer numeric value

sw is one of the following switches

/S= statement label
/L= source line
/E= entry point

The default starting value of n is 1, a single statement trace. The default switch is /L.

The command

```
NEXT 20/L
```

traces the execution of the next 20 source line numbers or until another pause is encountered.

Note that if no argument is specified, the last argument given is used.

For example:

```
NEXT /E
```

changes the tracing mode to trace only subprogram entries using the numeric argument previously supplied.

OPEN

Allows you to open a particular program unit of the loaded program so that the variables are accessible to FORDDT. Any previously opened program unit is closed automatically when a new one is opened. Only global symbols, symbols in the currently open unit, and unique locals are available at any one time. Note that starting FORDDT automatically opens the main program.

The command format is:

```
OPEN name
```

where name is the subprogram name. OPEN with no arguments reopens the main program.

PAUSE

Allows you to place a FORDDT breakpoint at a statement number, source line number, or subroutine entry point. Up to ten breakpoints may be set at any one time. When a breakpoint is encountered, execution is suspended at that point and control is returned to FORDDT. The symbol table of that subprogram is also automatically opened.

The command formats are:

```
PAUSE
PAUSE p
PAUSE p AFTER n
PAUSE p IF condition
PAUSE p TYPING /g
PAUSE p AFTER n TYPING /g
PAUSE p IF condition TYPING /g
```

where:

P is the point where the breakpoint is inserted

n is an integer constant, variable, or array element

g is a group number

```
PAUSE 100
```

sets a breakpoint at statement label 100, causes execution to be suspended, and causes FORDDT to be entered on reaching 100 in the program.

```
PAUSE #245 AFTER MAX(5)
```

causes a break to occur at source line number 245 after encountering this point the number of times specified by MAX(5). Note that AFTER can not be abbreviated.

```
PAUSE DELTA IF LIMIT(3,1).GT.2.5E-3
```

causes a FORDDT break to occur if the variable LIMIT(3,1) is greater than the value 2.5E-3. The IF can not be abbreviated, and the following FORTRAN logical connectives are allowed:

```
.GT., .GE., .LT., .LE., .EQ., .NE.
```

Double-precision comparisons and arithmetic operations are not allowed. However, comparisons can be made between variables, constants, and logical constants (such as .TRUE. and .FALSE.).

```
PAUSE 505 TYPING /5
```

sets a FORDDT breakpoint at label 505, and the variables in group 5 are displayed. The TYPING specification can not be abbreviated.

```
PAUSE #24 AFTER 16 TYPING /3
```

causes a break at source line number 24 after 16 times through; however, the contents of group 3 are displayed every time.

When the TYPING option is used with the PAUSE command, control can be transferred to FORDDT at the next typeout by typing any character on the terminal.

Note that pause requests remain after a <CTRL/C> REENTER sequence, a START command, or a <CTRL/C> START sequence.

PAUSE ON ERROR Causes the program to enter FORDDT whenever an error occurs (such as an arithmetic overflow). It has the same command format as the PAUSE command.

REMOVE Removes the previously set FORDDT breakpoints. The command format is:

```
REMOVE [p]
```

For example,

```
REMOVE L#123
```

removes a breakpoint from the program source line number 123.

```
REMOVE ALPHA
```

removes a breakpoint from the subroutine entry to ALPHA.

REMOVE with no arguments removes all your FORDDT breakpoints, and, in this case, no abbreviation of REMOVE is allowed.

REMOVE ON ERROR Removes a PAUSE ON ERROR breakpoint. It has the same command format as the REMOVE command.

START Starts your program at the normal FORTRAN main program entry point. The command format is:

```
START
```

STOP Terminates the program, closes all files opened by FOROTS, and causes an exit to the monitor. The usual command format is:

```
STOP

STOP /RETURN
```

allows a return to monitor mode without releasing devices or closing files so that a CONTINUE can be issued.

STRACE Displays a subprogram level traceback of the current state of the program. The command format is:

```
STRACE
```

TYPE Displays FORTRAN defined variables, arrays, or array elements on your terminal. The command format is:

```
TYPE list
```

where list may be one or more variables or array references and/or group numbers. These specifications must be separated by commas, and group numbers must be preceded by a slash (/). The command with no arguments uses the last argument list submitted to FORDDT.

An array element range can also be specified. For example:

```
TYPE PI(5)-PI(13)
```

displays the values from PI(5) to PI(13) inclusive. If an unsubscripted array name is specified, the entire array is typed.

There are several methods of choosing the form of typeout in conjunction with the MODE command:

1. If you do not specify a format, the default is real.

2. You can specify a format through the MODE command described in this chapter.

3. You can change the format(s) previously designated by the MODE command by including print modifiers in the TYPE or GROUP string. The print modifiers are:

A	ASCII(left-justified)
B	LONG
C	CHARACTER
D	DOUBLE-PRECISION
F	REAL
I	INTEGER
L	LOGICAL
O	OCTAL
R	RASCII(right-justified)
X	COMPLEX

4. If you type a variable in mode CHARACTER (C), the number of characters printed is equal to the length declared in the FORTRAN source program, up to a maximum of 256 characters. The /B switch can be used to override the 256 character maximum.

   The B switch may be used in conjunction with the A, O, and R switches. This modifier indicates that the variable is to be interpreted as two words long. The B switch can also be used with the C switch to display more than the first 256 characters of long character strings. The B switch can not be used alone.

   The first print modifier specified in a string of variables determines the mode for the entire string unless another mode is placed directly to the right of a particular variable. For example, in:

   ```
   TYPE /I K,L/O,M,N/A,/2
   ```

   the typeout mode is integer until another mode is specified. Therefore,

   K, M are integer - the default mode for group 2 is integer
   L is OCTAL
   N is ASCII

WHAT Displays on your terminal the name of the currently open program unit, any currently active breakpoints, any group specifications, and any user-set array dimensions. The command format is:

```
WHAT
```

17.6 ENVIRONMENT CONTROL

If a program enters an infinite loop, you can recover by typing a <CTRL/C>(twice) REENTER sequence. This action causes FORDDT to simulate a pause at the point of reentry and allows you to control your run-away program.

Most commands can be used once the program has been reentered; however, GOTO, STRACE, TYPE, and ACCEPT cause transfer of control to routines external to FORDDT. No guarantee can be made to ensure that any of these commands following a <CTRL/C> REENTER sequence will not destroy the program integrity. The program must be returned to a stable state before any of these four commands can be issued. In order to restore program integrity, you should set a pause at the next label and then CONTINUE to it. If the /DEBUG:TRACE switch is used, a NEXT 1 command can be issued to restore program integrity.

17.7 FORTRAN /OPTIMIZE SWITCH

You should never attempt to use FORDDT with a program that has been compiled with the /OPTIMIZE switch. The global optimizer causes variables to be kept in ACs. For this reason, attempts to examine or modify variables in optimized programs will not work.

17.8 CALLING FORDDT

FORDDT can be called directly from a user FORTRAN program. The appropriate statement is:

```
     CALL FORDDT
```

where no argument is required. FORDDT must be loaded and initialized before a CALL to FORDDT is made. This is done by starting the program in debug mode prior to the first call (see Section 17.3, item 1). All FORDDT commands are allowed. A CONTINUE will resume normal execution of the user program (similar to a RETURN from a subroutine).

NOTE

> Since FORDDT is defined as a global symbol, users should be careful if they decide to use FORDDT as a program, subroutine, or function name.

17.9 FORDDT AND FORTRAN-20 EXTENDED ADDRESSING

FORDDT V10 has been modified to be able to run in any section and access data and code in all sections. The user interface to FORDDT is the same regardless of whether or not a program uses extended addressing.

FORDDT V10 is section independent. The same FORDDT.REL will work in either section 0 or a non-zero section.

If a program is loaded with the /DEBUG:FORDDT option, LINK loads FORDDT.REL with the program. FORDDT.REL is a single-segment module (it has only low segment code); therefore, when loaded with a FORTRAN object program that was compiled with the /EXTEND switch (see Section 16.5), FORDDT, by default, is redirected to the .DATA. psect.

FORDDT Version 10 will not be guaranteed to work with previous versions of FORTRAN-10/20.

FORDDT and FORLIB must be in the same section. Since they would by default go into the .DATA./.CODE. section, the user normally would not need to be concerned about this. However, you should be cautious when you use the LINK /REDIRECT switch.

NOTE

> The first page of any section that contains code is reserved for FORDDT and FOROTS.

CHAPTER 18

USING THE FORTRAN OBJECT TIME SYSTEM (FOROTS)

This chapter describes the facilities that the FORTRAN Object Time System (FOROTS) provides for the FORTRAN user. FOROTS implements all standard FORTRAN I/O operations as set forth in the FORTRAN-77 standard In addition it provides the user with capabilities and programming features beyond those defined in the ANSI standard.

The primary function of FOROTS is to act as a direct interface between user-object programs and the TOPS-10 or TOPS-20 monitor during input and output operations. Other capabilities include:

1. Job initialization

2. Channel and memory management

3. Error handling and reporting

4. File management

5. Formatting of data

6. Mathematical library

7. User library (nonmathematical)

8. Specialized applications packages

9. Overlay facilities

FOROTS runs on any TOPS-10 or TOPS-20 system. FOROTS interfaces with all TOPS-10 or TOPS-20 peripheral devices.

18.1 FEATURES OF FOROTS

The following list briefly describes many specific features of FOROTS; more detailed information concerning the implementation of these features is given later in this chapter.

1. Your program can run in either batch or timesharing mode without requiring a program change. All differences between batch-mode and timesharing-mode operations are resolved by FOROTS.

2. Your programs can access both directory and nondirectory devices in the same manner.

3. FOROTS helps provide complete data file compatibility between all system devices.

4. FOROTS treats devices located at remote stations in the same way it treats local devices.

5. Programs written for magnetic tape operations will run correctly on disk under FOROTS supervision. FOROTS simulates the commands needed for magnetic tape operations.

6. You may change or specify object program device and file specifications with a FOROTS interactive dialogue.

7. Non-FORTRAN binary data files may be read in IMAGE mode by FOROTS.

8. FOROTS provides interactive program/operating system error-processing routines. These routines permit you to route the execution of the program to specific error-processing routines whenever designated types of errors are detected.

9. An error traceback facility for fatal errors provides the active execution path (by subroutine calls) between the main program and the subroutine where the fatal error occurred.

10. FOROTS provides a trap-handling system for arithmetic functions, including default values and error reports.

11. FOROTS permits your program to switch from READ to WRITE on the same I/O device without loss of data or buffering.

12. Although primarily designed for use with the FORTRAN-10/20 object programs, you can also use FOROTS as an independent I/O system, and as an I/O system for MACRO object programs.

18.2 ERROR PROCESSING

Whenever a run-time error is detected, the FOROTS error-processing system takes control of program execution. This system determines the class of the error and either outputs an appropriate message at the controlling terminal or branches the program to a predesignated processing routine.

18.3 INPUT/OUTPUT FACILITIES

On TOPS-10, FOROTS uses monitor-buffered I/O for SEQUENTIAL, SEQINOUT, SEQIN, and SEQOUT files access, and uses dump mode I/O for DIRECT(RANDOM), RANDIN, and dump mode files access.

On TOPS-20, FOROTS uses PMAP monitor calls for disk files access other than APPEND, and uses monitor-buffered I/O for all other file accesses.

The following sections describe I/O data channel and access modes.

18.3.1 Input/Output Channels Used By FOROTS (TOPS-10 Only)

FOROTS uses extended channels starting at channel 20 for I/O operations. User programs can request I/O channels 0 through 17 through the ALCHN. and FUNCT. routines.

When a request is made for an I/O channel, a table is scanned until a free channel is found. The first free channel is assigned to the requesting program. On completion of the assigned transfer, control of the I/O channel is returned to FOROTS by using the DECHN. routine.

18.3.2 File Access

Data can be transferred between processor storage and peripheral devices using either sequential or direct (random) access.

18.3.3 Closing Files After Non-standard Termination

When a FORTRAN program is aborted by <CTRL/C> or an error, open files cannot be closed with the monitor command CLOSE. The following command should be used:

```
     REENTER
```

FOROTS then asks if you want the files to be closed. If you answer YES, then, the files are closed.

18.3.3.1 Sequential Access - In a sequential-access transfer operation, the records involved are transferred in the same order as they appear in the source file. Each I/O statement executed transfers the record immediately following the last record transferred from the accessed source file.

A type of the sequential access is available for output (write) operations. This type of access is called APPEND and is specified by the OPEN statement specifier ACCESS='APPEND' (see Section 11.3.1). APPEND lets you write a record immediately after the last logical record of the accessed file. During APPEND transfer, the records already in the accessed file remain unchanged; the only function performed is the appending of the transferred records to the end of the file.

You must specify transfer types (other than SEQINOUT) by setting the ACCESS option of a FORTRAN OPEN statement to one of several possible arguments. For the sequential access, the arguments are:

ACCESS='SEQIN'	(file is opened for read-only access)
ACCESS='SEQOUT'	(file is opened for output)
ACCESS='SEQUENTIAL'	(file is opened for input or output)
ACCESS='SEQINOUT'	(same as SEQUENTIAL)
ACCESS='APPEND'	(sequential append access)

NOTE

> A common way to append data to a file opened with SEQUENTIAL access is to read past the end of file, and then begin writing. The FORTRAN-77 standard requires that a BACKSPACE operation be done to back over the 'end file record' preceding the WRITE.

18.3.3.2 Direct (Random) Access Mode - Direct access permits records to be accessed and transferred from a source file in any desired order. Direct access can only be used with disk files that have been set up for direct access. Direct-access files must contain a specified number of identically sized records that may be individually accessed by a record number.

Direct-access transfers may be done in either a read/write direction or a special read-only direction. You must specify random transfer direction by setting the ACCESS option of an OPEN statement (see Section 11.3.1) to one of several possible arguments.

```
ACCESS='DIRECT' (direct read/write access)
ACCESS='RANDOM' (same as DIRECT)
ACCESS='RANDIN' (direct read-only access)
```

18.4 ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS

The following sections describe the types of data files that are acceptable to FOROTS.

18.4.1 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters. A vertical paper-motion character (that is, a form feed, a vertical tab, or a line feed) terminates each set. Logical records may be split across physical blocks. There is no maximum length for logical records.

18.4.2 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data that the executing program can reference with either a READ or a WRITE statement. A logical record is preceded and ended by a control word and may have one or more control words embedded within it. In FORTRAN binary data files, there is no relationship between logical records and physical device block sizes. There is no implied maximum length for logical records.

18.4.2.1 Format of Binary Files - A FOROTS binary file can contain three forms of Logical Segment Control Words (LSCW). These LSCWs give FOROTS the ability to distinguish ASCII files from binary files. The value in the high-order 9 bits of an LSCW tells what kind of LSCW it is: START, CONTINUE, or END.

	LSCW	
START	001+	the number of words in the segment including the START LSCW word (exclusive of the END LSCW)
CONTINUE	002+	the number of words in the segment including the CONTINUE LSCW
END	003+	the number of words in the whole record including all LSCWs

If the access you specify for a file through the OPEN statement ACCESS= argument is 'SEQIN', 'SEQOUT', 'SEQUENTIAL', or 'SEQINOUT', all three LSCWs can appear in a record. If you specify a record size, all records are of the same length, and there are no CONTINUE LSCWs.

The following examples illustrate the LSCW. The direct-access binary file contains only 001 and 003 LSCWs.

```
C     LOOK AT A BINARY FILE AND SEE THE LOGICAL SEGMENT
C     CONTROL WORDS.

      OPEN(UNIT=1,ACCESS='DIRECT',MODE='BINARY',
      1     RECORDSIZE=100)

      I=5
      WRITE(1'1) (I, J=1,10)

      J=7
      WRITE(1'2) (J,K=1,10)
      END
```

```
0/      001000,000145◄—Number of    100/    000000,000000
1/      000000,000005  words in     101/    000000,000000
2/      000000,000005  record       102/    000000,000000
3/      000000,000005  counting     103/    000000,000000
4/      000000,000005  START LSCW   104/    000000,000000
5/      000000,000005               105/    000000,000000
6/      000000,000005               106/    000000,000000
7/      000000,000005               107/    000000,000000
10/     000000,000005               110/    000000,000000
11/     000000,000005               111/    000000,000000
12/     000000,000005               112/    000000,000000
13/     000000,000000               113/    000000,000000
14/     000000,000000               114/    000000,000000
15/     000000,000000               115/    000000,000000
16/     000000,000000               116/    000000,000000
17/     000000,000000               117/    000000,000000
20/     000000,000000               120/    000000,000000
21/     000000,000000               121/    000000,000000
22/     000000,000000               122/    000000,000000
23/     000000,000000               123/    000000,000000
24/     000000,000000               124/    000000,000000
25/     000000,000000               125/    000000,000000
26/     000000,000000               126/    000000,000000
27/     000000,000000               127/    000000,000000
30/     000000,000000               130/    000000,000000
31/     000000,000000               131/    000000,000000
32/     000000,000000               132/    000000,000000
33/     000000,000000               133/    000000,000000
34/     000000,000000               134/    000000,000000
35/     000000,000000               135/    000000,000000
36/     000000,000000               136/    000000,000000
37/     000000,000000               137/    000000,000000
40/     000000,000000               140/    000000,000000
41/     000000,000000               141/    000000,000000
42/     000000,000000               142/    000000,000000
43/     000000,000000               143/    000000,000000
44/     000000,000000               144/    000000,000000
45/     000000,000000               145/    003000,000146◄—END LSCW
46/     000000,000000               146/    001000,000145  containing
47/     000000,000000               147/    000000,000007  the number
50/     000000,000000               150/    000000,000007  of words in
51/     000000,000000               151/    000000,000007  the record
52/     000000,000000               152/    000000,000007  including
53/     000000,000000               153/    000000,000007  all LSCWs.
54/     000000,000000               154/    000000,000007
55/     000000,000000               155/    000000,000007
56/     000000,000000               156/    000000,000007
57/     000000,000000               157/    000000,000007
60/     000000,000000               160/    000000,000007
61/     000000,000000               161/    000000,000000
62/     000000,000000               162/    000000,000000
63/     000000,000000               163/    000000,000000
64/     000000,000000               164/    000000,000000
65/     000000,000000               165/    000000,000000
66/     000000,000000               166/    000000,000000
67/     000000,000000               167/    000000,000000
70/     000000,000000               170/    000000,000000
71/     000000,000000               171/    000000,000000
72/     000000,000000               172/    000000,000000
73/     000000,000000               173/    000000,000000
74/     000000,000000               174/    000000,000000
75/     000000,000000               175/    000000,000000
76/     000000,000000               176/    000000,000000
77/     000000,000000               177/    000000,000000
```

```
200/   000000,000000                246/   000000,000000
201/   000000,000000                247/   000000,000000
202/   000000,000000                250/   000000,000000
203/   000000,000000                251/   000000,000000
204/   000000,000000                252/   000000,000000
205/   000000,000000                253/   000000,000000
206/   000000,000000                254/   000000,000000
207/   000000,000000                255/   000000,000000
210/   000000,000000                256/   000000,000000
211/   000000,000000                257/   000000,000000
212/   000000,000000                260/   000000,000000
213/   000000,000000                261/   000000,000000
214/   000000,000000                262/   000000,000000
215/   000000,000000                263/   000000,000000
216/   000000,000000                264/   000000,000000
217/   000000,000000                265/   000000,000000
220/   000000,000000                266/   000000,000000
221/   000000,000000                267/   000000,000000
222/   000000,000000                270/   000000,000000
223/   000000,000000                271/   000000,000000
224/   000000,000000                272/   000000,000000
225/   000000,000000                273/   000000,000000
226/   000000,000000                274/   000000,000000
227/   000000,000000                275/   000000,000000
230/   000000,000000                276/   000000,000000
231/   000000,000000                277/   000000,000000
232/   000000,000000                300/   000000,000000
233/   000000,000000                301/   000000,000000
234/   000000,000000                302/   000000,000000
235/   000000,000000                303/   000000,000000
236/   000000,000000                304/   000000,000000
237/   000000,000000                305/   000000,000000
240/   000000,000000                306/   000000,000000
241/   000000,000000                307/   000000,000000
242/   000000,000000                310/   000000,000000
243/   000000,000000                311/   000000,000000
244/   000000,000000                312/   000000,000000
245/   000000,000000                313/   003000,000146
```

On TOPS-10, in the sequential-access binary file, the second record crosses the disk block boundary and contains an 002 (CONTINUE) LSCW.

On TOPS-20, the CONTINUE LSCW occurs on buffer boundaries, whose size is determined by the BUFFERCOUNT keyword in the OPEN statement (see Section 11.3.5) (default is four pages, 4000 octal words).

```
C     LOOK AT A BINARY FILE AND SEE THE LOGICAL SEGMENT
C     CONTROL WORDS.

      OPEN(UNIT=1,MODE='BINARY')

      I=5
      WRITE(1) (I, J=1,100)

      J=7
      WRITE(1) (J,K=1,100)
      END
```

```
0/      001000,000145           100/    000000,000005
1/      000000,000005           101/    000000,000005
2/      000000,000005           102/    000000,000005
3/      000000,000005           103/    000000,000005
4/      000000,000005           104/    000000,000005
5/      000000,000005           105/    000000,000005
6/      000000,000005           106/    000000,000005
7/      000000,000005           107/    000000,000005
10/     000000,000005           110/    000000,000005
11/     000000,000005           111/    000000,000005
12/     000000,000005           112/    000000,000005
13/     000000,000005           113/    000000,000005
14/     000000,000005           114/    000000,000005
15/     000000,000005           115/    000000,000005
16/     000000,000005           116/    000000,000005
17/     000000,000005           117/    000000,000005
20/     000000,000005           120/    000000,000005
21/     000000,000005           121/    000000,000005
22/     000000,000005           122/    000000,000005
23/     000000,000005           123/    000000,000005
24/     000000,000005           124/    000000,000005
25/     000000,000005           125/    000000,000005
26/     000000,000005           126/    000000,000005
27/     000000,000005           127/    000000,000005
30/     000000,000005           130/    000000,000005
31/     000000,000005           131/    000000,000005
32/     000000,000005           132/    000000,000005
33/     000000,000005           133/    000000,000005
34/     000000,000005           134/    000000,000005
35/     000000,000005           135/    000000,000005
36/     000000,000005           136/    000000,000005
37/     000000,000005           137/    000000,000005
40/     000000,000005           140/    000000,000005
41/     000000,000005           141/    000000,000005
42/     000000,000005           142/    000000,000005
43/     000000,000005           143/    000000,000005
44/     000000,000005           144/    000000,000005
45/     000000,000005           145/    003000,000146
46/     000000,000005           146/    001000,000032<----Number of
47/     000000,000005           147/    000000,000007     words to
50/     000000,000005           150/    000000,000007     next LSCW
51/     000000,000005           151/    000000,000007
52/     000000,000005           152/    000000,000007
53/     000000,000005           153/    000000,000007
54/     000000,000005           154/    000000,000007
55/     000000,000005           155/    000000,000007
56/     000000,000005           156/    000000,000007
57/     000000,000005           157/    000000,000007
60/     000000,000005           160/    000000,000007
61/     000000,000005           161/    000000,000007
62/     000000,000005           162/    000000,000007
63/     000000,000005           163/    000000,000007
64/     000000,000005           164/    000000,000007
65/     000000,000005           165/    000000,000007
66/     000000,000005           166/    000000,000007
67/     000000,000005           167/    000000,000007
70/     000000,000005           170/    000000,000007
71/     000000,000005           171/    000000,000007
72/     000000,000005           172/    000000,000007
73/     000000,000005           173/    000000,000007
74/     000000,000005           174/    000000,000007
75/     000000,000005           175/    000000,000007
76/     000000,000005           176/    000000,000007
77/     000000,000005           177/    000000,000007
```

```
200/    002000,000114<----Continue  246/    000000,000007
201/    000000,000007     LSCW      247/    000000,000007
202/    000000,000007               250/    000000,000007
203/    000000,000007               251/    000000,000007
204/    000000,000007               252/    000000,000007
205/    000000,000007               253/    000000,000007
206/    000000,000007               254/    000000,000007
207/    000000,000007               255/    000000,000007
210/    000000,000007               256/    000000,000007
211/    000000,000007               257/    000000,000007
212/    000000,000007               260/    000000,000007
213/    000000,000007               261/    000000,000007
214/    000000,000007               262/    000000,000007
215/    000000,000007               263/    000000,000007
216/    000000,000007               264/    000000,000007
217/    000000,000007               265/    000000,000007
220/    000000,000007               266/    000000,000007
221/    000000,000007               267/    000000,000007
222/    000000,000007               270/    000000,000007
223/    000000,000007               271/    000000,000007
224/    000000,000007               272/    000000,000007
225/    000000,000007               273/    000000,000007
226/    000000,000007               274/    000000,000007
227/    000000,000007               275/    000000,000007
230/    000000,000007               276/    000000,000007
231/    000000,000007               277/    000000,000007
232/    000000,000007               300/    000000,000007
233/    000000,000007               301/    000000,000007
234/    000000,000007               302/    000000,000007
235/    000000,000007               303/    000000,000007
236/    000000,000007               304/    000000,000007
237/    000000,000007               305/    000000,000007
240/    000000,000007               306/    000000,000007
241/    000000,000007               307/    000000,000007
242/    000000,000007               310/    000000,000007
243/    000000,000007               311/    000000,000007
244/    000000,000007               312/    000000,000007
245/    000000,000007               313/    000000,000007
                                    313/    003000,000147
```

Image files contain no LSCWs. You can only backspace an IMAGE file that is created with a record size.

```
C     LOOK AT AN IMAGE MODE FILE AND SEE NO LOGICAL SEGMENT
C     CONTROL WORDS.

      OPEN(UNIT=1,MODE='IMAGE')

      I=5
      WRITE(1) (I, J=1,100)

      J=7
      WRITE(1) (J,K=1,100)
      END
```

```
0/      000000,000005               100/    000000,000005
1/      000000,000005               101/    000000,000005
2/      000000,000005               102/    000000,000005
3/      000000,000005               103/    000000,000005
4/      000000,000005               104/    000000,000005
5/      000000,000005               105/    000000,000005
6/      000000,000005               106/    000000,000005
7/      000000,000005               107/    000000,000005
10/     000000,000005               110/    000000,000005
11/     000000,000005               111/    000000,000005
12/     000000,000005               112/    000000,000005
13/     000000,000005               113/    000000,000005
14/     000000,000005               114/    000000,000005
15/     000000,000005               115/    000000,000005
16/     000000,000005               116/    000000,000005
17/     000000,000005               117/    000000,000005
20/     000000,000005               120/    000000,000005
21/     000000,000005               121/    000000,000005
22/     000000,000005               122/    000000,000005
23/     000000,000005               123/    000000,000005
24/     000000,000005               124/    000000,000005
25/     000000,000005               125/    000000,000005
26/     000000,000005               126/    000000,000005
27/     000000,000005               127/    000000,000005
30/     000000,000005               130/    000000,000005
31/     000000,000005               131/    000000,000005
32/     000000,000005               132/    000000,000005
33/     000000,000005               133/    000000,000005
34/     000000,000005               134/    000000,000005
35/     000000,000005               135/    000000,000005
36/     000000,000005               136/    000000,000005
37/     000000,000005               137/    000000,000005
40/     000000,000005               140/    000000,000005
41/     000000,000005               141/    000000,000005
42/     000000,000005               142/    000000,000005
43/     000000,000005               143/    000000,000005
44/     000000,000005               144/    000000,000007
45/     000000,000005               145/    000000,000007
46/     000000,000005               146/    000000,000007
47/     000000,000005               147/    000000,000007
50/     000000,000005               150/    000000,000007
51/     000000,000005               151/    000000,000007
52/     000000,000005               152/    000000,000007
53/     000000,000005               153/    000000,000007
54/     000000,000005               154/    000000,000007
55/     000000,000005               155/    000000,000007
56/     000000,000005               156/    000000,000007
57/     000000,000005               157/    000000,000007
60/     000000,000005               160/    000000,000007
61/     000000,000005               161/    000000,000007
62/     000000,000005               162/    000000,000007
63/     000000,000005               163/    000000,000007
64/     000000,000005               164/    000000,000007
65/     000000,000005               165/    000000,000007
66/     000000,000005               166/    000000,000007
67/     000000,000005               167/    000000,000007
70/     000000,000005               170/    000000,000007
71/     000000,000005               171/    000000,000007
72/     000000,000005               172/    000000,000007
73/     000000,000005               173/    000000,000007
74/     000000,000005               174/    000000,000007
75/     000000,000005               175/    000000,000007
76/     000000,000005               176/    000000,000007
77/     000000,000005               177/    000000,000007
```

```
200/    000000,000007               244/    000000,000007
201/    000000,000007               245/    000000,000007
202/    000000,000007               246/    000000,000007
203/    000000,000007               247/    000000,000007
204/    000000,000007               250/    000000,000007
205/    000000,000007               251/    000000,000007
206/    000000,000007               252/    000000,000007
207/    000000,000007               253/    000000,000007
210/    000000,000007               254/    000000,000007
211/    000000,000007               255/    000000,000007
212/    000000,000007               256/    000000,000007
213/    000000,000007               257/    000000,000007
214/    000000,000007               260/    000000,000007
215/    000000,000007               261/    000000,000007
216/    000000,000007               262/    000000,000007
217/    000000,000007               263/    000000,000007
220/    000000,000007               264/    000000,000007
221/    000000,000007               265/    000000,000007
222/    000000,000007               266/    000000,000007
223/    000000,000007               267/    000000,000007
224/    000000,000007               270/    000000,000007
225/    000000,000007               271/    000000,000007
226/    000000,000007               272/    000000,000007
227/    000000,000007               273/    000000,000007
230/    000000,000007               274/    000000,000007
231/    000000,000007               275/    000000,000007
232/    000000,000007               276/    000000,000007
233/    000000,000007               277/    000000,000007
234/    000000,000007               300/    000000,000007
235/    000000,000007               301/    000000,000007
236/    000000,000007               302/    000000,000007
237/    000000,000007               303/    000000,000007
240/    000000,000007               304/    000000,000007
241/    000000,000007               305/    000000,000007
242/    000000,000007               306/    000000,000007
243/    000000,000007               307/    000000,000007
```

The following example illustrates the LSCWs for character data in binary files.

```
C    LOOK AT BINARY MODE FILE WITH CHARACTER DATA AND SEE THE
C    LOGICAL SEGMENT CONTROL WORDS

     OPEN (UNIT=1,MODE='BINARY')

     WRITE (1) 3, 'ABCDEF',4,'GHIJKL'

     WRITE (1) 'MNOPQR','STUVWX'
     END


0/      001000,000007
1/      000000,000003
2/      406050,342212
3/      430000,000000
4/      000000,000004
5/      436211,145226
6/      460000,000000
7/      003000,000010
10/     001000,000004
11/     466351,750242
12/     512472,452654
13/     536600,000000
14/     003000,000005
```

The following example illustrates the format of character data in image files. Image files contain no LSCWs.

```
C    LOOK AT IMAGE MODE FILE WITH CHARACTER DATA AND SEE
C    NO LOGICAL SEGMENT CONTROL WORDS

     OPEN (UNIT=1,MODE='IMAGE')

     WRITE (1) 3, 'ABCDEF',4,'GHIJKL'

     WRITE (1) 'MNOPQR','STUVWX'
     END


0/      000000,000003
1/      406050,342212
2/      430000,000000
3/      000000,000004
4/      436211,145226
5/      462331,647640
6/      506452,352252
7/      532573,000000
```

18.5 USING FOROTS

FOROTS has been designed to lend itself for use as an I/O system for programs written in languages other than FORTRAN. Currently, MACRO programmers may employ FOROTS as a general I/O system by writing simple MACRO calls that simulate the calls made to FOROTS by a FORTRAN compiler. The calls made to FOROTS are to routines that implement FORTRAN I/O statements such as READ, WRITE, OPEN, or CLOSE.

FOROTS will provide automatic memory allocation, data conversion, I/O buffering, and device interface operations to the MACRO user.

18.5.1 FOROTS Entry Points

FOROTS provides the following entry points for calls from either a FORTRAN compiler or a non-FORTRAN program. These entry points are contained in FORLIB.REL.

Table 18-1: FOROTS Entry Points

Entry Point	Function
ALCHN.	Allocates an I/O channel for use by a MACRO subroutine (see Section 18.5.3.12)
ALCOR.	Allocates memory (see Section 18.5.3.11)
CLOSE.	Closes a file. In a FORTRAN program, this call is made when the CLOSE statement is executed (see Section 18.5.3.10)
DBMS.	DBMS interface.
DEC.	DECODE routine. This call, coupled with an IOLIST call, handles decoding.
DECHN.	Deallocates an I/O channel that was obtained from ALCHN (see Section 18.5.3.12)
DECOR.	Deallocates memory that was allocated by an ALCOR call (see Section 18.5.3.11)
ENC.	ENCODE routine
EXIT.	Closes all files, clears interrupt system, and terminates program execution. In a FORTRAN program, this call is made when an END statement is executed in the main program.
EXIT1.	Writes out buffers, closes and unmaps all files
FIN.	I/O list termination routine (see Section 18.5.3.9)
FIND.	FIND statement
FORER.	Error processor
FOROP.	Miscellaneous FOROTS utilities
FUNCT.	OTS-independent interface to provide common functions (like memory and I/O channel management) for programs such as overlay handler and DBMS. FUNCT. is an interface that works the same way with FORTRAN, COBOL, and ALGOL run-time systems (see Section 18.6).
IFI.	Internal file input (see Section 18.5.3.2)

Table 18-1: FOROTS Entry Points (Cont'd)

Entry Point	Function
IFO.	Internal file output (see Section 18.5.3.2)
IN.	Formatted input routine (see Sections 18.5.3.1, 18.5.3.5, 18.5.3.6, and 18.5.3.8)
IOLST.	I/O list routine (see Section 18.5.3.9)
MTOP.	REWIND, BACKSPACE, and ENDFILE statements (see Section 18.5.3.7).
NLI.	NAMELIST input routine (see Section 18.5.3.3)
NLO.	NAMELIST output routine (see Section 18.5.3.3)
OPEN.	Opens a file. Connects FORTRAN Logical Unit Number to a file for I/O (see Section 18.5.3.10)
OUT.	Formatted output routine (see Sections 18.5.3.1, 18.5.3.5, 18.5.3.6, and 18.5.3.8)
RESET.	Job initialization entry
TB.	Binary input routine (see Sections 18.5.3.1 and 18.5.3.5)
TRACE.	Traces subroutine calls
WTB.	Binary output routine (see Sections 18.5.3.1 and 18.5.3.5)

18.5.2 Calling Sequences

You must use the following general form for all calls made to FOROTS:

```
     XMOVEI   16,ARGBLK
     PUSHJ    17,Entry Point
              (control is returned here)
```

where:

ARGBLK — is the address of a specifically formatted argument block that contains information needed by FOROTS to accomplish the desired operation.

Entry Point — is an entry point identifier (see Table 18-1) that specifies the entry point of the desired FOROTS routine.

With three exceptions, all returns from FOROTS will be made to the program instruction immediately following the call (PUSHJ 17, entry point instruction). The exceptions are:

1. An error return to a specified statement number, that is, READ or WRITE statement ERR=option (see Section 10.4.7),

2. An end-of-file return to a statement number, that is, READ or WRITE statement END=option (see Section 10.4.6),

3. A fatal error that returns to the monitor.

Sections 18.5.3.1 through 18.5.3.12 give the MACRO calls and required argument block formats needed to initialize FOROTS and FOROTS I/O operations.

18.5.3 MACRO Calls for FOROTS Functions

The following sections describe the forms of the MACRO calls to FOROTS that are made by the FORTRAN compiler. The calls described are identified by the language statement that they implement. The following terms and abbreviations are used in the description of the argument block (ARGBLK) of each call:

———► = pointer to the second word in the argument block (This is the address pointed to by the argument ARGBLK in the calling sequence.)

u = FORTRAN logical unit number

n = count of ASCII characters

f = FORMAT statement address

list = an Input/Output list

name = a NAMELIST name

r = a variable specifying the logical record number for random access mode

* = list-directed I/O (the FORMAT statement is not used)

type = type specification of a variable or constant

The argument block for all I/O statements is a sequence of keyword specifiers. Bits 2-8 of each argument specify which argument is being supplied, as follows:

1	UNIT
2	FMT address
3	FMT size (in words)
4	END= address
5	ERR= address
6	IOSTAT= address
7	REC=
10	NAMELIST table address
11	File-positioning function code
12	ENCODE/DECODE array address
13	Internal record length

The format of ARGBLK is:

0-1	2-8	9-12	13	14-17	18-35
(-count - negative of number of words in block not including this one)					0
ARGBLK: 1 0	kwd	type	I	0	y
. .					
. .					
. .					
1 0	kwd	type	I	0	y

where:

I = indirection bit
y = 18-bit address or data
kwd = keyword number

NOTE

> Future versions of FOROTS will not support argument blocks with index registers specified either in the arguments or in memory locations referenced indirectly by these arguments. Arguments must not reside in the ACs. In addition, so-called 'immediate' arguments (those with a type code of zero) will not be supported.

18.5.3.1 Sequential-Access Calling Sequences - The READ and WRITE statements for formatted sequential data transfer operations and their calling sequences are:

```
READ(u,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```
WRITE(u,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

The following arguments must be specified in ARGBLK:

1	UNIT
2	FMT address
3	FMT size

The following may also appear:

4	ERR
5	END
6	IOSTAT

The READ and WRITE statements for unformatted sequential data transfer operations and their calling sequences are:

```
READ(u) list

XMOVEI 16, ARGBLK
PUSHJ 17, RTB.
```

and

```
WRITE(u) list

XMOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

The following arguments must be specified in ARGBLK:

1   UNIT

The following may also appear:

4   END
5   ERR
6   IOSTAT

18.5.3.2 Internal File Calling Sequences - The READ and WRITE statements for formatted sequential data transfer operations using internal files and their calling sequences are:

```
READ(u,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, IFI.
```

and

```
WRITE(u,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, IFO.
```

The following arguments must be specified in ARGBLK:

1   UNIT (must be a character scalar or array)
2   FMT address
3   FMT size

The following may also appear:

4   ERR
5   END
6   IOSTAT

18.5.3.3 NAMELIST I/O, Sequential-Access Calling Sequences - The READ and WRITE statements for NAMELIST-directed sequential data transfer operations and their calling sequences are:

```
READ (u, name)

XMOVEI 16, ARGBLK
PUSHJ 17, NLI.
```

and

```
WRITE (u, name)

XMOVEI 16, ARGBLK
PUSHJ 17, NLO.
```

The following arguments must be specified in ARGBLK:

1	UNIT
10	NAMELIST address

The following may also appear:

4	END
5	ERR
6	IOSTAT

The NAMELIST table is generated from the FORTRAN NAMELIST statement. The first word of the table is the NAMELIST name; following that are a number of 2 word entries for scalar variables, and a number of (N+4) word entries for array variables, where N is the dimensionality of the array.

The names you specify in the NAMELIST statement are stored, in SIXBIT format, first in the table. Each name is followed by a list of arguments associated with the name. The NAMELIST table is terminated by a zero entry. The name argument list can be in either a scalar or an array form.

18.5.3.4 Array Offsets and Factoring - Address calculations used to reference a given array element involve factors and offsets. For example:

Array A is dimensioned

```
DIMENSION A (L1:U1,L2:U2,L3:U3,...Ln:Un)
```

The size of each dimension is represented by:

```
S1 = U1-L1+1
S2 = U2-L2+1
  .
  .
  .
```

In order to calculate the address of an element referenced by:

```
A (I1,I2,I3,...In)
```

the following formula is used:

```
A+(I1-L1)+(I2-L2)*S1+(I3-L3)*S2*S1+...+(In-Ln)*S[n-1]*...*S2*S1
```

The terms are factored out depending on the dimensions of the array, not on the element referenced, to arrive at the formula:

```
A+(-L1-L2*S1-L3*S2*S1...)+I1+I2*S1+I3*S2*S1...
```

The parenthesized part of this formula is the offset for a single-precision array; it is referred to as the Array Offset.

For each dimension of a given array, there is a corresponding factor by which a subscript in that position will be multiplied. From the last expression, one can determine the factor for dimension n to be:

```
S[n-1]*S[n-2]*...*S2*S1
```

For double-precision and complex arrays, the expression becomes:

```
A+2*(I1-L1)+2*(I2-L2)*S1+2*(I3-L3)*S2+S1+...
```

Therefore, the array offset for a double-precision array is:

```
2*(-L1-L2*S1-L3*S2*S1...)
```

and the factor for the nth dimension is:

```
2*S[n-1]*S[n-2]*...*S2*S1
```

The factor for the first dimension of a double-precision array is always 2. The factor for the first dimension of a single-precision array is always 1.

For character arrays, the offset is calculated in bytes instead of words. The byte offset from the start of a character array whose elements are of length X is:

```
X*((I1-L1)+(I2-L2)*S2+...)
```

This offset is X times the offset of a single-precision numeric array.

NAMELIST Table

0   35
NAMELIST name in SIXBIT
NAMELIST entry 1 NAMELIST entry 2 . . . NAMELIST entry n
4000,,0 (FOROTS FIN. word)

SCALAR ENTRY in a NAMELIST Table

0 1	2-8	9-12	13	14-17	18-35
SIXBIT/SCALAR NAME/					
1 0	type	0	I	0	Scalar addr

ARRAY ENTRY in a NAMELIST Table

0-1	2-8	9-12	13	14-17	18-35
SIXBIT/ARRAY NAME/					
1 0	#DIMS	type	I	0	BASE ADDR
ARRAY SIZE OFFSET Factor 1 Factor 2 . . . Factor n					

18.5.3.5 I/O Statements, Direct-Access Calling **Sequences** - The READ and WRITE statements for formatted direct-access data transfers and their calling sequences are:

```
READ (u'r,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, IN.

and

WRITE (u'r,f) list

XMOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

The following arguments must be specified in ARGBLK:

1	UNIT
2	FMT address
3	FMT size
7	REC

The following may also appear:

4	END
5	ERR
6	IOSTAT

The READ and WRITE statements for unformatted direct-access transfers and their calling sequences are:

```
READ (u'r)

XMOVEI 16,ARGBLK
PUSHJ P,RTB.
```

and

```
WRITE (u'r)

XMOVEI 16,ARGBLK
PUSHJ P,WTB.
```

The following argument must be specified in ARGBLK:

1	UNIT
7	REC

The following may also appear:

4	END
5	ERR
6	IOSTAT

18.5.3.6 Default Devices Statements, Calling Sequences - The FORTRAN statements that require the use of a reserved system default device and their calling sequences are:

	Default Device	
ACCEPT f, list	UNIT=-4	(TTY)
READ f, list	UNIT=-5	(CDR)
REREAD f, list	UNIT=-6	(REREAD)

```
XMOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

PRINT f, list	UNIT=-3	(LPT)
PUNCH f, list	UNIT=-2	(PTP)
TYPE f, list	UNIT=-1	(TTY)

```
XMOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

The arguments for these calls are the same as for the standard formatted sequential READ and WRITE statements.

18.5.3.7 **Statements to Position Files** - The formatted and unformatted FORTRAN statements that can be used to control the positioning of files and their calling sequences are:

Function (FORTRAN Statement)	FOROTS Code
SKIPFILE (u)	7
BACKFILE (u)	3
BACKSPACE (u)	2
ENDFILE (u)	4
REWIND (u)	0
SKIPRECORD (u)	5
UNLOAD (u)	1

CALL:

```
     XMOVEI 16, ARGBLK
     PUSHJ 17, MTOP.
```

The following arguments must be specified in ARGBLK:

1	UNIT
11	FOROTS code

The following may also appear:

4	END
5	ERR
6	IOSTAT

NOTE

For disk files, UNLOAD is the same as REWIND; BACKFILE and SKIPFILE are ignored.

18.5.3.8 **List-Directed Input/Output Statements** - You may write any form of a sequential I/O statement as a list-directed statement by replacing the referenced FORMAT statement number with an asterisk (*).

The list-directed forms of the READ and WRITE statements and their calling sequences are:

```
     READ (u, *) list

     XMOVEI 16, ARGBLK
     PUSHJ 17, IN.
```

and

```
     WRITE (u, *) list

     XMOVEI 16, ARGBLK
     PUSHJ 17, OUT.
```

The arguments for these calls are the same as for the standard formatted sequential READ and WRITE statements, except that the FORMAT statement address and FORMAT statement size must be specified as zero.

18.5.3.9 Input/Cutput Data Lists - The compiler generates a calling sequence to the run-time system for the I/O list in a READ or WRITE statement. The argument block associated with the calling sequence contains the addresses of the variables and arrays to be transferred to or from an I/O buffer.

The general form of an I/O list calling sequence is:

```
XMOVEI 16, ARGBLK
PUSHJ 17, IOLST.
```

Any number of elements may be included in the ARGBLK. The end of the argument block is specified by a zero entry or a FIN entry.

Mnemonic Name	FOROTS Value
DATA	1
SLIST	2
ELIST	3
FIN	4
F77 SLIST	5
F77 ELIST	6

The elements of an I/O list are:

1. DATA

   The DATA element converts one single- or double-precision or complex item from external to internal form for a READ statement and from internal to external form for a WRITE statement. Each DATA element has the following format:

0-1	2-8	9-12	13	14-17	18-35
1 0	DATA	type	I	0	SCALAR ADDR

2. SLIST

   The SLIST argument converts an entire array from internal to external form or vice versa, depending on the type of statement (that is, READ or WRITE) involved. An SLIST consist of a table of arguments that has the following form:

0-1	2-8	9-12	13	14-17	18-35
1 0	SLIST	type	I	0	#ELEMENTS
1 0	0	type	I	0	INCREMENT
1 0	0	type	I	0	BASE ADDR1

For example, the sequence:

```
DIMENSION A(100),B(100)
READ(-,-)A
```

or

```
READ(-,-)(A(I),I=1,100) !only when the /OPT switch is used
```

develops an SLIST argument of the form:

0-1	2-8	9-12	13	14-17	18-35
1 0	2	2	0	0	[144]
1 0	0	2	0	0	[1]
1 0	0	2	0	0	A
0 0	4	0	0	0	0

More than one base address may appear in a SLIST as long as the increment is the same. The sequence:

```
DIMENSION A(100), B(100)
WRITE (-,-) (A(I),B(I),I=100) !only when the /OPT
                                switch is used
```

develops an SLIST argument of the form:

0-1	2-8	9-12	13	14-17	18-35
1 0	2	2	0	0	[144]
1 0	0	2	0	0	[1]
1 0	0	2	0	0	A
1 0	0	2	0	0	B
0 0	4	0	0	0	0

3. ELIST

The SLIST format permits only a single increment to be specified for a number of arrays, while the ELIST permits different increments to be specified for different arrays. An ELIST consists of a table of arguments that has the following form:

The format of the ELIST is:

0-1	2-8	9-12	13	14-17	18-35
1 0	ELIST	type	I	0	No. Elements to transfer increment 1
1 0	0	type	I	0	Base ADDR 1 increment 2
1 0	0	type	I	0	Base ADDR 2 increment N
1 0	0	type	I	0	Base ADDR N

For example, the FORTRAN sequence:

```
DIMENSION IC(6,100), IB(100)
WRITE(-,-) (IB(I),IC(1,I),I=1,100)
```

produces the ELIST:

0-1	2-8	9-12	13	14-17	18-35
1 0	3	2	0	0	[144]
1 0	0	2	0	0	[1]
1 0	0	2	0	0	IB
1 0	0	2	0	0	[6]
1 0	0	2	0	0	IC
0 0	4	0	0	0	0

The increment may be zero. This could be produced by the sequence:

```
WRITE(-,-)(K,I=1,100)   !only when the /OPT switch is used
```

Produces the ELIST:

0-1	2-8	9-12	13	14-17	18-35
1 0	3	2	0	0	[144]
1 0	0	2	0	0	[0]
1 0	0	2	0	0	K
0 0	4	0	0	0	0

4. FIN

The end of an I/O list is indicated by a FIN element. When the I/O processor interprets this element, it performs a call to FIN to terminate the I/O. This call must be made after each I/O initialization call, including calls with a null I/O list.

The FIN routine may be entered by an explicit call or by an argument in this I/O list argument block. Both calls can not be used. The FIN element has the following format:

0-1	2-8	9-12	13	14-17	18-35
0 0	4	0	0	0	0

EXPLICIT CALL:

```
PUSHJ 17, FIN.
```

5. F77 SLIST

   This is the same as SLIST except that if the number of elements is less than or equal to zero, no I/O is done.

6. F77 ELIST

   This is the same as ELIST except that is the number of elements is less than or equal to zero, no I/O is done.

18.5.3.10 OPEN and CLOSE Statements, Calling Sequences - The form and calling sequences for the OPEN and CLOSE FORTRAN statements are:

OPEN statement call:

```
XMOVEI 16, ARGBLK
PUSHJ 17, OPEN.
```

CLOSE statement call:

```
XMOVEI 16, ARGBLK
PUSHJ 17, CLOSE.
```

where ARGBLK is:

0-1	2-8	9-12	13	14-17	18-35
Negative of the number of words in block not including this one.					0
1 0	G	type	I	X	H
1 0	G	type	I	X	H
1 0	G	type	I	X	H
. .	.	.	.	.	.
. .	.	.	.	.	.
. .	.	.	.	.	.
. .	.	.	.	.	.
1 0	G	type	I	X	H

The G field (bits 2 through 8) contains a value that defines the argument name; the H field (bits 18 through 35) contains an address that points to the value of the argument. Note that the G field values for OPEN and CLOSE statements are not the same as those for other I/O statements.

The numeric codes that may appear in the G field are:

G Field	Open Argument	G Field	Open Argument
01	DIALOG	27	FORM=
02	ACCESS=	30	BYTESIZE=
03	DEVICE=	31	PADCHAR=
04	BUFFERCOUNT=	32	RECORDTYPE=
05	BLOCKSIZE=	33	STATUS=
06	FILE=	34	TAPEFORMAT=
07	PROTECTION=	35	READONLY=
10	DIRECTORY=	36	UNIT=
11	LIMIT=	37	ERR=
12	MODE=	40	EXIST=
13	FILESIZE=	41	FORMATTED=
14	RECORDSIZE=	42	NAMED=
15	DISPOSE=	43	NEXTREC=
16	VERSION=	44	NUMBER=
17	Reserved	45	OPENED=
20	Reserved	46	SEQUENTIAL=
21	IOSTAT=	47	UNFORMATTED=
22	ASSOCIATEVARIABLE=	50	NAME=
23	PARITY=	51	Reserved
24	DENSITY=	52	Reserved
25	BLANK=	53	DIALOG=
26	CARRIAGECONTROL=		

18.5.3.11 Memory Allocation Routines - The memory management module is called to allocate or deallocate memory blocks. There are two entry points (ALCOR. and DECOR.) that control memory allocation and deallocation.

When TOPS-20 extended addressing is in effect, ALCOR. and DECOR. can be used; however, memory will be allocated in FOROT's section instead of in the user's section. You can use the LINK switch /OTSEGMENT:NONSHARABLE to put FOROTS in the user's section.

Use the ALCOR. entry to allocate the number of words specified in the argument block variable. Upon return, AC 0 will contain either the address of the allocated memory block or -1, which indicates that memory is not available.

The calling sequence for an ALCOR. call is:

```
XMOVEI 16, ARGBLK
PUSHJ 17, ALCOR.
```

where ARGBLK is:

0-1	2-8	9-12	13	14-17	18-35
-1					0
1 0	Reserved	type	I	0	Address of number of words

Use the DECOR. entry to deallocate a previously allocated block of memory; the argument variable must be loaded with the address of the memory block to be returned.

If the number of desired words is N, ALCOR. actually removes N+2 words from free storage. The pointer returned points to the third word (word 2 as opposed to word 0) removed from free storage. The first two words are used by FOROTS to maintain linked lists of allocated (using ALCOR.) and free storage, and must not be modified.

The calling sequence for a DECOR. call is:

```
XMOVEI 16, ARGBLK
PUSHJ 17, DECOR.
```

where ARGBLK is:

	0-1	2-8	9-12	13	14-17	18-35
	-1					0
→	1 0	Reserved	type	I	0	Pointer to word containing address of block to be returned

18.5.3.12 Channel Allocation and Deallocation Routines - You may allocate software channels in MACRO programs by means of calls to the ALCHN. routine and deallocate them by calls to the DECHN. routine. Values are returned in AC 0.

Use the ALCHN. entry to allocate a particular channel or the next available channel. The channel to be allocated is passed as an argument to ALCHN. Zero is passed as an argument to allocate the next available channel. Allowed channels are 1 through 17 (octal). If the channel requested is not available, or all channels are in use, ALCHN. returns with a -1 in AC 0. In normal returns, AC 0 contains the assigned number.

The calling sequence of an ALCHN. routine is:

```
XMOVEI 16, ARGBLK
PUSHJ 17, ALCHN.
```

where ARGBLK is:

	0-1	2-8	9-12	13	14-17	18-35
	-1					0
→	1 0	Reserved	type	I	0	Pointer to a word containing the channel # or zero

Use the DECHN. entry to deallocate a previously assigned channel. The channel to be released is passed as an argument to DECHN. If the channel to be deallocated was not assigned by ALCHN. and thus cannot be deassigned, AC 0 is set to -1 on return.

The calling sequence for a DECHN. routine is:

```
XMOVEI 16, ARGBLK
PUSHJ 17, DECHN.
```

where ARGBLK is:

	0-1	2-8	9-12	13	14-17	18-35
	-1					0
→	1 0	Reserved	type	I	0	Pointer to a word containing the channel # to be released

18.6 FUNCTIONS TO FACILITATE OVERLAYS

FOROTS provides a subroutine (FUNCT.) to serve as an interface with the LINK overlay handler. This subroutine consists of a group of functions that allow the overlay handler to perform I/O, memory management, and error message handling. These functions have only one entry point, FUNCT.; and they are called by the sequence:

```
XMOVEI 16, ARGBLK
PUSHJ 17, FUNCT.
```

The format of the ARGBLK is:

```
                  -<n+3>,,0
ARGBLK:           IFIW 2,address of integer function code
                  IFIW 17,address of 3-letter ASCII error prefix
                  IFIW 2,address of status code on return
                  IFIW type,address of first argument
                  . . .
                  IFIW type,address of nth argument
```

where:

type — is the FORTRAN argument type (see Chapter 14)
function code — is the number of one of the required functions
error prefix — is ignored by FOROTS
status — is undefined on the call and set on the return with one of the values below.

-1	Function not implemented
0	Successful return
1....n	Specific error message

When TOPS-20 extended addressing is in effect, FUNCT. can be used; however, memory will be allocated in FOROTS's section instead of in the user's section. You can use the LINK switch /OTSEGMENT:NONSHARABLE to put FOROTS in the user's section.

Table 18-2: Function Numbers and Function Codes

Function Number (Octal)	Function Mnemonic	Function Description
0	ILL	Illegal function; returns -1 status
1	GAD	Gets a specific segment of memory
2	COR	Gets a given amount of memory from anywhere in the space allocated to the overlay handler
3	RAD	Returns a specific segment of memory
4	GCH	Gets an I/O channel
5	RCH	Returns an I/O channel
6	GOT	Gets memory from the space allocated to the object-time system
7	ROT	Returns memory to the object-time system
10	RNT	Gets the initial run time from the object-time system
11	IFS	Gets the initial run time file specification from the object-time system
12	CBC	Cuts back memory (if possible) to reduce job size
13	RRS	Reads RETAIN status (DBMS)
14	WRS	Writes RETAIN status (DBMS)
15	GPG	Allocates memory on a page boundary
16	RPG	Deallocates memory obtained by function 15
17	GPSI	Gets TOPS-20 PSI channel
20	RPSI	Returns TOPS-20 PSI channel
21	MPG	Gets a contiguous set of pages
22	UPG	Returns a contiguous set of pages

Each function of the FUNCT. subroutine is described below. The arguments described in each of the following functions are what should be in the addresses pointed to by the argument block described above.

ILL (0) FUNCTION - This function is illegal. The argument block is ignored, and the status returned is -1.

GAD (1) FUNCTION - This function gets memory from a specific address in the space allocated to the overlay handler. The arguments are:

arg 1	address of requested memory
arg 2	address of number of words of memory to allocate

A call to GAD with arg 2 equal to -1 requests all available memory.

On return, the status is one of the following:

0	successful allocation
1	not enough memory available
2	memory not available at specified address
3	illegal arguments (such as, address + size is greater than 256K)

COR (2) FUNCTION - This function gets memory from any available space allocated to the overlay handler. The arguments are:

arg 1	undefined (address of allocated memory on return)
arg 2	address of size of requested allocation

On return, the status is one of the following:

0	memory allocated
1	not enough memory available
3	illegal argument (that is, size is greater than 256K)

RAD (3) function - This function returns memory at the specified address within the space allocated to the overlay handler. The arguments are:

arg 1	address of memory to be return
arg 2	address of size of memory to be returned (in words)

On return, the status is one of the following:

0	successful return of memory
1	memory cannot be returned
3	illegal argument (that is,the address or the size is greater than 256K)

GCH (4) FUNCTION - This function gets an I/O channel. The arguments are:

arg 1	undefined (address of channel number allocated on return)
arg 2	ignored

On return, the status is one of the following:

0	successful channel allocation
1	no I/O channels available

RCH (5) FUNCTION - This function returns an I/O channel. The arguments are:

arg 1	address of number of channel to be returned
arg 2	ignored

On return, the status is one of the following:

0	channel released
1	invalid channel number

GOT (6) FUNCTION - This function gets memory from the space allocated to the object-time system. The arguments are:

arg 1	undefined (address of allocated memory on return)
arg 2	address of size of memory requested

On return, the status is one of the following:

0	successful allocation
1	not enough memory available
3	illegal argument (such as, size is greater than 256K)

This function differs from function 1 in that if the object-time system has two free memory lists, then function 1 is used to allocate space for links, and this function is used to allocate space for I/O buffers. Function 1 uses the free memory list for LINK, and function 6 uses the list for the object-time system.

ROT (7) FUNCTION - This function returns memory to the object-time system. The arguments are:

arg 1 address of memory to be returned
arg 2 address of size of memory to be returned (in words)

On return, the status is one of the following:

0 successful return of memory
1 memory cannot be returned
3 illegal argument (such as, the address or the size is greater than 256K)

RNT (10) FUNCTION - This function returns the initial run time from the object-time system. (At the beginning of the program, the object-time system will have executed a RUNTIM UUO; that result is the time returned by RNT.) The arguments are:

arg 1 undefined (contains address of initial run time on return)
arg 2 ignored

On return, the run time is in arg 1, and the status is 0. The status is 0.

IFS (11) FUNCTION (TOPS-10 only) - This function returns the initial run-time file specification from the object-time system. (This initial file specification is the one used to begin the program; that is, it was given with the system RUN command.) The arguments are:

arg 1 undefined (address of SIXBIT device on return)
arg 2 undefined (address of SIXBIT filename on return)
arg 3 undefined (project-programmer number on return or address of path block if run from SFD

On return, the status is one of the following:

0 successful return
1 error

This function tells the overlay handler which file to read after the initial RUN command.

CBC (12) FUNCTION - This function cuts back memory if possible, which reduces the size of the user job. There are no arguments.

The returned status is:

0 always

RRS (13) FUNCTION
WRS (14) FUNCTION - These functions are reserved for use by DBMS.

GPG (15) FUNCTION - This function gets memory on a page boundary. The arguments are the same as for FUNCTION 6, GOT.

arg 1 ignored
arg 2 address of number of words of memory to allocate

On return, arg 1 has the address of the allocated memory. It will be on a page boundary. (That is, it will be a multiple of 1000 octal.)

On return, the status is one of the following:

0	successful
1	not enough memory available
3	illegal argument

RPG (16) FUNCTION - This function returns memory obtained by FUNCTION 15. The arguments are the sare as for FUNCTION 7, ROT.

arg 1	address of memcry to be returned
arg 2	address of size of memory to be returned (in words)

On return, the status is one of the following:

0	successful return of memory
1	was not allocated
3	illegal argument

GPSI (17) FUNCTION - This function gets the TOPS-20 PSI channel. It assigns a software interrupt channel number. GPSI provides only controlled access to the PSI tables. It arranges that the tables exist and that SIR and EIR have been done, but does not do AIC or any other JSYS necessary to set up the channel (ATI or MTOPR, for example).

arg 1	address of channel number to allocate, or -1 to allocate any user-assignable channel
arg 2	address of level number
arg 3	address of interrupt routine (if the FORTRAN program is running on a system that supports extended addressing, the interrupt routine address may be a 30-bit address in any section, including section 0. Otherwise, the interrupt address must be an 18-bit address.)

On return, the status is one of the following:

0	allocated OK (arg 1 is the channel number if -1 was sent)
1	requested channel was already assigned
2	no free channels
3	argument error

RPSI (20) FUNCTION - This function returns the TOPS-20 PSI channel. It returns a PSI channel allocated by FUNCTION 17. RPSI provides only controlled access to the PSI tables. It removes the given channel from the tables. This function does not do DIC or any other JSYS necessary to remove an interrupt condition from a channel.

arg 1	address of channel number to return

On return, the status is one of the following:

0	OK
1	channel was not allocated
3	argument error

MPG (21) FUNCTION - This function gets a contiguous set of pages. The pages requested are always allocated from the section FOROTS is in. The user cannot depend upon this call to either create or destroy the pages.

arg 1 first page number to allocate. The page number must be in the range 0 to 777.

arg 2 number of pages to allocate

On return, the status is one of the following:

0 successful allocation of all given pages
1 one or more pages were already allocated
3 illegal argument (bad page number or count)

UPG (22) FUNCTION- This function returns a contiguous set of pages. The pages returned are considered to be in the section FOROTS is in. The user cannot depend upon this call to either create or destroy the pages.

arg 1 first page number to deallocate. The page number must be in the range 9 to 777

arg 2 number of pages to deallocate

On return, the status is one of the following:

0 successful deallocation of all given pages
1 one or more pages was not allocated by MPG
3 illegal argument (bad page number or count)

18.7 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

You make FORTRAN logical and physical device assignments at run time, or standard system assignments are made according to a FOROTS Device Table, that is, DEVTB. Table 10-3 in Section 10.4.3 shows the standard assignments contained by the Device Table.

18.8 FOROTS AND INQUIRE BY FILE STATEMENT

See Section 11.7 for a description of the INQUIRE statement.

If no device is given for the FILE= specifier, FOROTS uses DSK: as the default. If no extension is given, FOROTS uses .DAT. For TOPS-20, if no generation number is given, FOROTS uses a generation number of 0.

FOROTS determines if the device specified is a disk. If the device is a disk, the following happens:

- FOROTS determines if a file exists with the file specification given in the INQUIRE statement. It returns the answer (either .TRUE. or .FALSE.) in the variable specified by the EXIST= specifier, if any. If such a file exists, FOROTS 'expands' the file specification as follows:

  - A logical name is translated into a physical device name.
  - For TOPS-20, the file specification, which is overlaid by the user-specified directory, filename, extension, and generation.
  - For TOPS-20, an actual file generation number is substituted for a generation number of 0, -1, or -2.

The resultant file specification, in string form, is called the 'full (expanded) file string.'

- FOROTS searches for a match between the file specification given in the INQUIRE statement and a file specification associated with a logical unit for which there is a "connection." This is to determine the values to be returned for the INQUIRE specifiers OPENED= and NUMBER= (see Section 11.7.3). FOROTS looks at all FORTRAN logical units for which there is a connection in ascending order, starting with zero.

  FOROTS compares the file specification given in the INQUIRE statement (with FILE= defaults applied) with the exact file specification given in the OPEN statement (with FILE= defaults applied) if the following is true:

  - The file does not exist on the directory.
  - An OPEN statement has been executed and STATUS='UNKNOWN' and ACCESS='SEQUENTIAL' (see Section 11.3.1).
  - No data transfer statements have been executed using the unit.

  If the file exists, FOROTS compares the full file string associated with the unit with the full (expanded) file string given in the INQUIRE statement. The file exists if the following is true:

  - An OPEN has been executed with STATUS other than 'UNKNOWN' or ACCESS other than 'SEQUENTIAL'.
  - An I/O transfer statement has been executed.

  If neither of these two comparisions are successful, FOROTS returns the current unit number in the variable specified with the NUMBER= specifier, and returns 'YES' in the variable specified with the OPENED= specifier. If the same file is connected on several units, the matching technique described will return the smallest unit number for which there is a match.

For non-disk devices specified in the INQUIRE statement file string specification, FOROTS looks at all the FORTRAN logical units for which there is a connection in ascending order, starting with zero.

If the device in the file string specified in the INQUIRE statement is not the user's controlling terminal, FOROTS expands the file specification given in the INQUIRE statement by translating a logical name given as the device into its corresponding physical name. It then compares the device part of this expanded file specification with the device part of the full (expanded) file string associated with the logical unit.

If the device is the user's controlling terminal (device 'TTY'), FOROTS determines if the device associated with the logical unit is also the user's controlling terminal.

CHAPTER 19

USING THE FORTRAN REAL-TIME SOFTWARE (TOPS-10 ONLY)

19.1 INTRODUCTION

The FORRTF library subroutines are designed to allow the timesharing FORTRAN user to do real-time programming on TOPS-10 systems. These subroutines, described in Section 19.3, are listed below:

LOCK
RTINIT
CONECT
RTSTRT
BLKRW
RTREAD
RTWRIT
STATO
STATI
RTSLP
RTWAKE
DISMIS
DISCON
UNLOCK

With these subroutines, the timesharing job can dynamically connect real-time devices to the Priority Interrupt (PI) system, respond to these devices at interrupt level, remove the devices from the PI system, and change their PI level. Use of these routines requires that you have real-time privileges and are able to lock your job in core. The privilege bits required are:

JP.RTT	(bit 13)	- real-time privileges
JP.LCK	(bit 14)	- locking privileges

The number of real-time devices that can be handled at one time is an assembly-time constant (RTDEVN) in the FORRTF source. The DIGITAL-distributed software has RTDEVN equal to 2, but it can be changed (up to 6) by editing the statement "RTDEVN==2" in FORRTF.MAC and reassembling.

The error messages output by FORRTF can be in either full message format or coded format (refer to Table 19-1). Use of the code and format saves over 100 words of run-time core. If core is limited, reassembly of FORRTF.MAC with the assembly-time constant SHORT changed from the DIGITAL-distributed 0 (full format) to -1 (coded format) accomplishes the core saving.

On multiprocessor systems, the real-time traps apply only to the processor specified by the job's CPU specification. If the specification indicates more than one processor, the specification is changed to indicate CPU0. Note that the priority interrupt channel is only for the indicated CPU.

19.2 USING FORRTF

Users of FORTRAN-10 real-time software must consider the following:

1. Use of memory

2. Device control in block or single mode

3. Priority-interrupt levels

4. Masks

19.2.1 Memory

The job being executed must be locked in memory with the LOCK subroutine (see Section 19.3.1). Any data being read into memory can only be read into the low segment and above the protected job data area (the first 140 locations). The BLKRW subroutine (see Section 19.3.5) tests the validity of the locations specified to receive data in block reading to ensure that no overwritings occur.

However, when in block mode, the block pointer must be reset before dismissing the end-of-block interrupt; otherwise, all memory could be overwritten.

19.2.2 Modes

Real-time jobs can control their devices in one of two ways: block mode or single mode. In block mode, an entire block of data is read or written before the user-interrupt routine is run; whereas, in single mode, the user-interrupt program is run every time the device interrupts.

There are two types of block mode: fast-block mode and normal-block mode. A device in fast-block mode requires that a PI channel be dedicated entirely to itself.

19.2.3 Priority-Interrupt Levels

Priority-interrupt levels 1 through 6 are legal depending on the system configuration. The lower the number of the level, the higher the priority of that level. Programs that execute for a long time should not be put on high-priority interrupt levels, since they could cause other real-time programs on lower levels to lose data. Specification of the PI level as zero for a particular device causes the device to be removed from the PI system.

19.2.4 Masks

For a description of the bits included in the startmsk and intmsk parameters of RTSTRT and the status word in STATO and STATI, see the DECsystem-10 Hardware Reference Manual.

19.3 SUBROUTINES

Each of the 14 subroutines associated with FORTRAN real-time software is described briefly in Sections 19.3.1 through 19.3.14. These subroutines have been programmed to be compatible with programs written according to the TOPS-10 Monitor Calls Manual.

19.3.1 LOCK

LOCK locks the job in memory and allocates and initializes the internal controlling tables for all real-time devices. LOCK must be called before any of the other real-time routines, and must be called exactly once.

The form of the LOCK subroutine is:

```
     CALL LOCK
```

19.3.2 RTINIT

RTINIT initializes the internal tables controlling a real-time device. RTINIT must be called for each individual device being used.

The form of the RTINIT subroutine is:

```
     CALL RTINIT (unit, dev, pi, trpadr, intmsk)
```

where:

unit — is the real-time device unit number (any number from 1 to RTDEVN). This number is not connected in any way with the FORTRAN logical unit number.

dev — is the device code for the real-time device (see the DECsystem-10 Hardware Reference Manual).

pi — is the priority-interrupt level on which the real-time device is to be run. Each individual device in fast-block mode must have a level dedicated to itself. If the level is equal to zero, the device will be removed from the priority-interrupt system altogether.

If it is necessary to connect one device to several levels simultaneously, a negative value for PI tells the system not to remove any other occurrences of the device from any other (or the same) PI level. (Note that this counts as another real-time device.)

trpadr — is the address of a FORTRAN entry to which real-time interrupts are to trap. This can be a function or subroutine subprogram. Any variables that must be shared between the user-level code and the interrupt-level routine must be passed by means of COMMON. Passing them as parameters causes disastrous results.

intmsk — is the mask of all interrupting flags for the real-time device. This is actually set up by RTSTRT and should be zero whenever the real-time device is inactive (that is, in a call to RTINIT, except in the case of fast-block mode). In fast-block mode, intmsk must be set to -1.

19.3.3 CONECT

CONECT tells the system to connect a real-time device to the proper PI level and sets up several elements of the device-controlling tables. Every device must be CONNECTED.

The form of the CONECT subroutine is:

```
     CALL CONECT (unit, mode)
```

where:

unit — is the real-time device unit number (see RTINIT).

mode — is either:

-2, write a block of data, fast mode; then interrupt.
-1, write a block of data, normal mode; then interrupt.
0, interrupt every word
+1, read a block of data, normal mode; then interrupt.
+2, read a block of data, fast mode; then interrupt.

19.3.4 RTSTRT

RTSTRT can be used to start a real-time device, as well as to stop it and zero its interrupt mask. A device must be started to be used and should be stopped before it is disconnected. The form of the RTSTRT subroutine is:

```
     CALL RTSTRT (unit, startmsk, intmsk)
```

where:

unit — is the real-time device unit number (see RTINIT).

startmsk — is the flags necessary to start the device (see the <u>DECsystem-10 Hardware Reference Manual</u>). If the device is being stopped, this parameter should be zero.

intmsk — is the mask of all interrupting bits for the particular device (see the <u>DECsystem-10 Hardware Reference Manual</u>). If the device is in fast-block mode and being started, intmsk should equal -1; if, however, the device in any mode is being stopped, the parameter must be 0.

19.3.5 BLKRW

BLKRW is used with either of the block modes. It sets up the size and starting address of the data block being handled. A new count and starting address must be set up each time the current one runs out.

The form of the BLKRW subroutine is:

```
     CALL BLKRW (unit, count, blkadr)
```

where:

unit — is the real-time device unit number (see RTINIT).

count — is the number of words to be read or written.

blkadr — is the array into which the data is to be written or from which it is to be read.

19.3.6 RTREAD

RTREAD, used with a device in single mode, reads a single word of data from the device.

The form of the RTREAD subroutine is:

```
     CALL RTREAD (unit, datadr)
```

where:

unit — is the real-time device unit number (see RTINIT).

datadr — is the address of the location in which to store the data read.

19.3.7 RTWRIT

RTWRIT sends a single word of data to a real-time device in single mode.

The form of the RTWRIT subroutine is:

```
     CALL RTWRIT (unit, datadr)
```

where:

unit — is the real-time device unit number (see RTINIT).

datadr — is the location of the data word to be sent to the device.

19.3.8 STATO

STATO sends the specified status word to the status register of a real-time device.

The form of the STATO subroutine is:

```
     CALL STATO (unit, statadr)
```

where:

unit — is the real-time device unit number (see RTINIT).

statadr — is the location of the word of status bits to be sent to the real-time device.

19.3.9 STATI

STATI reads the current device status bits into the location specified for inspection by the FORTRAN program.

The form of the STATI subroutine is:

```
CALL STATI (unit, adr)
```

where:

unit — is the real-time device unit number (see RTINIT).

adr — is the location into which the device status bits are to be read.

19.3.10 RTSLP

RTSLP is called from the timesharing level and causes the FORTRAN job to sleep until RTWAKE is called from interrupt level. The program goes to sleep for the specified number of seconds (up to 60). When it wakes up, it checks to see if RTWAKE has been called from interrupt level. If RTWAKE has been called, RTSLP returns to the calling program; otherwise the job goes back to sleep again.

The form of the RTSLP subroutine is:

```
CALL RTSLP (time)
```

where:

time — is the length of sleep time in seconds.

19.3.11 RTWAKE

RTWAKE is called at interrupt level to wake up the FORTRAN program.

The form of the RTWAKE subroutine is:

```
CALL RTWAKE
```

19.3.12 DISMIS

DISMIS dismisses the interrupt currently being processed. The user-interrupt routine must be sure to dismiss the interrupt that causes its execution to begin.

The form of the DISMIS subroutine is:

```
CALL DISMIS
```

19.3.13 DISCON

DISCON disconnects a real-time device from its PI level. All devices should be disconnected through calls to DISCON before the job is terminated.

The form of the DISCON subroutine is:

```
     CALL DISCON (unit)
```

where:

unit   is the real-time device unit number (see RTINIT).

19.3.14 UNLOCK

UNLOCK unlocks the job from core. When execution of a job is complete, the job is automatically unlocked before the return to the monitor. The UNLOCK subroutine provides a method to unlock a job before execution is complete. Note that all real-time device handling must be finished before the job is unlocked.

The form of the UNLOCK subroutine is:

```
     CALL UNLOCK
```

19.3.15 Error Messages

Table 19-1 lists real-time software error messages, including the code format, the full message format, and the subroutine in which the message occurs.

Table 19-1: Error Messages - Code Format and Full Message Format

Code Format	Full Message Format	Subroutine in which message occurs
1	?ILLEGAL UNIT NUMBER. TO HANDLE MORE DEVICES, REASSEMBLE FORRTF WITH A LARGER A ?ERROR COMES FROM THE SUBROUTINE "subroutine name"	"RTDEVN"
2	?RTINIT MUST BE CALLED BEFORE CONECT	CONECT
3	?CONECT MUST BE CALLED BEFORE RTSTRT OR BLKRW	RTSTRT,BLKRW
4	?REAL TIME BLOCK OUT OF BOUNDS A ?END OF BLOCK TOO HIGH [such as, overwrites some program or in high segment] B ?END OF BLOCK TOO LOW, such as, start address less than 140	 BLKRW
5	?JOB CANNOT BE LOCKED IN CORE A ?JOB NOT PRIVILEGED B ?NOT ENOUGH CORE AVAILABLE FOR LOCKING	LOCK
6	?APR ERROR AT INTERRUPT LEVEL A ?PDL OVERFLOW B ?ILLEGAL MEMORY REFERENCE	
7	?RTTRP ERROR realtime trap error of the following sort A ?ILLEGAL PI NUMBER PI channel not available B ?TRAP ADDRESS OUT OF BOUNDS C ?SYSTEM LIMIT FOR REALTIME DEVICES EXCEEDED D ?JOB NOT LOCKED IN CORE OR NOT PRIVILEGED E ?DEVICE ALREADY IN USE BY ANOTHER JOB 0 ?OCCURRED IN THE DISCON ROUTINE 1 ?OCCURRED IN THE CONECT ROUTINE	DISCON CONECT
8	A ?NOT ENOUGH CORE AVAILABLE FOR THE CONTROL BLOCKS B ?NOT ENOUGH CORE AVAILABLE	LOCK

APPENDIX A

SUMMARY OF FORTRAN STATEMENTS

This appendix summarizes the forms of all FORTRAN statements and provides a section reference where each statement is described in detail.

Form	**Section Reference**
ACCEPT(FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT(FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist] ACCEPT f[,iolist] ACCEPT *[,iolist]	Section 10.8
ASSIGN s to i	Section 8.3
BACKFILE un BACKFILE (UNIT=un[,ERR=][,IOSTAT=ios]) BACKFILE (un[,ERR=s][,IOSTAT=ios])	Section 11.8.8
BACKSPACE un BACKSPACE (UNIT=un[,ERR=s][,IOSTAT=ios]) BACKSPACE (un[,ERR=s][,IOSTAT=ios])	Section 11.8.4
BLOCK DATA [sub]	Section 13.1
CALL sub [([a1 [,a2]...])]	Section 13.4.2.2
CHARACTER[*LEN[,]]v[*LEN][,v[*LEN]]	Section 7.2.2
CLOSE (closelist)	Section 10.16
COMMON [/[cb]/]nlist[[,]/[cb]/nlist]...	Section 7.4
COMPLEX v [,v...]	Section 7.2
CONTINUE	Section 9.4
DATA nlist/clist/ [[,]nlist/clist/]...	Section 7.9
DECODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist]	Section 10.12
DIMENSION a(d) [,a(d)...]	Section 7.1
DO s [,] i=e1,e2[,e3]	Section 9.3.1
DO s [,] WHILE (e)	Section 9.3.2
DOUBLE PRECISION v [,v...]	Section 7.2
ELSE	Section 9.2.4
ELSE IF (e) THEN	Section 9.2.4
ENCODE(c,f,a[,ERR=s][,IOSTAT=ios])[iolist	Section 10.12
END	Section 9.7

SUMMARY OF FORTRAN STATEMENTS

Form	Section Reference
END DO	Section 9.4
END IF	Section 9.2.4

ENDFILE un
ENDFILE (UNIT=un[,ERR=s][,IOSTAT=ios])
ENDFILE (un[,ERR=s][,IOSTAT=ios])
	Section 11.8.5

ENTRY en [(d1 [,d2...])]	Section 13.4.3
EQUIVALENCE (nlist) [,(nlist)...]	Section 7.5
EXTERNAL proc [,proc]....	Section 7.6
FIND (UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios]) FIND (un'rn[,ERR=s][,IOSTAT=ios])	Section 11.8.1
FORMAT (fs)	Section 12.1.1
fun ([arg1,arg2...argn])	Section 13.3.4
[typ] FUNCTION fun ([arg1 [,arg2]...])	Section 13.3.2
GO TO i [[,](s [,s]...)]	Section 9.1.3
GO TO s	Section 9.1.1
GO TO (s [,s]...)[,] e	Section 9.1.2
INCLUDE filespec/switch	Section 6.4.2
IF (e) st	Section 9.2.2
IF (e) s1, s2, s3	Section 9.2.1
IF (e) s1, s2	Section 9.2.3
IF (e) THEN	Section 9.2.4
IMPLICIT type (a [,a...])[,type (a[,a...])... IMPLICIT NONE	Section 7.3
INQUIRE (FILE= fi[,flist]) INQUIRE ([UNIT=] u,ulist)	Section 11.7.1 Section 11.7.2
INTEGER v [,v...]	Section 7.2
INTRINSIC fun[,fun]	Section 7.7
LOGICAL v [,v...]	Section 7.2
NAMELIST /name/list[/name/list]...	Section 12.7
OPEN (openlist)	Section 10.14
PARAMETER (p=c [,p=c...]) PARAMETER p=c[,p=c]	Section 7.8
PAUSE [n]	Section 9.6
PRINT(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] PRINT(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] PRINT f[,iolist] PRINT *[,iolist]	Section 10.10
PROGRAM name	Section 6.4.1
PUNCH(FMT=f[,ERR=s][,IOSTAT=ios])[iolist] PUNCH(FMT=*[,ERR=s][,IOSTAT=ios])[iolist] PUNCH f[,iolist] PUNCH *[,iolist]	Section 10.11

Form	Section Reference
READ(UNIT=un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	Section 10.5
READ(un,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(un, f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(un,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(un, *[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])	
READ(un,FMT=name[,END=s][,ERR=s][,IOSTAT=ios])	
READ(un, name[,END=s][,ERR=s][,IOSTAT=ios])	
READ f[,iolist]	
READ *[,iolist]	
READ(UNIT=*,FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=*,FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(un[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]	
READ(un,FMT=f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]	
READ(un, f,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]	
READ(un'rn,FMT=f [,ERR=s][,IOSTAT=ios])[iolist]	
READ(un'rn, f [,ERR=s][,IOSTAT=ios])[iolist]	
READ(UNIT=un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]	
READ(un,REC=rn[,ERR=s][,IOSTAT=ios])[iolist]	
READ(un'rn [,ERR=s][,IOSTAT=ios])[iolist]	
REAL v [,v...]	Section 7.2
REREAD(FMT=f[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	Section 10.7
REREAD(FMT=*[,END=s][,ERR=s][,IOSTAT=ios])[iolist]	
REREAD f[,iolist]	
REREAD *[,iolist]	
RETURN [e]	Section 13.4.4
REWIND un	Section 11.8.2
REWIND (UNIT=un[,ERR=s][,IOSTAT=ios])	
REWIND (un[,ERR=s][,IOSTAT=ios])	
SAVE [a [,a]...]	Section 7.10
SKIPFILE un	Section 11.8.7
SKIPFILE (UNIT=un[,ERR=s][,IOSTAT=ios])	
SKIPFILE (un[,ERR=s][,IOSTAT=ios])	
SKIPRECORD un	Section 11.8.6
SKIPRECORD (UNIT=un[,ERR=s][,IOSTAT=ios])	
SKIPRECORD (un[,ERR=s][,IOSTAT=ios])	
STOP [n]	Section 9.5
SUBROUTINE sub [([d1 [,d2]...])]	Section 13.4.2.1
TYPE(FMT=f[,ERR=s][,IOSTAT=ios])[iolist]	Section 10.9
TYPE(FMT=*[,ERR=s][,IOSTAT=ios])[iolist]	
TYPE f[,iolist]	
TYPE *[,iolist]	
v = e	Section 8.2
UNLOAD un	Section 11.8.3
UNLOAD (UNIT=un[,ERR=s][,IOSTAT=ios])	
UNLOAD (un[,ERR=s][,IOSTAT=ios])	

Form	**Section Reference**
WRITE(UNIT = un,FMT = f[,ERR = s][,IOSTAT = ios])[iolist]	Section 10.6
WRITE(un,FMT = f[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un, f[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = un,FMT = *[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un,FMT = *[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un, *[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = un,FMT = name[,ERR = s][,IOSTAT = ios])	
WRITE(un,FMT = name[,ERR = s][,IOSTAT = ios])	
WRITE(un, name[,ERR = s][,IOSTAT = ios])	
WRITE f[,iolist]	
WRITE*[,iolist]	
WRITE(UNIT = *,FMT = f[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = *,FMT = *[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = un[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = un,FMT = f,REC = rn[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un,FMT = f,REC = rn[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un, f,REC = rn[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un'rn,FMT = f [,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un'rn, f [,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(UNIT = un,REC = rn[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un,REC = rn[,ERR = s][,IOSTAT = ios])[iolist]	
WRITE(un'rn [,ERR = s][,IOSTAT = ios])[iolist]	

APPENDIX B

ASCII-1968 CHARACTER CODE SET

The character code set defined in the X3.4-1968 Version of the American National Standard for Information Interchange (ASCII) is given in this appendix.

ASCII CODE

Control Characters

Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Class[1]	Remarks
0	000	000	NUL		Null, tape feed. Control @ (control shift P[2]).
1	001	001	SOH	CC	Start of heading [SOM, start of message]. Control A.
1	002	002	STX	CC	Start of text [EOA, end of address]. Control B.
0	003	003	ETX	CC	End of text [EOM, end of message]. Control C.
1	004	004	EOT	CC	End of transmission; shuts off TWX machines and disconnects some data sets. Control D.
0	005	005	ENQ	CC	Enquiry [WRU, "Who are you?"]. Triggers identification ("Here is . . .") at remote station if so equipped. Control E.
0	006	006	ACK	CC	Acknowledge [RU, "Are you . . .?"]. Control F.
1	007	007	BEL		Bell (audible or attention signal). Control G.
1	008	010	BS	FE	Backspace. Control H.
0	009	011	HT	FE	Horizontal tabulation. Control I.
0	010	012	LF[3]	FE	Line feed. Control J.
1	011	013	VT[3]	FE	Vertical tabulation. Control K.
0	012	014	FF[3]	FE	Form feed (to top of next page). Control L.
1	013	015	CR	FE	Carriage return (to beginning of line). Control M.
1	014	016	SO		Shift out; change character set or change ribbon color to red. Control N.
0	015	017	SI		Shift in; return to standard character set or color. Control O.
1	016	020	DLE	CC	Data link escape [DC0]. Control P.
0	017	021	DC1		Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	018	022	DC2		Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	019	023	DC3		Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	020	024	DC4		Device control 4 (stop), turns punch or auxiliary off. Control T (TAPE , AUX OFF).
1	021	025	NAK	CC	Negative acknowledge [ERR, error]. Control U.
1	022	026	SYN	CC	Synchronous idle [SYNC]. Control V.
0	023	027	ETB	CC	End of transmission block [LEM, logical end of medium]. Control W.
0	024	030	CAN		Cancel [S_0]. Control X.
1	025	031	EM		End of medium [S_1]. Control Y.
1	026	032	SUB		Substitute [S_2]. Control Z.
0	027	033	ESC		Escape, prefix [S_3]. Control [(control shift K[2]).
1	028	034	FS	IS	File separator [S_4]. Control \ (control shift L[2]).
0	029	035	GS	IS	Group separator [S_5]. Control] (control shift M[2]).
0	030	036	RS	IS	Record separator [S_6]. Control ^ (control shift N[2]).
1	031	037	US	IS	Unit separator [S_7]. Control - (control shift O[2]).

[1] CC communication control, FE format effector, IS information separator.

[2] On LT33, LT35 and similar units.

[3] Includes a carriage return on some equipment, but not on standard DEC units.

ASCII-1968 CHARACTER CODE SET

Graphic Characters

Figures				Upper Case				Lower Case			
Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character[3]
1	032	040	SP	1	064	100	@	0	096	140	`[2]
0	033	041	!	0	065	101	A	1	097	141	a
0	034	042	"	0	066	102	B	1	098	142	b
1	035	043	#	1	067	103	C	0	099	143	c
0	036	044	$	0	068	104	D	1	100	144	d
1	037	045	%	1	069	105	E	0	101	145	e
1	038	046	&	1	070	106	F	0	102	146	f
0	039	047	'	0	071	107	G	1	103	147	g
0	040	050	(	0	072	110	H	1	104	150	h
1	041	051	)	1	073	111	I	0	105	151	i
1	042	052	*	1	074	112	J	0	106	152	j
0	043	053	+	0	075	113	K	1	107	153	k
1	044	054	,	1	076	114	L	0	108	154	l
0	045	055	–	0	077	115	M	1	109	155	m
0	046	056	.	0	078	116	N	1	110	156	n
1	047	057	/	1	079	117	O	0	111	157	o
0	048	060	Ø[1]	0	080	120	P	1	112	160	p
1	049	061	1	1	081	121	Q	0	113	161	q
1	050	062	2	1	082	122	R	0	114	162	r
0	051	063	3	0	083	123	S	1	115	163	s
1	052	064	4	1	084	124	T	0	116	164	t
0	053	065	5	0	085	125	U	1	117	165	u
0	054	066	6	0	086	126	V	1	118	166	v
1	055	067	7	1	087	127	W	0	119	167	w
1	056	070	8	1	088	130	X	0	120	170	x
0	057	071	9	0	089	131	Y	1	121	171	y
0	058	072	:	0	090	132	Z	1	122	172	z
1	059	073	;	1	091	133	[	0	123	173	{
0	060	074	<	0	092	134	\[2]	1	124	174	|
1	061	075	=	1	093	135	]	0	125	175	}[4]
1	062	076	>	1	094	136	^[2]	0	126	176	~[2,5]
0	063	077	?	0	095	137	_	1	127	177	DEL[6]

[1] Zero–slash absent on many units.

[2] Under study by responsible American National Standards Committee for possible change at next revision of ASCII (ca. 1982).

[3] Codes 140-173 first defined in 1965. For a full ASCII character set the operating system accepts codes 140-176 as lower case. For a program requiring a character set that lacks lower case, the operating system translates input codes 140-174 into the corresponding upper case codes (100-134) and translates both 175 and 176 into 033, escape. Early versions of the DECsystem-10 Monitor used 175 as the escape code and translated both 176 and 033 to it.

[4] Unassigned control character (usually ALT MODE) before 1965. Code generated by ALT MODE key on some DEC units, especially earlier ones; on some more recent units, the ALT key generates the standard escape code, 033.

[5] Control character ESC before 1965; code generated by ESC key on some DEC units designed at that time.

[6] Delete, rub out (not part of lower case set).

ASCII-1968 CHARACTER CODE SET

Remarks on Special Graphic Characters

SP Space – normally nonprinting.

! Exclamation point.

" Quotation mark, diaeresis.

# Number sign. £ on some (non-DEC) units.

$ Dollar sign.

% Percent.

& Ampersand.

' Apostrophe, closing single quotation mark, acute accent. ´ in appearance on some DEC units.

(Opening parenthesis.

) Closing parenthesis.

* Asterisk.

+ Plus.

, Comma, cedilla.

– Hyphen, minus.

. Period, decimal point.

/ Slant, slash, solidus.

: Colon.

; Semicolon.

< Less than.

= Equals.

> Greater than.

? Question mark.

@ Commercial at. ` 1965-67, but never on DEC units.

[Opening bracket. Shift K on LT33, LT35 and similar units.

\ Reverse slant. ~ 1965-67, but never on DEC units. Shift L on LT33, LT35 and similar units.

] Closing bracket. Shift M on LT33, LT35 and similar units.

^ Circumflex, upward arrow head. ↑ before 1965, but used until 1972 on DEC units.

_ Underline, underscore. ← before 1965, but used until 1972 on DEC units.

` Grave accent, opening single quotation mark. @ 1965-67, but never on DEC units.

{ Opening brace.

| Vertical line. Control character ACK before 1965; ¬ 1965-67, but never on DEC units; ¦ in appearance 1968-1977, but generally not on DEC units.

} Closing brace. Unassigned control character (usually ALT MODE) before 1965.

~ Overline, tilde, general accent. Control char- after ESC before 1965; | 1965-67, but never on DEC units.

APPENDIX C

COMPILER MESSAGES

The FORTRAN compiler issues two types of messages: warning and fatal error. While compiling the program, if the compiler encounters a situation that does not prevent it from completing the compilation, but does warrant your knowing about, it prints a warning message and continues compilation. If, however, the problem in your program is such that compilation cannot continue, the compiler prints a fatal-error message and stops compilation of the program. Whenever a fatal error is generated a relocatable object module will not be produced.

Compiler messages are printed in the following form:

```
?FTNxxx LINE:n text
or
%FTNxxx LINE:n flag: text
```

where:

? indicates a fatal message
% indicates a warning message
FTN is the FORTRAN mnemonic
xxx is the 3-letter mnemonic for the error message
Line:n is the optional line number where the error occurred
text is the explanation of error
flag: is the prefix for warning messages generated when the compatibility flagger is invoked. This prefix describes the type of incompatibility the message refers to, and can be one of the following:

- Extension to Fortran-77:
- VMS incompatibility:
- Fortran-77 and VMS:

See Section 16.6 for more information on the compatibility flagger.

Square brackets ([ ]) in this appendix signify variables and are not output on the terminal.

Fatal Errors

Each fatal error in the following list is preceded by ?FTN on the user terminal and on listings. They are presented here in alphabetical order.

ABD [symbolname] has already been defined [definition]

The usage given conflicts with current information about the symbol. For example, a symbol defined in an EQUIVALENCE statement cannot be referenced as a subprogram name.

AHE IF at line [number] already has ELSE

AOA Assumed-size arrays only allowed in subprograms

ASA Assumed-size arrays cannot be used

ATL ARRAY [name] too large

The total amount of memory necessary to accommodate this array is greater than 512P.

AWN Array reference [name] has wrong number of subscripts

The array was defined to have more or fewer dimensions than the given reference.

BOV Statement too large to classify

To determine statement type, some portion of the statement must be examined by the compiler before actual semantic and syntactic analysis begins. During this classification, the entire portion of the required statement must fit into the internal statement buffer (large enough for a normal 20-line statement).

This error message is issued when the portion of a given statement required for classification is too large to fit in the buffer. Once FORTRAN-10/20 has classified a statement, there is no explicit restriction on its length.

CER Compiler error in routine [name]

Submit an SPR for any occurrence of this message.

CEL Character entry points must have the same length

CEN Character expression used where numeric expression required

CEX Constant or constant expression required

CFF Cannot find file

The file referenced in an INCLUDE statement was not found.

CFL Reference to character function [name] must have a fixed length

CNE Character and numeric entry points cannot be mixed

CPE Checksum or parity error in [source/listing/object] file [name]

CQL No closing quote in character constant

CSA Can't split string across numeric and character variables

CSF Illegal statement function reference in CALL statement

DDA [symbolname] is duplicate dummy argument

DDN DO loops too deeply nested - reduce nesting

DFC Variable dimension [name] must be scalar, defined as formal or in COMMON

DFD Double [type] name illegal

Duplicate fields were encountered in an INCLUDE file specification.

DIA DO index variable [name] is already active

In any nest of DO loops, a given index variable may not be defined for more than one loop.

DID Cannot initialize a dummy parameter in DATA

DLN Optional data value list not supported

The extended FORTRAN statement form that allows data values to be defined in type specification statements is not supported by FORTRAN-10/20.

DNL Implied DO specification without associated list of variables

DPR Dummy parameter [name] referenced before definition

DSF Argument [name] is same as FUNCTION name

DTI The dimensions of [arrayname] must be of the type integer

DVE Cannot use dummy variable in EQUIVALENCE

ECS [variable] EQUIVALENCE-d to COMMON is illegal in SAVE statement

ECT Attempt to enter [symbolname] into COMMON twice

EDN Expression too deeply nested to compile

EID ENTRY statement illegal inside a block IF or DO loop

EIL Expression illegal in an input list

EIM ENTRY statement illegal in main program

ENF Label [number] must refer to an executable statement, not a FORMAT

ETF Enter failure [number] [filespec]

EXB EQUIVALENCE extends COMMON block [name] backwards

FEE Found [symbol] when expecting either [symbol] or [symbol]

General syntax error message.

FER [file error text]

An error has occured when processing a command file specification.

FID Can't initialize character function name

FNE Label [number] must refer to a FORMAT, not an executable statement

FWE Found [symbol] when expecting [symbol]

IAC Illegal ASCII character [character] in source

IAL Incorrect argument type for library function [name]

IAN Illegal assignment between character and numeric data

IBD Illegal substring bound in DATA statement

IBK Illegal statement in BLOCK DATA subprogram

ICL Illegal character [character] in label field

ICN Illegal combination of character and numeric data

IDN DO loop at line: [number] is illegally nested

You are attempting to terminate a DO loop before terminating one or more loops defined after the given one.

IDS Implicit DO indices may not be subscripted

IDT Illegal or misspelled data type

IDV Implied DO index is not a variable

IED Inconsistent EQUIVALENCE declaration

The given EQUIVALENCE declaration would cause some symbolic name to refer to more than one physical location.

IFD INCLUDEd files must reside on disk

IFE [INCLUDE file error]

This error occured while trying to open the specified INCLUDE file on the DECSYSTEM-20.

IFS Illegal format specifier

IID Non-integer implied DO index

IIP Illegal implicit specification parameter

IIS Incorrect INCLUDE switch

ILF Illegal statement after logical IF

Refer to Section 9.2.2 for restrictions on logical IF object statements.

IND Improper nesting: DO at line [number] has not terminated

INI Improper nesting: IF at line [number] has not terminated

INN INCLUDE statements may not be nested

IOC Illegal operator for character data

IOD Illegal statement used as object of DO

ION Numeric operand of concatenation operator

IOR Substring bound out of range

IQB INQUIRE - both UNIT and FILE keywords were specified

IQN INQUIRE - neither UNIT nor FILE keywords were specified

ISD Illegal subscript expression in DATA statement

Subscript expressions may be formed only with implicit DO indexes and constants combined with +, -, *, or /.

ISN [symbolname] is not [symboltype]

The symbol cannot be used in the attempted manner.

ISS [variable] illegal in SAVE statement

ITL Illegal transfer into loop to label [number]

IUT Program units may not be terminated within INCLUDEd files

IVC Invalid character constant

IVH Invalid hollerith constant

IVP Invalid PPN

IXM Illegal mixed mode arithmetic

Complex and double-precision cannot appear in the same expression.

IXS Illegal [OPEN specifier] specifier

IZM Illegal [datatype] size modifier [number]

KA FORTRAN will not run on a KA

KAS FORTRAN can not compile for a KA

LAD Label [number] already defined at line: [number]

LED Illegal list directed [statement type]

LFA Label arguments illegal in FUNCTION or array reference

LGB Lower bound greater than upper bound for array [name]

LLS Label too large or too small

Labels cannot be 0 or greater than five digits.

LND Label [number] must refer to a [statement], not a declaration

LNI List directed I/O with no I/O list

LTL Too many items in list - reduce number of items

In rare instances, a combination of long lists in a single statement can exhaust the syntax stack.

MCE More than 1 COMMON variable in EQUIVALENCE group

MSP Statement name misspelled

MST [OPEN specifier] must be [integer or array]

MWL Attempt to define multiple RETURN without formal label arguments

NCC Can't store numeric constant in character variable

NCF Not enough core for the file specs. Total K needed = [number]

NEX No exponent after D or E in constant

NFS No filename specified

The INCLUDE statement requires a filename.

NGS Cannot get segment [segment name] - error code [GETSEG error code]

This message means the system is unable to GETSEG one of the compiler segments on the DECsystem-10.

NIF No matching IF

NIO NAMELIST directed I/O with I/O list

NIR Repeat count must be an unsigned integer

NIU Non-integer unit number in I/O statement

NLF Wrong number of arguments for library function [name]

NLS [variable] may not be declared length star

NMD No matching DO

NNA NAMELIST not allowed in ENCODE, DECODE, and REREAD

NNF No statement number on FORMAT

NNN NML= must specify a NAMELIST

NRC Statement not recognized

NUO .NOT. is a unary operator

NWB Numeric variable must be aligned on word boundary

NWD Incorrect use of * or ? in [filespec]

NYI Not yet implemented

OAG Octal or logical argument illegal to generic function

OBO [variable] may only be specified once

OPW OPEN/CLOSE parameter [name] is of wrong type

OUB Only upper bound of last dimension of [arrayname] may be asterisk

PD6 FORTRAN will not run on PDP6

PIC The DO parameters of [index name] must be integer constants

PN4 Project number must be 4 in ppn

This error is for DECsystem-10 file specifications on the DECSYSTEM-20.

PRF Protection failure [number] [filespec]

PTL Program too large

The program unit takes up more than 512P.

QEF Quota exceeded or disk full [number] [filespec]

RDE Rib or directory error [number] [filespec]

RFC [function name] is a recursive function call

RIC Complex constant cannot be used to represent the real or imaginary part of a complex constant

RUS Relational expression illegal as UNIT specifier

SAD Array [name] - signed dimensions may appear only as constant range limits

SMC Size modifier conflict for variable [name]

SNC Substring of non-character variable

SNL [statement name] statements may not be labeled

SOR Subscript out of range

STD Statement [number] is a declaration

TDO [symbol type] type declaration out of order

TFL Too many FORMAT labels specified

UCE User core exceeded at location [address] in phase [segment] while processing statement [number]

UEC Label [number] previously used in executable context

UFC Label [number] previously used in FORMAT context

UKW Unrecognized keyword

UMP Unmatched parentheses

UNS UNIT may not be specified

USI [symbol type] [symbol name] used incorrectly

The given symbol cannot be used in this way.

VNA Subscripted variable in EQUIVALENCE, but not an array

VSE EQUIVALENCE subscripts must be integer constants

VSO Variable dimension allowed in subprograms only

WIF [I/O type] is illegal with internal files

ZLD Zero-trip DO loop illegal in DATA statement

Warning Messages

Each warning message in the following list is preceded by %FTN on the user terminal and on listings. They are presented here in alphabetical order.

ACB Argument out of range of CHAR, high order bits ignored

ADS Variable [name] already declared in SAVE statement

AGA Opt - object variable, of assigned GOTO without optional list, was never assigned

AIL Illegal length argument for ICHAR, first character used

AIS Extension to Fortran-77: Apostrophe in I/O specifier

ANS VMS incompatibility: ASSOCIATEVARIABLE not set by VMS on OPEN

CAI COMPLEX expression used in arithmetic IF

CAO Consecutive arithmetic operators illegal

CAP Extension to Fortran-77: Consecutive arithmetic operators

CCC Fortran-77 and VMS: Carriage control character

CCN CHARACTER constant used where numeric expression required

CIS Conflicting INCLUDE switches

CNM Character and numeric variables mixed

CNS Extension to Fortran-77: Concatenation with variable of non-specific length

COS Extension to Fortran-77: Comment on statement

COV Extension to Fortran-77: Assigned variable appears in character expression

CSM Extension to Fortran-77: Comma field separator is missing

CTR Complex terms used in a relational other than EQ or NE

The result of the other relational operators with complex operands is undefined.

CUO Constant underflow or overflow

This message is issued when overflow or underflow is detected as the result of building constants or evaluating constant expressions at compile time.

DEB Extension to Fortran-77: DEBUG lines

DFN VMS incompatibility: Default file name on VMS differs from Fortran-10/20

DII Previous declaration of intrinsic function is ignored

DIM Possible DO index modification inside loop

A program that does this may be incorrectly compiled by the optimizer, since it assumes that indexes are never modified. Note that the number of iterations is calculated at the beginning of the loop and is never affected by modification of the index within the loop.

DIS Opt - program is disconnected - optimization discontinued

Submit an SPR if this message occurs.

DOW Extension to Fortran-77: DO WHILE statement

DPE VMS incompatibility: Different precedence in exponentiation

DWE Fortran-77 and VMS: Default widths with edit descriptor [descriptor]

DWL Extension to Fortran-77: DO without statement label

DXB DATA statement exceeds bounds of array [name]

EDD Extension to Fortran-77: END DO statement

EDS Extension to Fortran-77: DECODE statement

EDS Extension to Fortran-77: ENCODE statement

EDX Fortran-77 and VMS: FORMAT edit descriptor [descriptor]

EOC Fortran-77 and VMS: Exponential operator ^

EXD Extension to Fortran-77: Transfer of control into DO loop at label [label]

FAR Extension to Fortran-77: Format in numeric array

FIF Extension to Fortran-77: [function name] is not an intrinsic function in Fortran-77

FIN Extension to Fortran-77: FIND statement

FMR Multiple RETURNs defined in a FUNCTION

FMT VMS incompatibility: Keyword FMT instead of NML

FNA A function without an argument list

FNG Extension to Fortran-77: [function name] is not a generic function in Fortran-77

FNS Extension to Fortran-77: [name] is not a FORTRAN-77 subroutine

FOO Statement function declared out of order or array not dimensioned

HCP VMS incompatibility: Hollerith constant padded with spaces

HCN Hollerith constant used where numeric expression required

HCU Extension to Fortran-77: Hollerith constant

IAT Illegal type for argument [number] for statement function

ICC Illegal character, continuation field of initial line

Continuation lines cannot follow comment lines.

ICD Inaccessible code. Statement deleted

The optimizer will delete statements that cannot be reached during execution.

ICS Illegal character in line sequence number

IDN Opt - illegal DO nesting - optimization discontinued

A GO TO within a DO loop goes to the ending statement of an inner, nested DO loop. The line number printed out with the warning message is that of the OUTER DO.

```
     DO        20
     .
     .
     .
     GO TO     10
     .
     .
     .
     DO        10
     .
     .
     .
10   CONTINUE
     .
     .
     .
20   CONTINUE
```

IFL Opt - infinite loop. Optimization discontinued

IMN IMPLICIT NONE

INC Extension to Fortran-77: INCLUDE statement

INS VMS incompatibility: /NOCREF switch

INS VMS incompatibility: /CREF switch

INS VMS incompatibility: Default for VMS is /NOLIST

IUA	Illegal use of an array - use scalar variable instead
KIS	Obsolete switch /KI
KWU	Fortran-77 and VMS: Keyword [keyword name]
KWV	Fortran-77 and VMS: Keyword value for [keyword name]
LID	Identifier [name] more than six characters The remaining characters are ignored.
LNC	Fortran-77 and VMS: Non-numeric operand in numeric context
LNE	VMS incompatibility: Logical and numeric variables EQUIVALENCE-d
LOL	VMS incompatibility: List of labels
LSP	Extension to Fortran-77: [data type] length specifier
MBD	IMPLICIT NONE - [variable] must be explicitly declared
MLN	Fortran-77 and VMS: Mixing logical and numeric
MSL	Fortran-77 and VMS: Multi-statement line
MVC	Number of variables [is less than/is greater than] the numbers of constants in DATA statement
NAM	Extension to Fortran-77: NAMELIST statement
NDP	Fortran-77 and VMS: No decimal places with edit descriptor
NEC	Extension to Fortran-77: Numeric expression in character context
NED	No END statement in program
NIB	Extension to Fortran-77: Non-integer substring bounds
NIG	Extension to Fortran-77: Non-integer as index to computed GOTO
NIK	Extension to Fortran-77: Non-integer used with [keyword]
NIS	Extension to Fortran-77: Non-integer subscript
NIX	Extension to Fortran-77: Non-integer as index to RETURN
NLC	Fortran-77 and VMS: Non-logical operand in logical context
NLK	Extension to Fortran-77: Use of NAMELIST
NOD	Global optimization not supported with /DEBUG - /OPT ignored
NOF	No output file given
NPC	VMS incompatibility: Null padding before [symbolic name]
NPP	Extension to Fortran-77: No parentheses around PARAMETER list
NSC	Fortran-77 and VMS: Non-standard character in column 1

OCU	Fortran-77 and VMS: Octal constant
OHC	Octal or hexadecimal constant
OIO	Extension to Fortran-77: [statement name] statement
OIO	Fortran-77 and VMS: [statement name] statement
PAV	PARAMETER used as associative variable
PLP	PARAMETER list must be enclosed in parentheses
PPS	PROGRAM statement parameters ignored Used for compatibility purposes.
PSR	Pound sign (#) in random access - use REC= or apostrophe (') instead
PWS	Fortran-77 and VMS: PRINT (Specifiers) statement
RDI	Attempt to redeclare implicit type
RIM	RETURN statement illegal in main program
RLC	Extension to Fortran-77: & used with return label
RLC	Fortran-77 and VMS: $ used with return label
RLX	Fortran-77 and VMS: Return label [label]
SBC	Extension to Fortran-77: Substring bounds not constant
SEP	VMS incompatibility: [symbolic name] is the same as program name or entry point
SID	Slash (/) in dimension bound - use colon (:) instead
SMD	Extension to Fortran-77: Single subscript with multi-dimensioned array [array name]
SNN	VMS incompatibility: [symbolic name] is the same as NAMELIST name
SOD	[name] statement out of order
SOR	Fortran-77 and VMS: Subscript out of range for array [array name]
SPN	VMS incompatibility: [symbolic name] is the same as PARAMETER name
SRO	Fortran-77 and VMS: Symbolic relational operator [operator]
SVN	VMS incompatibility: [symbolic name] is same as variable or function name
TLF	Fortran-77 and VMS: Two-branch logical IF
TSI	Type of symbolic constant ignored
VAI	[name] already initialized

VFS VMS incompatibility: [function name] is a Fortran-supplied routine on VMS

VGF VMS incompatibility: [function name] is a generic function on VMS

VIF VMS incompatibility: [function name] is an intrinsic function on VMS

VND FUNCTION return value is never defined

VNF VMS incompatibility: [function name] is not an intrinsic function on VMS

VNG VMS incompatibility: [function name] is not a generic function on VMS

VNI Opt - variable [name] is not initialized

The optimizer analysis determined that the given variable was never initialized prior to its use in a calculation.

VNS VMS incompatibility: [subroutine name] is not a VMS-supplied subroutine

VSD VMS incompatibility: Subroutine [subroutine name] may differ

WDU Fortran-77 and VMS: WRITE with default unit

WNA Wrong number of arguments for statement function

WOP Opt - warnings given in Phase 1. Optimized code may be incorrect

One or more of the messages issued prior to this message resulted from situations that violate assumptions made by the optimizer, and thus may cause it to generate code that does not execute as desired.

XCR Extraneous carriage return

Carriage return was not immediately preceded or followed by a line termination character.

XEN Fortran-77 and VMS: [* or &] with external name

XOR Extension to Fortran-77: Logical .XOR. operator

ZMT Size modifier [number] treated as [data type]

This message is issued when one of the data type size modifiers that is accepted only for compatibility is used.

Internal Compiler Errors

An internal compiler error is an attempt by either the compiler or the monitor to document an error inside the FORTRAN compiler. An occurrence of an internal compiler error signifies that something is wrong with the FORTRAN compiler.

Monitor-detected internal errors are of the form:

```
?  Internal compiler error
?  [message] at location [address] in phase [segment]
?  While processing statement [line-number]
```

where [message] can be one of the following for TOPS-10:

```
Illegal memory reference
     A read or write was attempted to a non-existent page

Stack exhausted
     Monitor detected PDL overflow

Memory protection violation
     Illegal reference to high segment
```

or where [message] can be one of the following for TOPS-20:

```
Illegal memory reference
     A read was attempted to a non-existent page

Non-existent memory write
     A write was attempted to a non-existent page

Illegal memory read
     A memory read failed

Illegal memory write
     A memory write failed

Stack exhausted
     Monitor detected PDL overflow
```

Compiler-detected errors are of the form:

```
? Internal compiler error-processing statement [line-number]
? Call to [routine-name] from [address]
```

Submit an SPR if you receive an internal compiler error.

At the end of program unit compilation, the compiler prints an error summary line, which is one of the following:

```
[ No error detected ]
%FTNWRN   no warning(s)
%FTNWRN   [warning count] warnings(s)
?FTNFTL   [fatal count] fatal error(s) and no warning(s)
?FTNFTL   [fatal count] fatal error(s) and [warning count]
warning(s)
```

APPENDIX D

FOROTS ERROR MESSAGES

Errors detected at run time by FOROTS fall into the following categories:

1. System errors (SYS) - errors internal to FOROTS
2. Open errors (OPN) - I/O errors that occur during a file OPEN and CLOSE
3. Arithmetic fault errors (APR) - errors in numeric calculations
4. Library errors (LIB) - errors generated by FORLIB library routines
5. Data errors (DAT) - errors in data conversion on I/O
6. Device errors (DEV) - I/O hardware errors
7. Compatibility errors (COM) - errors generated by the compatibility flagger

The messages generated by FOROTS contain the following elements:

1. A 3-letter code that identifies the type of message (TOPS-10 only)
2. The message itself, which describes what FOROTS has encountered
3. For I/O errors, two integer values which are retrieved by the ERRSNS subroutine
4. For compatibility errors, a prefix precedes the message that describes the type of incompatibility the messages refers to; one of the following:

   - Extenstion to FORTRAN-77:
   - VMS incompatibility:
   - FORTRAN-77 and VMS:

   See Section 16.6 for more information on the compatibility flagger.

The 3-letter code (TOPS-10 only) and the message are, by default, printed at your terminal when an error occurs; the two ERRSNS values are stored within the arguments you have supplied for the ERRSNS subroutine. If you do not include a call to the ERRSNS subroutine in your program, your program cannot have access to the two ERRSNS values. (For instructions on how to use the ERRSNS subroutine, see Section 13.4.1.15.)

Table D-1 contains a list of all the 3-letter message codes and the ERRSNS values that are generated by FOROTS.

Table D-1: FOROTS Error Codes

1st Value	2nd Value(5)	Code(1)	Meaning
0			No error detected
	0		No error detected
1			Arithmetic trap
	n (7)	IDC (3)	Integer divide check
2			Input Conversion Error
	n (7)	IOV (3)	Integer overflow
3			Input Conversion Error
	n (7)	FOV (3)	Floating overflow
4			Arithmetic trap
	n (7)	FOV (3)	Floating overflow
5			Arithmetic trap
	n (7)	FDC (3)	Floating divide check
6			Arithmetic trap
	n (7)	FUN (3)	Floating underflow
7			Input Conversion Error
	n (7)	FUN (3)	Floating underflow
9			Output Conversion Error
	0	FTS (3)	Output field width too small
21			FORLIB errors and warnings
	104	IDU	DIVERT: illegal to divert to unit
	105	UNO	DIVERT: unit not open
	106	NOF	DIVERT: unit not open for formatted I/O
	107	CWU	DIVERT: Can't write to unit
	108	CLE	Concatenation result longer than expected
	109	ICE	Illegal length character expression
	110	NCS	No character stack allocated

Table D-1: FOROTS Error Codes (Cont'd)

1st Value	2nd Value(5)	Code(1)	Meaning
	112	AQS	First argument of SORT must be a quoted string
	113	SSE (3)	Substring range error
	114	SRE (3)	Subscript range error
	115	TMA	Too many arguments in call to SORT
	116	CGP	Can't get pages 600:677 for SORT
	117	CRP (1)	Can't return pages 600:677 after call to SORT
	118	NSS (2)	No free section available for SORT
	119	CFS (2)	Can't find SYS:SORT.EXE
	120	CGS (2)	Can't get SYS:SORT.EXE
	121	CPP (3)	Can't preallocate pages 600:677 for SORT
	122	IPN	Illegal page number
	123	CCS	Not enough memory for creating character stack
	124	ECS	Not enough memory for expanding character stack
	125	ALZ	Argument less than or equal to zero
	126	DMA (2,3)	Must give lower and upper bounds to dump in non-zero sections
22			I/O warnings
	509	ETL (3)	Attempt to WRITE beyond fixed-length record
	532	ARC (3)	Ambiguous repeat count
	583	FVM (3)	Format and variable type do not match
	584	RIF (3)	Reading into FORMAT non-standard
	590	DQW (2,3)	Disk full or quota exceeded - Please EXPUNGE, then type CONTINUE
23			FORLIB bounds check warnings
	113	SSE (3)	Substring range error
	114	SRE (3)	Subscript range error
24			End of file
	-1	EOF	End of file
25			Record or record number error
	302	BBF	Bad format binary file
	510	RNR	Attempt to read a record that has not been written
	512	IRN	Illegal record number
	517	RTL	Record too large - memory full
	536	CBI	Cannot backspace image file with no RECORDSIZE
	536	CSI	Cannot skiprecord image file with no RECORDSIZE

Table D-1: FOROTS Error Codes (Cont.)

1st Value	2nd Value(5)	Code(1)	Meaning
	570	ICD	Non-digit in record delimiter
	572	RSM	Record size different from that specified in OPEN
	573	FCL	Unexpected continuation LSCW found
	576	WBA	Attempt to WRITE beyound variable or array
	577	SLN	Record length negative or zero
26			OPEN/CLOSE warnings
	502	CSF (2,3)	Can't set FORTRAN carriage control attribute
	535	BSI (3)	BLOCKSIZE ignored: device is not a magnetic tape
	541	UOA (3)	Unknown OPEN keyword, ignored
	542	NCK (3)	OPEN-only keyword specified in CLOSE, ignored
	550	CQF (1,3)	Cannot QUEUE file
	595	OGX (1,3)	Galaxy version 2 not supported
28			CLOSE error
	J	CLF (2)	Cannot CLOSE file
	J	RNM (2)	Cannot RENAME file
	250+n	CLS (1)	"Close" FILOP. error n (4)
	250+n	DEL (1)	"Delete" FILOP. error n (4)
	250+n	RNM (1)	"Rename" FILOP. error n (4)
30			OPEN error
	J	APP (2)	Cannot set up to append to magnetic tape file
	240	FRR	/RECORDTYPE:FIXED requires /RECORDSIZE
	240	RR1	Random I/O requires RECORDSIZE specifier in OPEN statement
	240	RRR	Random I/O requires /RECORDSIZE
	242	NFC (1)	Too many OPEN units
	243	CIR	/CARRIAGECONTROL:TRANSLATED illegal with this /RECORDTYPE
	244	RLB	/RECORDSIZE larger than /BLOCKSIZE
	245	NSD	No such device
	248	IAC	Specified ACCESS illegal for this device
	249	IDM	Specified MODE is illegal for this device
	250+n	OPN (1)	Cannot OPEN file
	405	PPN (2)	JSYS error - PPN cannot be translated
	503	CEF (2)	End of command file encountered

Table D-1: FOROTS Error Codes (Cont.)

1st Value	2nd Value(5)	Code(1)	Meaning
	506	ICA	Incompatible attributes
	540	SDO	Same device open on another unit with conflicting specifiers
	569	TFM	Tape format conflicts with OPEN statement or default
	585	IAV	Illegal value for OPEN specifier
31			Mixed ACCESS modes
	315	CDR	Can't do direct I/O to sequential file
	315	CDS	Can't do sequential I/O to direct file
	593	POI	Illegal for DIRECT (RANDOM) files
	594	CDF	Can't determine whether formatted or unformatted
32			Illegal logical unit number
	239	IUN	Illegal unit number
33			Compatibility error
	321	CFC (3)	FORTRAN-77 and VMS: Carriage control character
	322	CFF (3)	VMS incompatibility: Intrinsic routine invoked incompatibly
	323	CFR (2,3)	FORTRAN-77 extension: FORTRAN-20 supplied routine invoked
	323	CFX (2,3)	FORTRAN-77 and VMS: FORTRAN-20 supplied routine invoked
	323	CFR (1,3)	FORTRAN-77 extension: FORTRAN-10 supplied routine invoked
	323	CFX (1,3)	FORTRAN-77 and VMS: FORTRAN-10 supplied routine invoked
	324	CFK (3)	FORTRAN-77 and VMS: Keyword [keyword]
	325	CFT (3)	FORTRAN-77 and VMS: Trailing spaces in output record
	326	CFO (3)	FORTRAN-77 extension: Overlap of character assignments
	327	CFG (3)	FORTRAN-77: and VMS: G format descriptor used with character
39			REREAD error
	310	RBR	REREAD not proceeded by READ

Table D-1: FOROTS Error Codes (Cont.)

1st Value	2nd Value(5)	Code(1)	Meaning
45			OPEN/CLOSE statement syntax errors
	241	ESV	Unknown or ambiguous keyword
	241	USW (1)	Unknown switch
	241	ASW (1)	Ambiguous switch
	533	DLT	Dialog string too long
	539	EDS/EDA (2)	Error parsing DIALOG string
	544	NDI (1)	No device specified with ":"
	545	IPP (1)	Illegal PPN
	546	TMF (1)	Too many SFDs
	547	NSI (1)	Null SFD
	548	IDD (1)	Illegal character in DIALOG string
	551	NQS (1)	PADCHAR must be single character in double quotes
47			WRITE on READ-only file
	263	CDT	Cannot WRITE to READ-only file
	554	CWL	Cannot write a file with MODE=LINED
62			Syntax error in FORMAT
	301	ILF	Illegal character in FORMAT
	306	DLF	Data in I/O list but not in FORMAT
	524	RIC	Reading into character format illegal
	538	IRC	Illegal repeat count
	552	IHC	Illegal Hollerith constant
	553	IFW	Illegal field width
	575	UDT	Undefined data type or internal FOROTS error
	583	FVF	Format and variable type do not match
64			Input conversion error
	307	ILC	Illegal character in data
81			FOROTS calling errors
	501	UNS	Unit not specified
	508	IOL	Bad I/O list
	574	IMV	Illegal MTOP value
	579	IDI (1)	Illegal DUMP mode I/O list
	581	DLL (1)	Dump mode I/O list too long
	582	IWI	Illegal to initiate another I/O statement
	599	ICE	Illegal length for character expression

Table D-1: FOROTS Error Codes (Cont.)

1st Value	2nd Value(5)	Code(1)	Meaning
96			Error in magnetic tape operations
	J	ILM (2)	Unexpected MTOPR% error (2)
	J	UMO (2,3)	Error trying to set tape
	530	UTE (1)	Unexpected TAPOP. error
	537	UTO (1)	Unexpected TAPOP. error trying to set parameters
	537	UTO (1,3)	Unexpected TAPOP. error trying to set parameters
	587	ITE (1)	Tape is not usable by your job
97			NAMELIST data errors
	309	VNN	Variable not in namelist
	513	NEQ	"=" not found in namelist data
	514	NRP	Missing right paren
	515	ILN	Variable or namelist does not start with letter
	516	ILS	Illegal subscript
	519	CCC	Cannot convert constant to correct type
	521	RPE	Illegal repeat count
	522	SNV	Sign with null value
	580	NLS	Null string illegal
	596	NEC	Found character when expecting ":"
	597	ISS	Substring descriptor illegal
	598	SNQ	String not within single quotes
98			Unclassifiable device errors
	J	ISW (2)	Cannot switch to input
	J	IOE (2)	General purpose I/O error
	J	OSW (2)	Cannot switch to output
	250+n	ISW (1)	Cannot switch to input
	250+n	OSW (1)	Cannot switch to output
	400	IOE (1)	General-purpose I/O error
	586	MFU	Memory full
	590	DQE (2)	Disk full or quota exceeded
			No ERRSNS values
		BLK (3,6)	Blocks allocated but not deallocated
		CCP (1,6)	Cannot create page
		CDP (1,6)	Cannot destroy page
		CGD (6)	Can't get DBMS
		DBM (6)	DBMS not loaded
		DST (1,6)	Error in dialog string

Table D-1: FOROTS Error Codes (Cont.)

1st Value	2nd Value(5)	Code(1)	Meaning
		EFS (6)	Enter correct file specs
		FFX (6)	FOROP function code exceeds range
		IEM (6)	Error in memory management
		IJE (2,6)	"Impossible" JSYS error
		MFU (6)	Memory full
		NOR (3,6)	Error number out of range
		PAG (3,6)	Pages allocated but not deallocated
		PGD (6)	Deallocating more pages than allocated
		POV (6)	PDL overflow
		SNH (6)	Internal FOROTS error
		TDT (6)	Trap occured during trap processing

(1) TOPS-10 only

(2) TOPS-20 only

(3) This is a warning, not an error. The error cannot be trapped with an ERR= branch, but IOSTAT and ERRSNS will be set.

(4) See the TOPS-10 Monitor Calls Manual for the list of FILOP. error codes and their meanings.

(5) "J" means the TOPS-20 JSYS error code. This number will be between 600000 and 610000 (octal).

(6) No ERRSNS values

(7) Where n is the number of times the error occurs

APR and LIB errors are usually reported as warnings, and the program continues. The number of APR and LIB errors listed on the user's terminal can be changed by the FORTRAN Library Subroutine ERRSET (see Section 13.4.1.14). The I/O errors either cause messages to be printed on the terminal or can be trapped by an error exit argument (ERR=statement number) on OPEN, READ, WRITE, and CLOSE.

The FORTRAN Library Subroutine ERRSNS (see Section 13.4.1.15) allows you to find out which I/O error occurred. When called, ERRSNS returns one or two integer values that describe the status of the last I/O operation performed by FOROTS. (The second integer value is optional.) For example,

```
     CALL ERRSNS (I,J)
```

calls this subroutine. J, the second integer value, is optional.

D.1 ALPHABETICAL DESCRIPTION OF FOROTS MESSAGES

This section contains alphabetical descriptions of each warning and fatal error message that is generated by FOROTS during program execution. Each message is first listed; then is followed by a brief description of how it is generated; followed by, in some cases, a recovery procedure; and finally, where applicable, followed by the ERRSNS values associated with the message.

ALZ ? Argument less than or equal to zero

Cause: An argument (such as a memory size) was specified with a value less than or equal to zero.

Recovery: Specify the correct value for the argument.

ERRSNS values: First Value = 21 Second Value = 125

APP ? Can't setup to append to magtape file <JSYS error> (TOPS-20 only)

Cause: The MTOPR% JSYS failed trying to position the magnetic tape with the function .MOFWF or .MOBKR.

Recovery: Use the information provided to determine a recovery.

ERRSNS values: First Value = 30 Second Value = JSYS error number

AQS ? First argument to SORT must be a quoted string

Cause: The SORT routine detected that the first argument was of type CHARACTER, but the string was not word aligned.

Recovery: If the first argument to the SORT program is of type CHARACTER, it must be word aligned and terminated by an ASCII null. The most reliable way to generate such an argument is to use a character constant.

ERRSNS values: First Value = 21 Second Value = 112

ARC % Ambiguous repeat count

Cause: In a FORMAT statement, a number between two format specifiers can be considered belonging to either one.

Recovery: Insert a comma before or after the number, depending on which specifier the number belongs with.

ERRSNS values: First Value = 22 Second Value = 532

ASW ? Ambiguous switch /<sw> (TOPS-10 only)

Cause: In dialog mode, a switch was specified, but not enough of the switch was given to uniquely specify which switch was intended.

Recovery: Retype the line, completely specifying which switch you mean.

ERRSNS values: First Value = 45 Second Value = 241

BBF ? Bad format binary file

Cause: The control information stored in a binary file is incorrect. The file cannot be read using MODE='BINARY'. This error can be caused when the file you are reading was not written by FORTRAN using MODE='BINARY'.

Recovery: Make sure that you are using the correct file and data mode. Any file can be read with MODE='IMAGE'. (See Section 11.3.19)

ERRSNS values: First Value = 25 Second Value = 302

BLK % Pages allocated but not deallocated

Cause: Internal FOROTS error in memory management.

Recovery: Submit an SPR and include your program.

BSI % Blocksize ignored: device is not a magnetic tape

Cause: A BLOCKSIZE specifier was given in an OPEN statement (see Section 11.3.4) but was not used because the device being opened is not a magnetic tape.

ERRSNS values: First Value = 26 Second Value = 535

CBI ? Can't BACKSPACE IMAGE file with no RECORDSIZE

Cause: An OPEN statement with the MODE='IMAGE' specifier (but not the RECORDSIZE= specifier) was executed prior to a BACKSPACE statement that referred to the open unit.

Recovery: If you are using fixed-length records, specify the RECORDSIZE parameter in the OPEN statement (see Section 11.3.27). Otherwise, the BACKSPACE cannot be done.

ERRSNS values: First Value = 25 Second Value = 536

CCC ? Can't convert constant to correct type

Cause: In NAMELIST input, a variable was assigned a value that does not match. For example, if C is a complex variable, the input:

```
C=.TRUE.
```

is in error, since .TRUE. is not a legal complex number.

Recovery: Correct the error in the source program.

ERRSNS values: First Value = 97 Second Value = 519

CCP ? Can't create page <n> (PAGE. error <n>) (TOPS-10 only)

Cause: FOROTS attempted to use a page of memory for some task, but was unable to. The monitor error code gives the reason. This can be caused by erroneous MACRO subroutines. If no such cause is found, it is an internal FOROTS error.

CCS ? Not enough memory for creating character stack

Cause: A character stack was requested that was larger than 36 sections (larger that the maximum virtual memory available).

Recovery: Specify correct call argument.

ERRSNS values: First Value = 21 Second Value = 124

CDF ? Can't determine whether formatted or unformatted

Cause: The specified file has had both formatted and unformatted I/O operations (or OPENs) performed on it.

Recovery: Use I/O operations and OPENs with the same FORM= specifier.

ERRSNS values: First Value = 31 Second Value = 594

CDP ? Can't destroy page <n> (PAGE. error <n>) (TOPS-10 only)

Cause: FOROTS attempted to use a page of memory for some task, but was unable to. The monitor error code gives the reason. This can be caused by erroneous MACRO subroutines. If no such cause is found, it is an internal FOROTS error.

CDR ? Can't do direct I/O to sequential file

Cause: An attempt was made to perform I/O to a file that is already open in a conflicting mode.

Recovery: Open file in the appropriate mode.

ERRSNS values: First Value = 31 Second Value = 315

CDS ? Can't do sequentail I/O to direct file

Cause: An attempt was made to perform I/O to a file that is already open in a conflicting mode.

Recovery: Open file is the appropriate mode.

ERRSNS values: First Value = 31 Second Value = 315

CDT ? Can't <read/write> an <input/output>-only file

Cause: An attempt was made to perform I/O to a file, but the file is not open for I/O in the appropriate direction.

Recovery: Open the file with ACCESS='SEQINOUT' or 'RANDOM', as appropriate. It is not possible to open a file for APPEND access and then read from it. (See Section 11.3.1.)

ERRSNS values: First Value = 47 Second Value = 263

CEF ? End of command file encountered (TOPS-20 only)

Cause: An indirect file was specified as a DIALOG argument, and the end of the file was encountered before a terminator character (line-feed).

Recovery: Edit the file and insert a line-feed.

ERRSNS values: First Value = 30 Second Value = 503

CFC % Fortran-77 and VMS: Carriage control character

Cause: A carriage-control character was used that is incompatible with ANSI FORTRAN and VAX FORTRAN.

Recovery: If you want the program to be compatible with ANSI FORTRAN or VAX FORTRAN, use a compatible carriage-control character.

ERRSNS values: First Value = 33 Second Value = 321

CFF % VMS incompatibility: Intrinsic routine invoked incompatibly

Cause: An intrinsic routine was invoked in a method incompatible with VAX FORTRAN (such as use of an EXTERNAL statement for an intrinsic function).

Recovery: If you want the program to be compatible with VAX FORTRAN, change to a method of invoking intrinsic routines that is compatible with VAX.

ERRSNS values: First Value = 33 Second Value = 322

CFG % Fortran-77 and VMS: G format descriptor used with character

Cause: The G format descriptor was used with character data, which is an extension to ANSI FORTRAN and VAX FORTRAN.

Recovery: If you want the program compatible with ANSI FORTRAN or VAX FORTRAN, do not use the G format descriptor to edit character data.

ERRSNS values: First Value = 33 Second Value = 327

CFK % Fortran-77 and VMS: Keyword [keyword]

Cause: An OPEN or CLOSE keyword was used that is incompatible with ANSI FORTRAN and VAX FORTRAN.

Recovery: If you want the program compatible with ANSI FORTRAN or VAX FORTRAN, use a compatible OPEN or CLOSE keyword.

ERRSNS values: First Value = 33 Second Value = 324

CFO % Fortran-77 extension: Overlap of character assignments

Cause: A character assignment statement was used in which the character positions defined in the character variable, array element or substring on the left of the equal sign are referenced in the character expression on the right of the equal sign. This is incompatible with ANSI FORTRAN.

Recovery: If you want the program to be compatible with ANSI FORTRAN, use a character assignment statement that does not overlap the character expression and the character variable, array element, or substring.

ERRSNS values: First Value = 33 Second Value = 326

CFR % Fortran-77 extension: FORTRAN-20 supplied routine invoked (TOPS-20 only)
CFR % Fortran-77 extension: FORTRAN-10 supplied routine invoked (TOPS-10 only)

Cause: A FORTRAN-10/20-supplied subroutine was invoked that is not available with ANSI FORTRAN.

Recovery: If you want your program to be compatible to ANSI FORTRAN, use a compatible subroutine.

ERRSNS values: First Value = 33 Second Value = 323

CFS ? Can't find SYS:SORT.EXE - <JSYS error> (TOPS-20 only)

Cause: The file SORT.EXE cannot be found on SYS:. A monitor supplied error message will give more detail.

Recovery: Use the information provided by the monitor to determine the proper course of action.

ERRSNS values: First Value = 21 Second Value = 119

CFT % Fortran-77 and VMS: Trailing spaces in output record

Cause: Your program contains a FORMAT that specifies trailing blanks (X format and $ format). In this case, FORTRAN-10/20 preserves the trailing spaces.

Recovery: If you want the program compatible with ANSI FORTRAN and VAX FORTRAN, do not use this form of the FORMAT statement.

ERRSNS values: First Value = 33 Second Value = 325

CFX % Fortran-77 and VMS: FORTRAN-20 supplied routine invoked (TOPS-20 only)
CFX % Fortan-77 and VMS: FORTRAN-10 supplied routine invoked (TOPS-10 only)

Cause: A FORTRAN-10/20-supplied subroutine was invoked that is not available with ANSI FORTRAN or VAX FORTRAN.

Recovery: If you want the program to be compatible with ANSI FORTRAN or VAX FORTRAN, use a compatible subroutine.

ERRSNS values: First Value = 33 Second Value = 323

CGP ? Can't get pages 600:677 for SORT

Cause: The SORT subroutine brings the SORT program into core in pages 600 through 677. Some of these pages were already occupied by programs or data at the time that SORT was called.

Recovery: Decrease the size of your program. Having fewer files open or using a BUFFERCOUNT=1 specifier in OPEN statement may help (see Section 11.3.5). If this does not help, you can segment the program by using LINK's overlay facility (see the LINK Reference Manual).

ERRSNS values: First Value = 21 Second Value = 116

CGS ? Can't get SYS:SORT.EXE - <JSYS error> (TOPS-20 only)

Cause: The file SORT.EXE was found on SYS:, however for some reason it could not be merged into your program in order to sort files. A monitor supplied error message will give more detail.

Recovery: Use the information provided by the monitor to determine the proper course of action.

ERRSNS values: First Value = 21 Second Value = 120

CIR ? /CARRIAGECONTROL:TRANSLATED illegal with this /RECORDTYPE

Cause:

Recovery:

ERRSNS values: First Value = 30 Second Value = 243

CLE ? Concatenation result larger than expected

Cause: The specified substring bounds are out of range.

Recovery: Specify legal substring bounds.

ERRSNS values: First Value = 21 Second Value = 108

CLS ? CLOSE failed, <I/O error message> (TOPS-10 only)

Cause: A CLOSE UUO. or FILOP. CLOSE function failed.

Recovery: Use the information provided to determine a recovery.

ERRSNS values: First Value = 28 Second Value = 250+n

CPP % Can't preallocate pages 600:677 for SORT

Cause: A call to the SRTINI subroutine was unable to preallocate SORT's pages because they were already allocated.

Recovery: Decrease the size of your program. Having fewer files open or using a BUFFERCOUNT=1 specifier in the OPEN statement may help (see Section 11.3.5).

ERRSNS values: First Value = 21 Second Value = 121

CQF % Can't queue file: QUEUE. UUO ERROR <N> (TOPS-10 only)

Cause: This error may occur when executing a CLOSE statement in which the DISPOSE specifier is given with one of the values: 'LIST', 'PRINT', or 'PUNCH', and GALAXY release 4 is running. (See Section 11.5.4.)

Recovery: Refer to the TOPS-10 Monitor Calls Manual for an explanation of the QUEUE. error number <n>.

ERRSNS values: First Value = 26 Second Value = 550

CRP ? Can't return pages 600:677 after call to SORT (TOPS-10 only)

Cause: Before the SORT subroutine returns to the user, it tries to deallocate the pages that the SORT program resides in. The pages could not be deallocated. This is not expected to occur.

Recovery: Submit an SPR.

ERRSNS values: First Value = 21 Second Value = 117

CSF % Can't set FORTRAN carriage control attribute (TOPS-20 only)

Cause: You are running a TOPS-20 system that does not have Edit 2981, which allows support for the FORTRAN carriage-control attribute.

Recovery: Autopatch the monitor to include Edit 2981.

ERRSNS values: First Value = 26 Second Value = 502

CSI ? Can't skiprecord image file with no RECORDSIZE

Cause: A SKIPRECORD statement was executed for a file opened with MODE=IMAGE, and the file did not have a RECORDSIZE(RECL) specified. Since there are no record markers in IMAGE mode files, FOROTS cannot determine how far to skip.

Recovery: Add a RECL= specifier (see Section 11.3.27) to the OPEN statement.

ERRSNS values: First Value = 25 Second Value = 536

CWL ? Can't write a file with MODE='LINED'

Cause: The program attempted to execute an output operation such as a WRITE statement after an OPEN statement for the same unit. The OPEN statement contained a MODE='LINED' specifier.

Recovery: Change the OPEN statement specifier to MODE='ASCII'. (See Section 11.3.20.)

ERRSNS values: First Value = 47 Second Value = 554

CWU ? DIVERT: Can't write to unit <n>

Cause: DIVERT file is not opened for output.

Recovery: Open DIVERT file for output.

ERRSNS values: First Value = 21 Second Value = 107

DBM ? DBMS not loaded

Cause: A DBMS call to a sharable FOROTS was attempted without DBMS.

Recovery: The system manager must build DBMS into sharable FOROTS or remove the file from FORLIB that directs the DBMS call to the sharable FOROTS.

DEL ? Can't delete file:<FILOP. error message> (TOPS-10 only)

Cause: The "DELETE" FILOP. failed. The file is not deleted.

Recovery: Usually you can correct the problem when the program is finished and delete the file with the monitor DELETE command.

ERRSNS values: First Value = 28 Second Value = 250+n

DLF % Data in IO list but not in format

Cause: An I/O statement has requested data to be transferred, but the FORMAT statement does not specify any descriptor that would translate the data.

Recovery: Fix the FORMAT statement. It must contain one of the following descriptors:

A,B,E,F,G,I,L,O,Q,R,Z

ERRSNS values: First Value = 62 Second Value = 306

DLL ? DUMP mode I/O list too long (TOPS-10 only)

Cause: The I/O list specified for a DUMP mode READ or WRITE statement is too long for the FOROTS internal DUMP mode control list.

Recovery: Split the I/O list across two or more READ or WRITE statements.

ERRSNS values: First Value = 81 Second Value = 581

DMA % Must give lower and upper bounds to dump in non-zero sections (TOPS-20 only)

Cause: A call to PDUMP or DUMP was made without specifying memory bounds. In section zero, this is interpreted as 'all of memory'. For extended addressing, lower and upper bounds must be specified.

Recovery: Specify lower and upper bounds for memory.

ERRSNS values: First Value = 21 Second Value = 126

DQE ? Disk full or quota exceeded

Cause: The disk quota for the disk on which a file is being written is exhausted, or the entire disk structure is full. If this error is encountered while running under batch, the program is aborted and an attempt is made to close all files. If this error is encountered while timesharing, the user is requested to type an EXPUNGE command and then a CONTINUE command.

Recovery: EXPUNGE or create more room on the specified disk structure.

ERRSNS values: First Value = 98 Second Value = 590

DQW % Disk full or quota exceeded - Please EXPUNGE, then type CONTINUE (TOPS-20 only)

Cause: The file or files being written on the disk have either exhausted your disk quota or filled the structure.

Recovery: FOROTS leaves the terminal at EXEC level so that you have more options to use to provide space on the disk structure. You can type CONTINUE to resume processing if you don't reset the current fork.

ERRSNS values: First Value = 22 Second Value = 590

DSS % DISPOSE='SAVE' assumed - device is not a disk

Cause: A DISPOSE value other than 'SAVE' was specified for a file on a device other than disk. (See Section 11.3.13.)

ERRSNS values: First Value = 26 Second Value = 549

DST ? Error in dialog string (TOPS-10 only)

Cause: A syntax error in the DIALOG= specifier.

Recovery: Correct the error in the program or in DIALOG mode. (See Section 11.5.3.)

DTL % Dialog string too long

Cause: The argument to DIALOG= cannot be parsed because it is too long.

Recovery: Use a shorter string.

ERRSNS values: First Value = 45 Second Value = 533

ECS ? Not enough memory for expanding character stack

Cause: More memory than is available on a KL10 was requested by either a dynamic concatenation (concatenation of character variables of length*) or by a call to the ALCCHR subroutine.

Recovery: Reduce the size of your concatenation or argument to ALCCHR. If you are running extended addressing, you can present parts of the character stack by invoking SORT and dynamic libraries at the beginning of the program.

ERRSNS values: First Value = 21 Second Value = 124

EDS/EDA ? Error in DIALOG string - <message> (TOPS-20 only)

Cause: A syntax error in the DIALOG= specifier. (See Section 11.5.3.)

Recovery: Retype the specifier correctly in DIALOG mode.

ERRSNS values: First Value = 45 Second Value = 539

EFS [Enter correct file specs]

Cause: Dialog mode. You should respond to this message by entering any information you wish to change about the indicated file. This can include the device, filename, directory, or any OPEN parameter. The form of a response is:

```
TOPS-10   DEV:FILE.EXT[DIRECTORY] /SWITCH:VAL /SWITCH:VAL
TOPS-20   DEV:<DIRECTORY>FILE.EXT.GEN /SWITCH:VAL /SWITCH:VAL
```

All parts of this specification are optional.

EOF ? End of file

Cause: An input statement has attempted to read more data than the file contains.

Recovery: Use an END= specifier in the READ statement, or lengthen the file.

ERRSNS values: First Value = 24 Second Value = -1

ESV ? <unknown/ambiguous> keyword value /<switch>:<value> (TOPS-20 only)

Cause: A switch entered in dialog mode was not recognized or was not specified uniquely.

Recovery: Retype the line, specifying the correct switch.

ERRSNS values: First Value = 45 Second Value = 241

ETL % Attempt to write beyond fixed-length record

Cause: In an ENCODE statement, the format specified more characters than the string will hold. The excess characters are ignored. (See Section 10.12.)

Recovery: Shorten the format or lengthen the string.

ERRSNS values: First Value = 22 Second Value = 509

FCL ? Found unexpected continuation LSCW

Cause: A RECL(RECORDSIZE) has been specified in an OPEN statement, and FOROTS has encountered a continuation (type 2) logical segment control word (LSCW). This type LSCW is never written in fixed-length binary records.

Recovery: Remove the RECL(RECORDSIZE) specification in the OPEN statement.

ERRSNS values: First Value = 25 Second Value = 573

FDC % Floating divide check

Cause: The program contains a floating-point division in which the divisor is too small compared to the dividend to yield a result that is in the floating-point range.

Recovery: Correct program so that division is within floating-point range.

ERRSNS values: First Value = 5 Second Value = n where n is the number of times the error occurs

FFX ? FOROP. function code exceeds range

Cause: A library routine has called for an operation that is not available. This can be caused by using a mismatched FOROTS and FORLIB, or by an erroneous MACRO subroutine.

Recovery: Make sure the versions of FOROTS.EXE and FORLIB.REL that you are loading from are matching versions. If this doesn't help, find the subroutine causing the problem and alter or remove the erroneous call.

FOV % Floating overflow

Cause: A REAL or DOUBLE PRECISION number was read that is too large in magnitude (see Chapter 3). This is only a warning and does not stop execution of the program. The results of a calculation that overflows are set to the largest representable number with the sign of the correct result.

Recovery: Modify the data so that its values fall in the range of values that can be represented for the data type.

ERRSNS values: First Value = 3 or 4 Second Value = n where n is the number of times the error occurs

FRR ? /RECORDTYPE:FIXED requries /RECORDSIZE

Cause: A RECORDTYPE='FIXED' was specified in an OPEN statement without a RECORDSIZE (RECL) specifier.

Recovery: Specify RECL in the OPEN statement (see Section 11.3.27).

ERRSNS values: First Value = 30 Second Value = 240

FTS % Output field width too small

Cause: The field width specified in a FORMAT statement was not large enough to allow the printing of the value being output. For example, this error would occur if the number 100 is output with the format specifier "I2".

ERRSNS values: First Value = 9 Second Value = 0

FUN % Floating underflow

Cause: A REAL or DOUBLE PRECISION number was read that is too small in magnitude (see Chapter 3). This only a warning and does not stop execution of the program. The result of a calculation that underflows is set to zero.

Recovery: Modify the data so that its values fall in the range of values that can be represented for the data type.

ERRSNS values: First Value = 6 or 7 Second Value = n where n is the number of times the error occurs

FVF ? Format and variable type do not match

Cause: An attempt was made to READ or WRITE character data with other than A or G format.

Recovery: Specify A or G edit descriptors when reading character data.

ERRSNS values: First Value = 62 Second Value = 583

FVM % Format and variable type do not match

Cause: An I/O statement has been executed that uses a format edit descriptor with a type that does not match the I/O list item being processed.

Recovery: Specify the appropriate format edit descriptor for the I/O list item.

ERRSNS values: First Value = 22 Second Value = 583

IAC ? /ACCESS illegal for this device

Cause: An attempt was made to OPEN a device for which the access specified (or implied) is illegal.

Recovery: Change the ACCESS specifier in the OPEN statement or data transfer statement. (See Section 11.3.1.)

ERRSNS values: First Value = 30 Second Value = 248

IAV ? Illegal value for OPEN specifier

Cause: An OPEN statement specifier has a value illegal for that specifier.

Recovery: Specify a legal value for that OPEN specifier.

ERRSNS values: First Value = 30 Second Value = 585

ICA ? Incompatible attributes

Cause: An illegal combination of open attributes has been specified.

Recovery: Change one or more of the conflicting specifiers.

ERRSNS values: First Value = 30 Second Value = 506

ICD ? Non-digit in record delimiter

Cause: The format of the tape being read is not 'D' (DELIMITED). The Record Control Word (RCW) contained a non-digit, or the data on the tape is incompatible.

Recovery: Specify the correct TAPEFORMAT in the OPEN statement.

ERRSNS values: First Value = 25 Second Value = 570

ICE ? Illegal length for character expression

Cause: A program has specified a zero length or negative length character substring as an I/O list element.

Recovery: Fix program to specify a positive length substring.

ERRSNS values: First Value = 81 Second Value = 599

IDC % Integer divide check

Cause: Program contains an integer division by 0.

Recovery: Correct division in program.

ERRSNS values: First Value = 1 Second Value = n where n is the number of times the error occurs

IDD ? Illegal character <chr> (TOPS-10 only)

Cause: An illegal character was encountered in dialog mode.

Recovery: Retype the response without illegal characters.

ERRSNS values: First Value = 45 Second Value = 548

IDI ? Illegal DUMP mode I/O list (TOPS-10 only)

Cause: An I/O list entry has been specified whose entry size (number of words) is different from its increment. This can only happen if an implied DO loop is specified for the I/O list, the index increment is set to a value other than 1, and the program is compiled with /OPTIMIZE.

Recovery: Use an index increment of 1, or do not compile the program with /OPTIMIZE.

ERRSNS values: First Value = 81 Second Value = 579

IDM ? /MODE:<mode> illegal for this device

Cause: Not all devices can do I/O in all modes. For example, terminals cannot do binary I/O.

Recovery: Change the MODE= specifier or the device. (See Section 11.3.19.)

ERRSNS values: First Value = 30 Second Value = 249

IDU ? DIVERT: illegal to divert to unit <n>

Cause: Unit specified is an input-only device.

Recovery: Specify a unit for which output is legal.

ERRSNS values: First Value = 21 Second Value = 104

IEM ? FOROTS internal error in memory management

Cause: This is an internal error that is not expected to occur. It means that the memory management routines have detected a problem with their control information.

Recovery: Submit an SPR.

IFW ? Illegal field width

Cause: An illegal (negative) field width was specified in a FORMAT statement.

Recovery: Specify a legal field width in the FORMAT statement.

ERRSNS values: First Value = 62 Second Value = 553

IHC ? Illegal Hollerith constant

Cause: A format specification contains an H edit descriptor that is not preceded by a length or does not contain enough characters.

Recovery: Use the correct format for an H edit descriptor. (See Section 12.4.2.)

ERRSNS values: First Value = 62 Second Value = 552

IJE ? "Impossible" JSYS error at <PC> - <JSYS ERROR> (TOPS-20 only)

Cause: This is an internal FOROTS error that is not expected to occur. A monitor call failed that was not expected to. A monitor-supplied error message may be of assistance in avoiding the problem.

Recovery: Submit an SPR.

ILC ? Illegal character in data

Cause: A format descriptor that requires a number found a nonnumeric character.

Recovery: Fix the input data or FORMAT statement.

ERRSNS values: First Value = 64 Second Value = 307

ILF ? Illegal character in format

Cause: A format specification contains a character with no defined meaning.

Recovery: Correct the error in the format list and rerun the program.

ERRSNS values: First Value = 62 Second Value = 301

ILM ? Unexpected MTOPR% ERROR (TOPS-20 only)

Cause: An error was encountered during a file operation that FOROTS did not expect.

Recovery: This type of error should not happen. Please submit an SPR.

ERRSNS values: First Value = 96 Second Value = JSYS error number

ILN ? Variable or namelist does not start with letter

Cause: NAMELIST input contains something other than a legal variable or NAMELIST name in a context where a variable or NAMELIST name is required.

Recovery: Correct the source program with a legal variable or NAMELIST name. (See Section 12.6.)

ERRSNS values: First Value = 97 Second Value = 515

ILS ? Illegal subscript

Cause: In NAMELIST I/O, an illegal subscript was given for an array.

ERRSNS values: First Value = 97 Second Value = 516

IMV ? Illegal MTOP value

Cause: A MARCO program has issued an MTOP call to FOROTS with an illegal value for the function.

Recovery: Specify a legal function value in the call.

ERRSNS values: First Value = 81 Second Value = 574

IOE ? <IO error message>

Cause: An I/O error has occurred. The monitor error code is given, along with an interpretation of the probable meaning of the error bits. This message normally indicates that the data recorded on an external device has been damaged and cannot be read correctly.

ERRSNS values: First Value = 98 Second Value = 400 (TOPS-10)

JSYS error number (TOPS-20)

IOL ? Bad format IO list

Cause: The code generated by the compiler for an I/O list is not understood by this version of FOROTS. The erroneous entry in the I/O list is ignored. This probably indicates an internal error in the compiler or in FOROTS.

Recovery: Locate the problem area of the I/O list and simplify it.

ERRSNS values: First Value = 81 Second Value = 508

IOV % Integer overflow

Cause: An attempt was made to read data that was out of range for an integer variable.

ERRSNS values: First Value = 2 Second Value = 0

IPP ? Illegal PPN (TOPS-10 only)

Cause: A directory specification starts with something that is not a legal PPN specification. The forms of legal PPNs are:

[n,n], [n,], [,n], or [,]

where n represents a 1- to 6-digit octal number.

Recovery: Use a legal directory specification.

ERRSNS values: First Value = 45 Second Value = 545

IPN ? Illegal page number <n>

Cause: A call to TOPMEN or SRTINI has specified a page number outside the range 1:777.

Recovery: Specify a correct page number

ERRSNS values: First Value = 21 Second Value = 122

IRC ? Illegal repeat count

Cause: An illegal repeat count was given in a FORMAT statement.

Recovery: Correct the FORMAT statement.

ERRSNS values: First Value = 62 Second Value = 538

IRN ? Illegal record number <n>

Cause: A direct-access I/O statement has specified a record number that is zero or negative.

Recovery: Correct the invalid record number in the program.

ERRSNS values: First Value = 25 Second Value = 512

ISS ? Illegal substring descriptor

Cause: An I/O statement refers to an illegal substring delimiter (substring not within bounds of string).

Recovery: Correct the substring specifier.

ERRSNS values: First Value = 97 Second Value = 597

ISW ? Can't switch to input

Cause: A file that was being written cannot be open for output. The file is either protected against reading, or has been deleted before the OPEN for read is executed.

Recovery: Specify correct protection for OPEN write.

ERRSNS values: First Value = 98 Second Value = 250+n (TOPS-10)

JSYS error number (TOPS-20)

ITE ? Tape is not usable by your job (TOPS-10 only)

Cause: A tape unit was specified or implied that is not owned by your job, and is probably owned by another job.

Recovery: ASSIGN the drive or MOUNT the tape.

ERRSNS values: First Value = 96 Second Value = 587

IUN ? Illegal unit number <n>

Cause: An I/O statement has specified a unit number that is negative or too large.

Recovery: Change the UNIT specifier value and rerun the program.

ERRSNS values: First Value = 32 Second Value = 239

IWI ? Illegal to initiate another I/O statement while processing <I/O statement)

Cause: An I/O statement, STOP statement, or PAUSE statement has been initiated while processing another I/O statement (such as in a function reference used as an I/O list element), or while within a subroutine called as a result of an I/O error through ERRSET.

Recovery: Remove the offending I/O statement, STOP statement, or PAUSE statement.

ERRSNS values: First Value = 81 Second Value = 582

MFU ? Memory full

Cause: There is insufficient memory to complete execution of the program.

Recovery: Some memory can be saved by opening fewer files at a time, by using BUFFERCOUNT=1 in OPEN statements, and by using minimal tape block sizes. If these techniques do not help, you can segment the program using LINK's overlay facility (see the LINK Reference Manual).

NCA ? No memory available for character stack

Cause: For non-overlay programs, this message indicates that the memory manager has allocated all available space between the user's low segment and FOROTS. For overlay programs, this messages indicates that /SPACE:0 has been specified to LINK.

Recovery: For non-overlay programs, LINK with /OTS:NONSHARE. For overlay programs, specify at least 1000 to the /SPACE switch in LINK (see the LINK Reference Manual).

ERRSNS values: First Value = 21 Second Value = 111

NCK % <keyword> in CLOSE is meaningless - ignored

Cause: Options have been included in the CLOSE statement that are meaningless for closing the file.

Recovery: Use valid CLOSE options. (See Section 10.17.)

ERRSNS values: First Value = 26 Second Value = 542

NCS ? No character stack allocated - compiler error

Cause: An internal compiler error has occurred.

Recovery: Submit an SPR.

ERRSNS values: First Value = 21 Second Value = 110

NDI ? No device specified with ":" (TOPS-10 only)

Cause: An OPEN statement has specified a null device name.

ERRSNS values: First Value = 45 Second Value = 544

NEC ? Found "<chr>" when expecting ":"

Cause: Substring parameters not separated by ":"

Recovery: Insert a ":" between substring parameters.

ERRSNS values: First Value = 97 Second Value = 596

NEQ ? Found "<chr>" when expecting "="

Cause: NAMELIST input found an illegal character in a context that requires an equal sign.

Recovery: Replace illegal character with equal sign.

ERRSNS values: First Value = 97 Second Value = 513

NFC ? Too many open units (TOPS-10 only)

Cause: On TOPS-10 monitors before version 7.00, at most, only 16 units can be open at the same time.

Recovery: Arrange the program so that it never needs to have more than 16 simultaneously open units.

ERRSNS values: First Value = 30 Second Value = 242

NLS ? Null string illegal

Cause: An attempt was made to input to a zero length string during list-directed input.

Recovery: Insert characters into the string, or remove the string delimiting quotes.

ERRSNS values: First Value = 97 Second Value = 580

NQS ? PADCHAR must be single char in double quotes (TOPS-10 only)

Cause: In dialog mode, the PADCHAR specifier must be followed by the pad character in double quotes.

ERRSNS values: First Value = 45 Second Value = 551

NRP ? Missing right paren

Cause: In NAMELIST or list-directed complex input, the closing right parenthesis that ends a complex number was not found.

ERRSNS values: First Value = 97 Second Value = 514

NSD ? No such device <dev>

Cause: The specified device does not exist.

Recovery: Change the device name to one that does exist.

ERRSNS values: First Value = 30 Second Value = 245

NSI ? Null SFD (TOPS-10 only)

Cause: A directory specification contains a null SFD.

ERRSNS values: First Value = 45 Second Value = 547

NSS ? No free section available for SORT (TOPS-20 only)

Cause: SORT runs it its own section on machines that support extended addressing. There are no free sections available.

Recovery: There are 31 sections normally available when a simple FORTRAN program runs. If your application is not trying to use extended addressing, this error should not occur, and you should submit an SPR.

ERRSNS values: First Value = 21 Second Value = 118

OGX % Galaxy version 2 not supported (TOPS-10 only)

Cause: Your system is using an unsupported version of GALAXY.

Recovery: Inform the system administrator to upgrade to the supported version of GALAXY.

ERRSNS values: First Value = 26 Second Value = 595

OPN ? Can't OPEN file

Cause: The specified file could not be opened. The reason given is taken from the monitor error code (see the TOPS-10 Monitor Calls Manual).

ERRSNS values: First Value = 30 Second Value = 250+n

OSW ? FILOP. error n - can't switch to output

Cause: An attempt to open a file for write access which has previously been open for read-only access failed.

Recovery: Change protection code; remove other file access.

ERRSNS values: First Value = 98 Second Value = 250+n (TOPS-10)

JSYS error number (TOPS-20)

PAG % Pages allocated but not deallocated

Cause: Internal FOROTS error in memory management.

Recovery: Submit an SPR and include your program.

PGD ? Deallocating more pages than allocated

Cause: Internal FOROTS error in memory management.

Recovery: Submit an SPR and include your program.

POI ? <file positioning operation> Illegal for DIRECT (RANDOM) file

Cause: A file positioning operation (such as REWIND or BACKSPACE) was attempted on a file open for DIRECT(RANDOM) access.

Recovery: Remove the file positioning statement.

ERRSNS values: First Value = 31 Second Value = 593

PPN ? <JSYS error> (TOPS-20 only)

Cause: A TOPS-20 OPEN statement has specified a PPN instead of a directory name, but the PPN has no matching directory.

Recovery: Specify the correct PPN, or better yet, specify the directory name instead.

ERRSNS values: First Value = 30 Second Value = 405

RBR ? REREAD not preceded by READ

Cause: A REREAD statement was encountered before any READ statement. A READ must be executed first so there is something to reread.

Recovery: Cause a READ statement to be executed first.

ERRSNS values: First Value = 39 Second Value = 310

RLB ? /RECORDSIZE larger than /BLOCKSIZE

Cause: A RECORDSIZE was specified in an OPEN statement that is larger than the specified or implied BLOCKSIZE.

Recovery: Correct either RECORDSIZE or BLOCKSIZE, or specify BLOCKSIZE if it is not specified.

ERRSNS values: First Value = 30 Second Value = 244

RIC ? Reading into character format illegal

Cause: An attempt was made to READ into a character format.

Recovery: Correct program to avoid this construct. READ into a character variable and use this variable (perhaps concatenated with other character expressions) for a modifiable format.

ERRSNS values: First Value = 62 Second Value = 524

RIF % Reading into FORMAT nonstandard

Cause: A READ statement was executed that reads data into a Hollerith or quoted string in a FORMAT statement. This is a practice contrary to the ANSI standard and is likely not to work in future releases of FORTRAN-10/20.

Recovery: READ into character variables and use (perhaps concatenated with character constants) as the FORMAT.

ERRSNS values: First Value = 22 Second Value = 584

RNM ? Can't rename file

Cause: An attempt to rename the specified file failed.

Recovery: Change file protection, remove other file access.

ERRSNS values: First Value = 28 Second Value = 250+n (TOPS-10)

JSYS error number (TOPS-20)

RNR ? Record <n> has not been written

Cause: In direct-access input, an attempt was made to read a record that was never written. This may indicate the use of an incorrect record number.

Recovery: Make sure you are requesting the correct record.

ERRSNS values: First Value = 25 Second Value = 510

RPE ? Illegal repeat count

Cause: In NAMELIST or list-directed input, a repeated constant was found, but the repeat count is not a positive integer.

Recovery: Correct the input and try again.

ERRSNS values: First Value = 97 Second Value = 521

RR1 ? Random I/O requires RECORDSIZE specifier in OPEN statement

Cause: Direct-access I/O was attempted to a file that has not been opened with the RECL or RECORDSIZE specifier to give the size of the record(s).

Recovery: Specify a record size. The record size is in characters for formatted files, words for unformatted files. (See Section 11.3.27.)

ERRSNS values: First Value = 30 Second Value = 240

RRR ? Random IO requires /RECORDSIZE

Cause: An OPEN statement was attempted that specified ACCESS=DIRECT(RANDOM) with no record size specified

Recovery: Specify a record size.

ERRSNS values: First Value = 30 Second Value = 240

RSM ? Record size different from that specified

Cause: A record size found in a binary record is different than that specified in the OPEN statement.

Recovery: Specify the correct record size in the OPEN statement.

ERRSNS values: First Value = 25 Second Value = 572

SDO ? Same device open on unit with conflicting specifiers

Cause: An OPEN statement was attempted for a device for which another OPEN or data transfer statement had been already executed, and the file specifications were in conflict.

Recovery: Change file specifications

ERRSNS values: First Value = 30 Second Value = 540

SLN ? Record length negative or zero

Cause: An ENCODE or DECODE statement was initiated that has a negative or zero value for the record (string) length.

Recovery: Correct program to specify a legal record length.

ERRSNS values: First Value = 25 Second Value = 577

SNH ? Internal FOROTS error at <PC>

Cause: This error is not expected to occur. An internal consistency check has found a bug.

Recovery: Please submit an SPR if you get this message.

SNQ ? String not within single quotes

Cause: A character data item read as list-directed or NAMELIST input is not enclosed in single quotes.

Recovery: Enclose character data item in single quotes.

ERRSNS values: First Value = 97 Second Value = 598

SNV ? Sign with null value

Cause: List-directed or NAMELIST input contains a + or - sign not followed by a value.

Recovery: Correct the input and try again.

ERRSNS values: First Value = 97 Second Value = 522

SRE % Subscript range error - subscript <n> of array <name> on line <n>

Cause: An illegal subscript or range has been specified for an array reference.

Recovery: Specify a legal array reference.

ERRSNS values: First Value = 23 Second Value = 114

SSE % Substring range error <var(bound)> on line <n>

Cause: An illegal substring bound or range has been specified in a character expression.

Recovery: Specify a legal reference.

ERRSNS values: First Value = 23 Second Value = 113

TFM ? Tape format conflicts with OPEN statement or default

Cause: The actual format of the tape (either CORE-DUMP or INDUSTRY) conflicts with the format specified in the OPEN statement or by the monitor. At this point, the file is already opened in the wrong format.

Recovery: Specify the correct TAPEFORMAT in the OPEN statement or with the EXEC.

ERRSNS values: First Value = 30 Second Value = 569

TMA ? Too many arguments in call to SORT

Cause: When the first argument in a call to SORT is a character constant, the argument list must be copied in order to convert the argument to a Hollerith constant. At most, 10 arguments can be copied.

Recovery: Change the first argument to a Hollerith constant, or use less than 10 arguments.

ERRSNS values: First Value = 21 Second Value = 115

TMF ? Too many SFDs (TOPS-10 only)

Cause: A directory specification contains more than five SFDs.

Recovery: Specify the correct directory.

ERRSNS values: First Value = 45 Second Value = 546

UDT ? Undefined data type or internal FOROTS error

Cause: Internal FOROTS error.

Recovery: Submit an SPR.

ERRSNS values: First Value = 62 Second Value = 575

UME ? Unexpected MTCHR error <n> (TOPS-10 only)

Cause: This message is not expected to occur. It indicates that a MTCHR UUO has failed. The monitor-supplied error code may give some indication of the reason.

Recovery: Submit an SPR.

ERRSNS values: First Value = 96 Second Value = 531

UMO % <JSYS error> trying to set tape <density/parity/data mode> (TOPS-20 only)

Cause: It was not possible to set the indicated parameter of the tape. The monitor error message gives the reason.

Recovery: Make sure you are using a drive that supports the requested operations.

ERRSNS values: First Value = 96 Second Value = JSYS error number

UNO ? DIVERT: unit <n> is not open

Cause: The file to which error messages are diverted must be opened for output before DIVERT is called.

Recovery: Open the file for output before calling DIVERT.

ERRSNS values: First Value = 21 Second Value = 105

UNS ? Unit not specified

Cause: A call was made to FOROTS which did not contain a unit number.

Recovery: Correct calling code.

ERRSNS values: First Value = 81 Second Value = 501

UOA % Unknown OPEN keyword <n>, ignored

Cause: The compiler has generated an OPEN call that contains an unknown keyword. The keyword is ignored.

Recovery: Make sure you are using the correct versions of the compiler, FORLIB, and FOROTS.

ERRSNS values: First Value = 26 Second Value = 541

USW ? Unknown switch /<sw> (TOPS-10 only)

Cause: In dialog mode, an unknown switch was specified.

Recovery: Retype the line, specifying the correct switch.

ERRSNS values: First Value = 45 Second Value = 241

UTE ? Unexpected TAPOP error <n> (TOPS-10 only)

Cause: This message is not expected to occur. It indicates that a TAPOP UUO has failed. The monitor-supplied error code may give some indication of the reason.

Recovery: Submit an SPR.

ERRSNS values: First Value = 96 Second Value = 530

UTO % Unexpected TAPOP. error <n> trying to set <density/parity/data mode/blocksize> (TOPS-10 only)

Cause: It is not possible to set the indicated parameter of the tape. The monitor error message gives the reason.

Recovery: Make sure you are using a drive that supports the requested operation.

ERRSNS values: First Value = 96 Second Value = 537

VNN ? Variable <var> not in namelist

Cause: NAMELIST input contains an assignment to a variable that is not in the namelist.

Recovery: Correct the input and try again.

ERRSNS values: First Value = 97 Second Value = 309

WBA ? Attempt to WRITE beyond variable or array

Cause: An attempt was made to write beyond the end of a character variable or array with an internal file WRITE statement.

Recovery: Correct program to stay within limits of character variable or array.

ERRSNS values: First Value = 25 Second Value = 576

APPENDIX E

INTERACTIVE DEBUGGER (FORDDT) ERROR MESSAGES

FORDDT responds with two levels of messages - fatal error and warning. Fatal error messages indicate that the processing of a given command has been terminated. Warning messages provide helpful information. The format of these messages is:

```
?FDTxxx text
or
%FDTxxx text
```

where:

```
?    indicates a fatal message
%    indicates a warning message
FDT  is the FORDDT mnemonic
xxx  is the 3-letter mnemonic for error message
text is the explanation of error
```

Square brackets ([ ]) in this section signify variables and are not output on the terminal.

Fatal Errors

The fatal errors in the following list are each preceded by ?FDT on the user terminal. They are listed in alphabetical order.

BDF [symbol] is undefined
[symbol] is multiply defined

BOI Bad octal input

An illegal character was detected in an octal input value.

CCN Cannot continue

A pause has been placed on some form of skip instruction causing FORDDT to loop; should never be encountered in FORTRAN-compiled programs.

CFO Core file overflow

The storage area for GROUP text has been exhausted.

CNU The command [name] is not unique

More letters of the command are required to distinguish it from the other commands.

INTERACTIVE DEBUGGER (FORDDT) ERROR MESSAGES

CSH Cannot START here

The specified entry point is not an acceptable FORTRAN main program entry point.

DNA Double-precision comparisons not allowed

DTO Dimension table overflow

FORDDT does not have the space to record any more array dimensions until some are removed.

FCX Format capacity exceeded

An attempt was made to specify a FORMAT statement requiring more space than was originally allocated by the FORTRAN compiler.

FNI Formal not initialized, please retype

There was a reference to a formal parameter of some subprogram that was never executed.

FNR [array name] is a formal and may not be redefined

Formal parameters may not be DIMENSIONed.

IAF Illegal argument format [rest of user line]

The parameters to the given command were not specified properly. Refer to the documentation for correct format.

IAT Illegal argument type = [number]

An unrecognized subprogram argument type was detected. Submit an SPR if this message occurs.

ICC Comparison of two constants is not allowed

A conditional test involves two constants.

IER Internal FORDDT error [number]

Internal FORDDT error - please report through an SPR.

IGN Invalid group number

Group numbers must be integers and in the range one through eight.

INV Invalid value [rest of user line]

A syntax error was detected in the numeric parameter.

IPN Illegal program name

IRS Illegal range specification

The particular range specified for an array is not defined.

ITM Illegal type modifier - S

The mode S is only valid for ACCEPT statements.

IWI I/O within I/O error

An attempted TYPE or ACCEPT command cannot be executed because a fatal "I/O within I/O" error from FOROTS would result, since the user program is currently processing an IOLST call.

JSE [JSYS error message]

Error reading program name (on TOPS-20 or some other FORDDT input).

LGU [array name] lower subscript .GE. upper

The lower bound of any given dimension must be less than or equal to the upper bound.

LNF [label] is not a FORMAT statement

MCD Compile program with the DEBUG switch to type a FORMAT statement

MLD [array name] multi-level array definition not allowed

The same array cannot be dimensioned more than once (by means of the [dimensions] construct) in a single command.

MSN More subscripts needed

The array is defined to have more dimensions than were specified in the given reference.

NAL Not allowed

An attempt has been made to modify something other than data or a FORMAT.

NAR Not after a reenter

The given command is not allowed until program integrity has been restored by means of a CONTINUE or NEXT command.

NDT DDT not loaded

NFS Cannot find FORTRAN start address for [program name]

Main program symbols are not loaded.

NFV [symbol] is not a FORTRAN variable

Names must be 6-character alphanumeric strings beginning with a letter.

NGF Cannot GOTO a FORMAT statement

NPH Cannot insert a PAUSE here

An attempt has been made to place a breakpoint at other than an executable statement or subprogram entry point.

NSP [symbol] no such PAUSE

An attempt has been made to REMOVE a breakpoint that was never set up.

NUD [symbol] not a user-defined array

An attempt has been made to remove dimension information for an array that was never defined.

PAR Parentheses required, please retype

Parentheses are required for the specification of FORMAT statements and complex constants.

PRO Too many PAUSE requests

The PAUSE table has been exhausted. The maximum limit is 10.

RGR Recursive group reference

A group may not reference itself.

SER Subscript error

The subscript specified is outside the range of its defined dimensions.

STL [array name] size too large

An attempt has been made to define an array larger than 256K.

TMS Too many subscripts [dimensions]

The array is defined to have fewer dimensions than are specified in the given element reference.

URC Unrecognized command [command]

Warning Messages

Each warning message is preceded by %FDT on your terminal. The warning messages are listed here in alphabetical order.

ABX [array name] compiled array bounds exceeded

FORDDT has detected another symbol defined in the specified range of the array. Note that this will occur in certain EQUIVALENCE cases and can be ignored at that time.

CAB Cannot allocate buffer for help file

CHI Characters ignored: "[text]"

The portion of the command string included in "text" was thought to be extraneous and was ignored.

ECI Buffer full - excess characters ignored

EOH Error opening help file

IOE I/O error reading help file

IWI I/O within I/O

FORDDT has PAUSEd at a breakpoint while the user program is currently processing an IOLST call. TYPE and ACCEPT commands cannot be processed at this breakpoint.

NAA [symbol] is not an array

NHF Cannot find help file; I'm sorry I can't help you

NSL No symbols loaded

FORDDT cannot find the symbol table.

NST Not STARTed

The specified command requires that a START be previously issued to ensure that the program is properly initialized.

POV Program overlayed by [program name]
Program overlayed by ***

The symbol table is different from the last time FORDDT had control. The program name is printed only if it has changed, otherwise '***' is printed.

SCA Supersedes compiled array dimension

The FORTRAN generated dimension is being superseded for the given array.

SPO Variable is single-precision only

WSP Writing to shared page

XPA Attempt to exceed program area with [symbol name]

An attempt has been made to access memory outside the currently defined program space.

APPENDIX F

FORTRAN-SUPPLIED PLOTTER SUBROUTINES

The FORTRAN subroutine library contains a set of subroutines that are used with plotting devices. To successfully use these routines, a plotter must be connected to your system. The FORTRAN software contains the following plotter subroutines:

PLOTS — The PLOTS subroutine initializes the plotter or reports if the plotter is not available. This must be the first plotter subroutine specified. (See Section F.1.)

AXIS — The AXIS subroutine draws an axis with tic marks and scale values at 1-inch increments. An identifying label may also be plotted along the axis. (See Section F.2.)

CAXIS — The CAXIS subroutine performs the same functions as the AXIS subroutine. (See Section F.3.)

LINE — The LINE subroutine draws a line through the points specified. (See Section F.4.)

MKTBL — The MKTBL subroutine specifies a special character set. (See Section F.5.)

NUMBER — The NUMBER subroutine causes floating-point numbers to be plotted as text. (See Section F.6.)

PLOT — The PLOT subroutine moves the plotter pen to a new position. Raising and lowering the pen is also specified in the PLOT subroutine. (See Section F.7.)

SCALE — The SCALE subroutine selects scale values for the AXIS or CAXIS subroutine. (See Section F.8.)

SETABL — The SETABL subroutine specifies a character set. (See Section F.9.)

SYMBOL — The SYMBOL subroutine raises the plotter pen, moves it to the position specified by x and y, and plots a string of characters. (See Section F.10.)

WHERE — The WHERE subroutine reports on the current position of the plotter pen. (See Section F.11.)

PLOTS
Subroutine

F.1 PLOTS SUBROUTINE

The PLOTS subroutine initializes the plotter or reports if the plotter is not available. This must be the first plotter subroutine specified.

The form of the PLOTS subroutine is:

```
     CALL PLOTS(i[,steps])
```

where:

i — is an integer variable, which is set to -1 if the plotter is not available, or set to 0 if the plotter is available.

The plotter may not be available because: the system does not have a plotter; the plotter is in use by another user; the plotter is turned off; or the plotter is being spooled, but you are trying to write to it directly.

steps — is an optional floating-point variable or constant that specifies the number of steps (per inch) used by the plotter. The default is 100 steps per inch, and may be changed by the installation. Commonly, plotters plot 100 steps per inch, 200 steps per inch, or 100 steps per centimeter (about 254 steps per inch).

AXIS
Subroutine

F.2 AXIS SUBROUTINE

The AXIS subroutine draws an axis with tic marks and scale values at 1-inch increments. An identifying label may also be plotted along the axis.

NOTE

> The AXIS subroutine is provided for compatibility with previous versions of FORTRAN-10/20. The AXIS subroutine uses a numeric array to contain the label that is plotted along the axis. The CAXIS subroutine (see Section F.3) allows a character expression to contain the label.

The form of the AXIS subroutine is:

```
CALL AXIS(x,y,asc,nasc,size,theta,xmin,dx)
```

where:

x,y — is a variable or constant pair that specifies the starting point of the axis.

asc — is the name of a numeric array that contains a label that is plotted along the axis.

nasc — is an integer constant or variable that specifies the number of characters contained in array asc. If nasc is negative, the label in the array is placed on the clockwise side of the axis. If nasc is positive, the tic marks, label, and scale values are placed on the counterclockwise side of the axis.

size — is a constant or variable that specifies the length of the axis in inches.

theta — is a constant or variable that specifies the angle at which the axis is plotted. The value of theta is usually 0.0 or 90.0.

xmin — is a variable or constant that specifies the value of the scale at the beginning of the axis.

dx — is a variable or constant that specifies the change in scale for a 1-inch increment.

NOTE

> The proper values for xmin and dx may be determined by calling the SCALE subroutine (see Section F.8).

CAXIS
Subroutine

F.3 CAXIS SUBROUTINE

The CAXIS subroutine draws an axis with tic marks and scale values at 1-inch increments. An identifying label may also be plotted along the axis.

The form of the CAXIS subroutine is:

```
CALL CAXIS(x,y,asc,sign,size,theta,xmin,dx)
```

where:

x,y — is a variable or constant pair that specifies the starting point of the axis.

asc — is the name of a character expression that contains a label that is plotted along the axis.

sign — If nasc is negative, the label in the array is placed on the clockwise side of the axis. If nasc is positive, the tic marks, label, and scale values are placed on the counterclockwise side of the axis.

size — is a constant or variable that specifies the length of the axis in inches.

theta — is a constant or variable that specifies the angle at which the axis is plotted. The value of theta is usually 0.0 or 90.0.

xmin — is a variable or constant that specifies the value of the scale at the beginning of the axis.

dx — is a variable or constant that specifies the change in scale for a 1-inch increment.

NOTE

The proper values for xmin and dx may be determined by calling the SCALE subroutine (see Section F.8).

LINE
Subroutine

F.4 LINE SUBROUTINE

The LINE subroutine draws a continuous line through a set of points.

The form of the LINE subroutine is:

```
CALL LINE(x,y,n,k)
```

where:

x — is the name of an array that contains the floating-point x-coordinates of the points to be plotted.

y — is the name of an array that contains the floating-point y-coordinates of the points to be plotted.

n is an integer constant or variable that specifies the total number of points to be plotted.

k is an integer constant or variable that equals the number of elements of x and y. Since single-precision one-dimensional arrays are usually used, this value is usually 1.

MKTBL Subroutine

F.5 MKTBL SUBROUTINE

The MKTBL subroutine defines a special character set to be used when plotting; the SETABL subroutine (see Section F.9) enables you to use the character set defined by the MKTBL subroutine.

The form of the MKTBL subroutine is:

```
     CALL MKTBL(setnumber,tableaddress)
```

where:

setnumber is an integer variable from 1 to 10 that specifies the numeric identifier of the character set, for example, the number of the ASCII character set is 1. If the character set cannot be defined by a call to MKTBL, a value of zero is returned in this variable.

tableaddress is a constant or variable that specifies the starting address of a character table that has 128 (200 octal) consecutive words. Each character table word contains the number of strokes (line segments) for the character in the left half, and the address of the table of strokes in the right half. See Section F.5.1 for a description of these tables.

F.5.1 Character Tables

The next sections describe how to define and organize a character set table. The character set, called by the MKTBL subroutine, enables you to create and use a character set other than the default character set that is used to plot characters. (Usually the default character set is ASCII.)

To create your own character set, you need to create a character table and a character stroke table. These two tables are described in Sections F.5.1.1 and F.5.1.2, respectively.

F.5.1.1 Creating a Character Table - A character table contains 128 entries (200 octal). Each entry in the character table, regardless of whether the table is defined in the plotter subroutine library or by you through a call to the MKTBL subroutine, indicates the character to be plotted for the ASCII character that has that numerical value.

Figure F-1 is a diagram of an entry word in a character table.

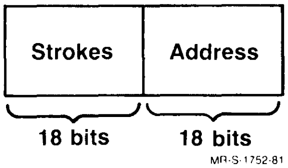


Figure F-1: Plotter Character Table Entry

As shown in Figure F-1, each entry in the character table contains the number of strokes (line segments) required to plot the character in its left half and an address reference in the right half of the word. The address in the right half of the entry references an entry in the character stroke table for the character set. The character stroke table is described in the next section.

F.5.1.2 Creating a Stroke Table - Each character in the character table has a corresponding character stroke table. The purpose of the character stroke table is to define the number and type of strokes (drawn line segments) it takes to produce a character in the character table.

Figure F-2 is a diagram of an entry in a character stroke table.

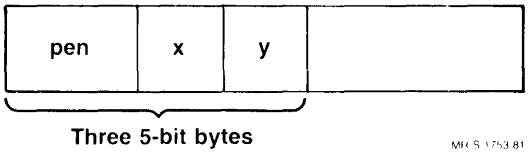


Figure F-2: Character Stroke Table Entry

As shown in Figure F-2, each stroke is described in the character stroke table by three 5-bit bytes. The possible values for these three bytes are:

Byte	Value and Meaning	
1	0	pen is raised (off the paper)
	1	pen is lowered (on the paper)
2	x-coordinate value at end of stroke	
3	y-coordinate value at end of stroke	

When determining the height and width of each character to be plotted, consider the following conventions:

1. Characters are drawn within a grid that is 15 units high by eight units wide.

2. Characters are generally plotted six units above the base line. Two units are generally left blank to the right of each character; one unit is generally left blank on top of each character. The spacing above, below, and on either side of each character provides adequate spacing between characters and between lines.

3. The plotter starts drawing each character at the lower left corner of the character grid. If the grid is set up in the conventional manner, the lower left corner is grid position (0,6). Normal width characters end at the lower right corner of the character grid after allowing spacing between characters.

   The last coordinate in each character grid is usually (8,6). The character grid for the next character has as its origin (0,0) at the ending coordinate in the previous grid (8,6).

4. Accents, circumflexes, underscores, and other characters that are to be plotted in the same character grid as another character should end at the same grid position as they began. By doing this, for example, the ASCII character that represents the accent character is plotted before the character that is the letter to be accented.

F.5.1.3 Sample Character Stroke Table - In this sample, you want to plot the Greek letter beta as a normal sized character. When plotted, the letter is drawn on a 15 x 8 unit grid above the base line, allowing for spacing between characters and between lines.

The strokes (line segments) for beta, including the invisible segments, are then determined. The character will be plotted more smoothly if as many of the line segments as possible are connected, and if doubled segments are avoided. Note that stroke 9 (the return stroke for the crossbar of the beta) and stroke 15 (moving the pen to ending point (8,6) are not visible; the pen is raised.

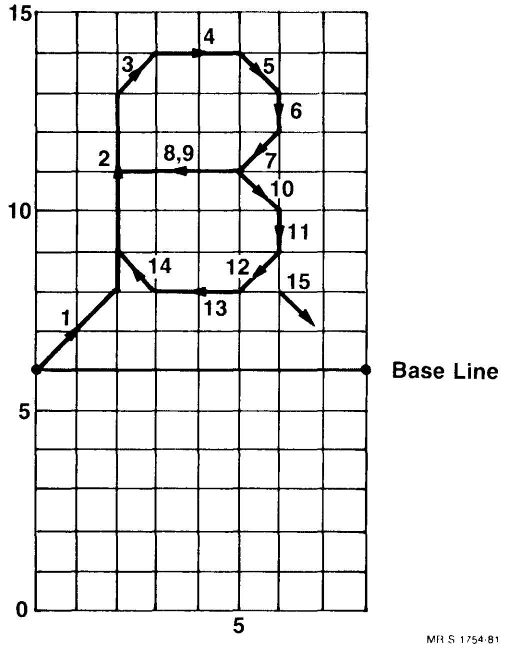


Figure F-3: Sample Character Stroke Table

If the address at the beginning of the table that includes the character in Figure F-3 is called GREEK, then the FORTRAN call to set up the table could be:

```
IGREEK=4
CALL MKTBL (IGREEK, GREEK)
```

The above call to the MKTBL subroutine defines the GREEK character table to be the fourth character table. To use the GREEK table for plotting, use the FORTRAN call:

```
CALL SETABL (IGREEK, IFLAG)
```

The entry in the character table for beta is:

```
RADIX ^ D10  ;VALUES IN DECIMAL
BETA:  15,,TBETA ;CHARACTER TABLE ENTRY FOR BETA
```

Note that the character table entry for beta contains the starting address of the character stroke table (TBETA) in its right half.

The character stroke table for beta is:

```
        SEEN==1        ;IF 'SEEN' THEN THE STROKE MARKS THE PAPER
        UNSEEN==0      ;IF 'UNSEEN' THEN THE STROKE IS INVISIBLE
        RADIX ^D10     ;ALL VALUES ARE IN DECIMAL
TBETA:  BYTE(5) SEEN,2,8,SEEN,2,13
        BYTE(5) SEEN,3,14,SEEN,5,14
        BYTE(5) SEEN,6,13,SEEN,6,12
        BYTE(5) SEEN,5,11,SEEN,2,11
        BYTE(5) UNSEEN,5,11,SEEN,6,10
        BYTE(5) SEEN,6,9,SEEN,5,8
        BYTE(5) SEEN,3,8,SEEN,2,9
        BYTE(5) UNSEEN,8,6
```

F.5.1.4 FORTRAN- and User-Defined Character Sets - The standard ASCII character set is always defined and is character set 1 for calls to the SETABL subroutine, unless character set 1 is redefined by a user call to the MKTBL subroutine. If SETABL is not called, the ASCII character set is the default.

The Cyrillic (Russian) character set is available as character set 2, and the Feanorian character set is available as character set 3 (unless character sets 2 or 3 has been redefined by a user call to the MKTBL subroutine). In order to use these character sets, the user program must contain an EXTERNAL statement for variable PLOTF (Feanorian) or PLOTC (Cyrillic).

User-defined character sets should use character sets 4 through 10 to avoid conflicts with the standard character sets.

NUMBER
Subroutine

F.6 NUMBER SUBROUTINE

The NUMBER subroutine causes floating-point numbers to be plotted as text.

The form of the NUMBER subroutine is:

```
     CALL NUMBER(x,y,size,fnum,theta,ndigit)
```

where:

x,y — are variables or constants that specify the x and y coordinates of the point to be plotted. The specified point is the lower left corner of the first character to be plotted.

size — is a variable or constant that specifies the size (in inches) of the digits to be plotted. The specified value should be a multiple of .08 inches (or centimeters if plotter is metric) if a small value is used.

fnum — is a variable or constant that is the floating-point number to be plotted.

theta — is a variable or constant that specifies the direction (in degrees) of the base line on which the characters are plotted.

ndigit — is a variable or constant that specifies the number of digits to be plotted to the right of the decimal point. If ndigit is negative, only the integer part of the number is plotted; the resulting integer is rounded.

PLOT Subroutine

F.7 PLOT SUBROUTINE

The PLOT subroutine moves the plotter pen to a new position. Raising and lowering the pen is also specified in the PLOT subroutine.

NOTE

> The plotter is not released after completion of the specified movement.

The form of the PLOT subroutine is:

```
     CALL PLOT(x,y,penup/down)
```

where:

x,y — are the variables or constants that specify the x and y coordinates for the point to which the pen will be moved.

penup/down — is an integer constant or variable that specifies whether the pen is on the paper or above the paper. The possible values for this variable are:

3 = raise pen before movement
2 = lower pen before movement
1 = leave pen in current state (raised or lowered)

-1, -2, or -3 = same as corresponding positive values except that on completion of the indicated motion, the new pen position is taken as a new origin and the output buffer is sent to the plotter. Using the negative values is helpful if you are plotting consecutive characters in the same program.

SCALE Subroutine

F.8 SCALE SUBROUTINE

The SCALE subroutine scales values for the AXIS subroutine.

The form of the SCALE subroutine is:

```
     CALL SCALE(x,n,s,xmin,dx)
```

where:

x — is an array name of a one-dimensional floating-point array to be scaled for the AXIS subroutine.

n — is an integer constant or variable that specifies the length of the array in words (36-bit).

s — is a constant or variable that specifies the length (in inches) of the desired axis.

xmin — is a constant or variable that specifies the smallest element in array x. The value of xmin will be the value of the scale at the beginning of the axis.

dx — is a constant or variable that equals the change in scale for a 1-inch interval so that array x can be plotted in inches.

SETABL Subroutine

F.9 SETABL SUBROUTINE

The SETABL subroutine enables you to select the character set that is used to plot characters.

The form of the SETABL subroutine is:

```
CALL SETABL (setnum,status)
```

where:

setnum — is an integer constant or variable that equals the number of the character set. The standard ASCII character set is defined to be set 1 and is the default. Character sets are defined by the MKTBL subroutine (see Section F.5).

status — is an integer variable whose value after the call to the SETABL subroutine is either 0, if the character set specified in setnum is valid, or -1, if the character set specified by setnum is invalid.

NOTE

> If you use a character set other than the character sets defined by default in the plotter subroutine library, you must call the MKTBL subroutine before calling the SETABL subroutine.

SYMBOL
Subroutine

F.10 SYMBOL SUBROUTINE

The SYMBOL subroutine plots a specified string of characters (from either the default character set or the character set specified by the last successful call to the SETABL subroutine).

The form of the SYMBOL subroutine is:

```
     CALL SYMBOL(x,y,size,asc,theta[,nasc])
```

where:

x — is a constant or variable that equals the x coordinate of the lower left corner of the first character to be plotted.

y — is a constant or variable that equals the y coordinate of the lower left corner of the first character to be plotted. The plotter pen is raised and moved to position x,y before the string of characters is plotted.

size — is a constant or variable that specifies the height (in inches) of the character to be plotted. The specified value should be a multiple of .08 inches (or centimeters if you have a metric plotter).

asc — is the name of a character expression or numeric array that contains the ASCII characters to be plotted.

theta — is a constant or variable that specifies the direction (in degrees) of the base line on which the characters are to be plotted.

nasc — is an integer constant or variable that is equal to the number of characters in numeric array asc that are to be plotted. This is ignored if a character expression is specified for asc.

WHERE
Subroutine

F.11 WHERE SUBROUTINE

The WHERE subroutine reports on the current position of the plotter pen, in inches, relative to the origin.

NOTE

> The plotter origin is set by a call to the AXIS subroutine or a call to the PLOT subroutine that has a negative value for the penup/down variable. Also, the WHERE subroutine does not allow you to determine whether the plotter pen is raised or lowered.

The form of the WHERE subroutine is:

```
     CALL WHERE(x,y)
```

where:

x   is a variable in which the subroutine returns the x coordinate of the current print position.

y   is a variable in which the subroutine returns the y coordinate of the current pen position.

INDEX

TOPS–10/TOPS–20
FORTRAN Language Manual
AA–N383B–TK

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- □ Assembly language programmer
- □ Higher-level language programmer
- □ Occasional programmer (experienced)
- □ User with little programming experience
- □ Student programmer
- □ Other (please specify) ____________

Name ____________ Date ____________

Organization ____________ Telephone ____________

Street ____________

City ____________ State ________ Zip Code
or Country ________

-----Do Not Tear - Fold Here and Tape---

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MRO1-2/L12
MARLBOROUGH, MA 01752

---- Do Not Tear - Fold Here and Tape---