60481300

CONTROL DATA CORPORATION

FORTRAN
VERSION 5
REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1
SCOPE 2

60481300

CONTROL DATA CORPORATION

FORTRAN
VERSION 5
REFERENCE MANUAL

CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1
SCOPE 2

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original release.
(7-20-79)	
B	Revised to reflect the released version of the FORTRAN 5 compiler. Numerous technical and
(9-28-79)	miscellaneous corrections, including added shading, have been made.
C	Revised to include SCOPE 2 operating system. Additional technical and miscellaneous corrections
(2-15-80)	have been made.
D	Released at PSR level 527. Revised to include CMM interface and update to PMD. Additional
(9-26-80)	technical and miscellaneous corrections have been made.
E	Revised at PSR level 533. Revised to reflect release of PMD and STATIC option under SCOPE 2
(1-16-81)	operating system.
Publication No. 60481300	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	-
Title Page	-
ii	E
iii	E
iv	E
v	E
vi	E
vii	D
viii thru xii	E
xiii/iv	E
xv	B
1-1	D
1-2	D
1-3	B
1-4	C
1-5	D
1-6	B
1-7	E
1-8	B
1-9 thru 1-12	D
2-1	B
2-2	D
2-3 thru 2-5	E
2-6	D
2-7	E
2-8 thru 2-10	D
2-11	B
2-12	B
3-1	A
3-2 thru 3-4	B
3-5 thru 3-7	D
3-8	A
3-9	B
4-1	E
4-2	E
4-3	D
4-4	E
4-5 thru 4-10	D
5-1 thru 5-4	D
5-5 thru 5-7	E
5-8	C
5-9 thru 5-12	D
5-13	C
5-14	B
5-15	B
5-16	E
5-17	D
5-18	B
5-19	B
5-20	C
5-21	E
5-22	E
5-23	D
5-24 thru 5-26	B
5-27	C
5-28 thru 5-30	E
5-31	D

Page	Revision
5-32 thru 5-37	E
5-38	B
6-1	D
6-2	D
6-3 thru 6-5	E
6-6	C
6-7	C
6-8	B
6-9 thru 6-11	E
7-1	E
7-2	B
7-3 thru 7-6	A
7-7	E
7-8	E
7-9	B
7-10	C
7-11	C
7-12	D
7-13	E
7-14	D
7-15 thru 7-17	C
7-18 thru 7-20	D
7-20.1/7-20.2	D
7-21	C
7-22	C
7-23	B
7-24	C
7-25	B
7-26	E
7-27	B
7-28	B
7-29 thru 7-31	E
8-1 thru 8-6	E
8-6.1/8-6.2	E
8-7	B
8-8	C
8-9	C
8-10	B
8-11	C
8-12	B
8-13	B
9-1 thru 9-5	D
10-1 thru 10-5	E
11-1 thru 11-6	D
11-7	E
11-8	D
11-9	D
11-10 thru 11-16	E
11-16.1	E
11-16.2	E
11-17 thru 11-20	C
11-21	D
11-22	E
12-1	A
12-2	A
12-3	B
12-4	B

Page	Revision
12-5	E
12-6	B
12-7	C
12-8	B
12-9	E
12-10	B
12-11	B
12-12	C
12-13 thru 12-21	B
12-22	C
12-23	B
12-24	B
12-25	C
12-26	B
12-27	B
12-28	E
12-29	B
12-30	E
12-31	B
12-32	B
12-33	A
12-34	C
12-35	C
12-36	B
A-1	C
A-2	D
A-3	E
A-4	A
A-5	B
B-1 thru B-37	E
C-1	C
C-2 thru C-4	D
D-1	E
D-2	E
D-3	C
D-4	B
D-5	E
D-6	E
E-1 thru E-4	B
F-1	D
F-2	C
F-3 thru F-5	E
F-6 thru F-9	C
G-1	D
G-2	E
Index-1 thru Index-6	E
Comment Sheet	E
Mailer	-
Back Cover	-

PREFACE

This manual describes the FORTRAN Version 5 language. FORTRAN Version 5 complies with the American National Standards Institute FORTRAN language described in document X3.9-1978 and known as FORTRAN 77. FORTRAN Version 5 extensions to FORTRAN 77 are indicated by shading.

The reader should be familiar with FORTRAN Extended Version 4 or an existing FORTRAN language. The reader should also be familiar with the operating system on which FORTRAN Version 5 jobs will be compiled and executed.

The FORTRAN Version 5 (FORTRAN 5) compiler is available under control of the following operating systems:

NOS 1 for the CONTROL DATA® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

NOS/BE 1 for the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

SCOPE 2 for CONTROL DATA® CYBER 170 Model 176, CYBER 70 Model 76, and 7600 Computer Systems.

Extended memory for the CYBER 170 Model 176 is large central memory (LCM) or large central memory extended (LCME). Extended memory for all other computer systems is extended core storage (ECS) or extended semi-conductor memory (ESM). In this manual, the acronym ECS refers to all forms of extended memory unless otherwise noted. Programming information for the various forms of extended memory can be found in the COMPASS reference manual and in the appropriate computer system hardware reference manual.

Related material is contained in the listed publications. The NOS manual abstracts and the NOS/BE manual abstracts are instant-sized manuals containing brief descriptions of the contents and intended audience of all NOS operating system and NOS product set manuals, and NOS/BE operating system and NOS/BE product set manuals, respectively. The abstracts manuals can be useful in determining which manuals are of greatest interest to a particular user. The Software Publications Release History serves as a guide in determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software. Other publications serve as references for information that requires greater detail.

The following publications are of primary interest:

Publication	Publication Number
FORTRAN Extended Version 4 to FORTRAN Version 5 Conversion Aid Program Reference Manual	60483000
FORTRAN Version 5 Common Library Mathematical Routines Reference Manual	60483100
FORTRAN Version 5 Instant	60483900
NOS Version 1 Reference Manual, Volume 1 of 2	60435400
NOS/BE Version 1 Reference Manual	60493800
SCOPE Version 2 Reference Manual	60342600

The following publications are of secondary interest:

Publication	Publication Number
Common Memory Manager Version 1 Reference Manual	60499200
COMPASS Version 3 Reference Manual	60492600
CYBER Interactive Debug Version 1 Reference Manual	60481400
CYBER Loader Version 1 Reference Manual	60429800
CYBER Record Manager Advanced Access Methods Version 2 Reference Manual	60499300
CYBER Record Manager Advanced Access Methods Version 2 User's Guide	60499400

CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual	60495700
CYBER Record Manager Basic Access Methods Version 1.5 User's Guide	60495800
DMS-170 DDL Version 3 Reference Manual Volume 1: Schema Definition for Use With: COBOL FORTRAN Query Update	60481900
FORTRAN Data Base Facility Version 1 Reference Manual	60482200
INTERCOM Interactive Guide for Users of FORTRAN Extended	60455950
INTERCOM Version 5 Reference Manual	60455010
Network Products Interactive Facility Version 1 Reference Manual	60455250
NOS Version 1 Manual Abstracts	84000420
NOS Version 1 Time-Sharing User's Reference Manual	60435500
NOS/BE Version 1 Manual Abstracts	84000470
SCOPE Version 2 Loader Reference Manual	60454780
SCOPE Version 2 Record Manager Reference Manual	60495700
Software Publications Release History	60481000
Sort/Merge Versions 4 and 1 Reference Manual	60497500

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

NOTATIONS

Certain notations are used throughout the manual with consistent meaning. The notations are:

UPPERCASE	In language syntax, uppercase indicates a statement keyword or character that is to be written as shown.
Lowercase	In language syntax, lowercase indicates a name, number, symbol, or entity that is to be supplied by the programmer.
[ ] Brackets	In language syntax, brackets indicate an optional item that can be used or omitted.
{ } Braces	In language syntax, braces indicate that only one of the vertically stacked items can be used.
... Ellipsis	In language syntax, an ellipsis indicates that the preceding optional item in brackets can be repeated as necessary.
. . . Ellipsis	In program examples, an ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.
Δ Delta	A delta indicates a blank character.
Shading	In language syntax, language descriptions, and program examples, shading indicates extensions to FORTRAN 77.

A FORTRAN program is written to perform a specific sequence of operations. Each FORTRAN program uniquely deals with the solution of a particular problem or set of problems. Each program typically works with input values, performs calculations and data manipulation, and produces output values that are either printed or saved in some way. This manual describes the full capabilities of FORTRAN Version 5. The FORTRAN programmer must select and use the capabilities needed for each particular program.

CDC offers guidelines for the use of the software described in this manual. These guidelines appear in appendix G. Before using the software described in this manual, the reader is strongly urged to review the content of this appendix. The guidelines recommend use of this software in a manner that reduces the effort required to migrate application programs to future hardware or software systems.

WRITING FORTRAN STATEMENTS

The FORTRAN character set is used for writing FORTRAN statements. The FORTRAN character set consists of 26 letters, 10 digits, and 13 or 14 special characters. The FORTRAN character set is shown in table 1-1.

TABLE 1-1. FORTRAN CHARACTER SET

Type	Characters
Alphabetic	A through Z
Numeric	0 through 9
Special Characters	= equal + plus - minus * asterisk / slash (left parenthesis) right parenthesis , comma . decimal point $ currency symbol ' apostrophe (CDC graphic ↑) : colon " quote (CDC graphic ≠) blank

The representations of characters are described in appendix A. In all but two cases, the FORTRAN character and the representation are identical. If the CDC 63-character set or 64-character set is in use, the two exceptions are ' and ", which are represented as ↑ and ≠, respectively. If the ASCII 63-character set or 64-character set is in use, the characters and representations are all identical.

Characters that are not included in the FORTRAN character set can be used in character and Hollerith constants; in apostrophe, H, and quote descriptors in format specifications; and in comment lines.

FORTRAN statements can be written in normal (nonsequenced) mode. FORTRAN statements can also be written in sequenced mode. Each program must be written entirely in one mode. Normal mode is principally used for batch jobs. Sequenced mode is suited to most time-sharing applications. The SEQ parameter of the FTN5 control statement (described in section 11) selects sequenced mode.

NONSEQUENCED MODE

The FORTRAN source program can be written on the coding form shown in figure 1-1. Each line on the coding form represents a source line, either a card image or terminal line.

The lines coded in a FORTRAN program are initial lines, continuation lines, and comment lines. Lines can also be compiler directives. The column usage for nonsequenced mode lines is shown in figure 1-2.

A nonsequenced mode line consists of characters in columns 1 through 72. The identification field in columns 73 through 80 is not defined as part of the line.

Initial Lines

Each statement contains an initial line. The initial line of a statement is written in columns 7 through 72. Blanks can be used to improve readability. The initial line of a statement can contain a statement label in columns 1 through 5.

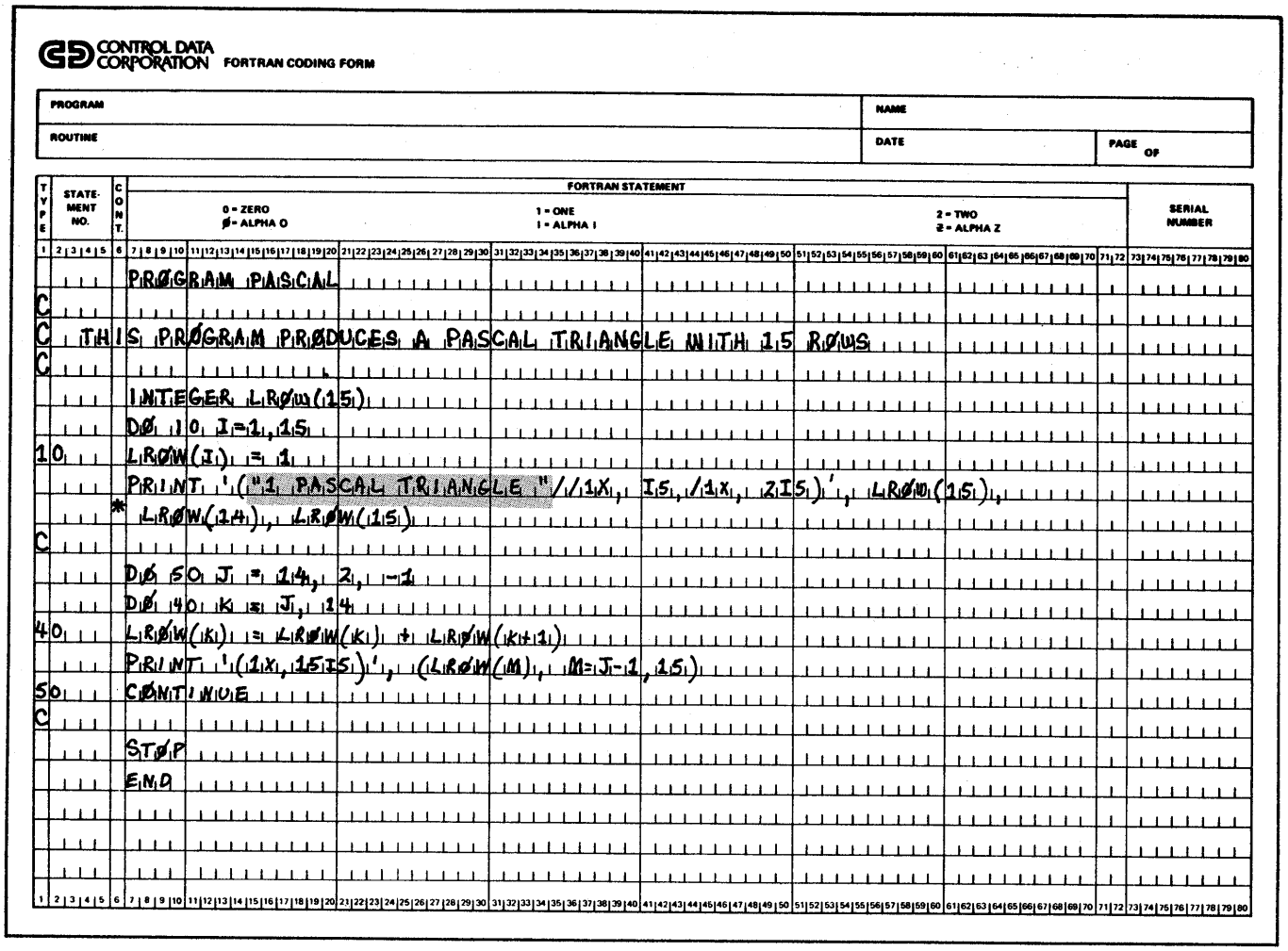
Continuation Lines

Statements are coded in columns 7 through 72. If a statement is longer than 66 characters, it can be continued on as many as 19 continuation lines. A character other than blank or zero in column 6 indicates a continuation line. Columns 1 through 5 must be blank.

The length of a statement cannot exceed 1320 characters. The maximum length includes one initial line and 19 continuation lines, at 66 characters per line since the statement is contained in columns 7 through 72.

Statement Labels

A statement label (any 1- to 5-digit positive nonzero integer) can be written in columns 1 through 5 of the initial line of a statement. A statement label uniquely identifies a statement so that it can be referenced by other statements. Statements that will not be referenced do not need labels. Blanks and leading zeros are not significant. Labels need not occur in numerical order, but



CONTROL DATA CORPORATION FORTRAN CODING FORM

PROGRAM | NAME

ROUTINE | DATE | PAGE OF

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT (0 = ZERO Ø = ALPHA O; 1 = ONE I = ALPHA I; 2 = TWO Z = ALPHA Z)	SERIAL NUMBER
			PRØGRAM PASCAL	
C				
C			THIS PRØGRAM PRØDUCES A PASCAL TRIANGLE WITH 15 RØWS	
C				
			INTEGER LRØW(15)	
			DØ 10 I=1,15	
10			LRØW(I) = 1	
			PRINT '("1 PASCAL TRIANGLE "//1X, I5,/1X, 2I5)', LRØW(15),	
		*	LRØW(14), LRØW(15)	
C				
			DØ 50 J = 14, 2, -1	
			DØ 40 K = J, 14	
40			LRØW(K) = LRØW(K) + LRØW(K+1)	
			PRINT '(1X, 15I5)', (LRØW(M), M=J-1, 15)	
50			CØNTINUE	
C				
			STØP	
			END	

Figure 1-1. Program on FORTRAN Coding Form

a given label must not be defined more than once in the same program unit. A label is known only in the program unit containing it and cannot be referenced from a different program unit. Any statement can be labeled, but only FORMAT and executable statement labels can be referenced by other statements.

Comment Lines

One of the characters C or * in column 1 indicates a comment line. Comments do not affect the program and can be placed anywhere within the program. Comments can appear between an initial line and a continuation line, or between two continuation lines. Comments provide a method of placing program documentation in the source program.

Any line with blanks in columns 1 through 72 is also a comment line. Comment lines following an END statement are listed at the beginning of the next program unit.

Additional characters that are not in the FORTRAN character set can be included in comment lines. Comment lines can include any characters listed in appendix A for the character set being used.

Compiler Directive Lines

The characters C and $ in columns 1 and 2 indicate a compiler directive line. A compiler directive must appear on a single line, and any compiler directive terminates statement continuation.

Compiler directives are effective unless the DS parameter of the FTN5 control statement is used to suppress interpretation of compiler directives. If directive suppression is specified, any compiler directives are interpreted as comment lines.

The directive, including keyword and parameters, is written in columns 7 through 72. Compiler directives are described in appendix E.

Columns 73 through 80

Any identification information can appear in columns 73 through 80 and is not considered part of the statement or the line. Characters in the identification field are ignored by the compiler but are copied to the source program listing. If input comes from other than cards, columns 73 through 90 can be used for identification information.

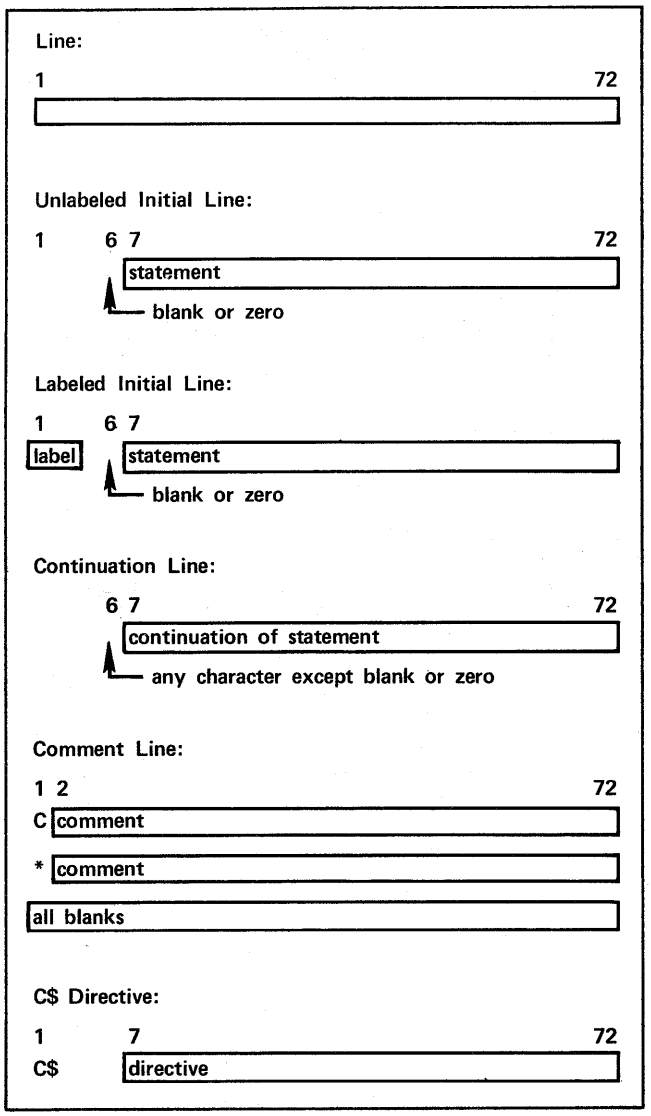


Figure 1-2. Normal Column Usage

SEQUENCED COLUMN USAGE

The FORTRAN program can be written with sequenced lines, as shown in figure 1-3. Each line represents a source line, and sequenced lines usually begin with a sequence number of one through five digits. The sequence numbers for source lines are usually in ascending order and can be supplied for the user during interactive creation of the program (under NOS only). The user can also simply write the program using sequence numbers. Source lines are interpreted as sequenced lines if the SEQ parameter of the FTN5 control statement is specified.

Like nonsequenced mode lines, sequenced mode lines can be initial lines, continuation lines, comment lines, and compiler directive lines. The column usage for sequenced mode lines is shown in figure 1-4.

A line consists of characters in columns 1 through 80. The sequence number of a sequenced line must appear to the left of all other nonblank characters in the line. The

```
00100    PROGRAM PASCAL
00110---PRODUCES A PASCAL TRIANGLE
00120    INTEGER LROW(15)
00130    DO 10 I=1,15
00140 10 LROW(I)=1
00150    DO 50 I=2,14
00160    J=16-I
00170    DO 40 K=J,14
00180 40 LROW(K)=LROW(K)+LROW(K+1)
00190    L=J-1
00200    PRINT '(1X,15I5)',(LROW(M),M=L,15)
00210 50 CONTINUE
00220    STOP
00230    END
```

Figure 1-3. Listing of Sequenced Program

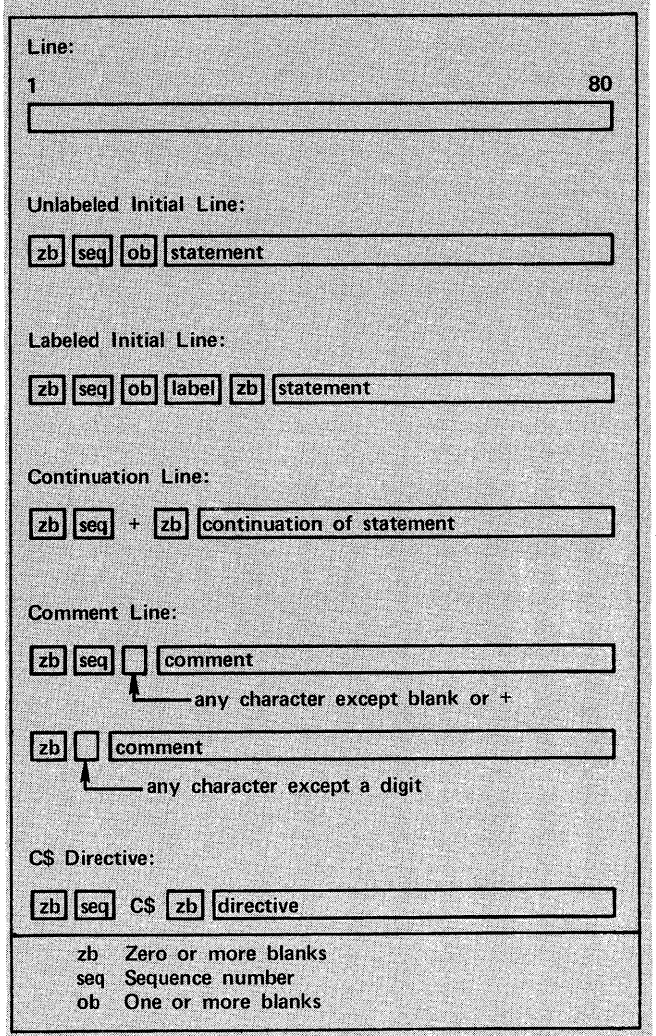


Figure 1-4. Sequenced Column Usage

sequence number consists of one through five digits, usually at the beginning of the line. Blanks can precede the sequence number.

The statement can begin immediately after one or more blanks following the sequence number. Blanks can be used within the statement to improve readability.

The rules for writing sequenced lines are the same as for nonsequenced lines, with the following exceptions:

If a statement label is included, it follows the sequence number and must be separated from the sequence number by at least one blank. The statement can begin immediately after the label or it can be separated from the label by one or more blanks.

A continuation line has the character + immediately following the sequence number. Blanks can be used between the + and the continuation of the statement.

A comment line has any character except blank or + immediately following the sequence number. Any line without a sequence number is also a comment line. Note that in sequenced mode, comment lines can begin with characters other than C or *.

A compiler directive line has the characters C and $ immediately following the sequence number. Blanks can be used between the characters C$ and the beginning of the directive.

SYMBOLIC NAMES

A symbolic name is assigned by the user and consists of one through seven letters and digits (ANSI only allows six), beginning with a letter. Symbolic names are used for the following:

Main program name

Common block name

Subroutine name

External function name

Block data subprogram name

Variable name

Array name

Symbolic constant name

Intrinsic function name

Statement function name

Dummy procedure name

NAMELIST group name

Names which are FORTRAN keywords can be used as user-assigned symbolic names without conflict. For example:

```
PROGRAM TEST

PRINT = 1.0

PRINT*, PRINT
      .
      .
      .
```

The name PRINT is legally used as a variable name and FORTRAN keyword.

In general, however, it is good programming practice to avoid naming conflicts by assigning unique names to program entities. Certain of these conflicts are illegal and are diagnosed. For example:

```
PROGRAM ALPHA
ALPHA = 1.0
      .
      .
      .
```

illegally uses the name ALPHA as a program unit name and a variable name.

CONSTANTS

A constant is a fixed quantity. The seven types of constants are integer, real, double precision, complex, Boolean, logical, and character constants. The PARAMETER statement described in section 2 can be used to declare a symbolic constant. Integer, real, double precision, complex, and Boolean constants are considered arithmetic constants.

INTEGER

An integer constant is a string of 1 through 18 decimal digits written without a decimal point, as shown in figure 1-5. It can be positive, negative, or zero. If the integer is positive, the plus sign can be omitted; if it is negative, the minus sign must be present. An integer constant must not contain a comma. The range of an integer constant is $-(2^{59}-1)$ to $2^{59}-1$ ($2^{59}-1 = 576\,460\,752\,303\,423\,487$).

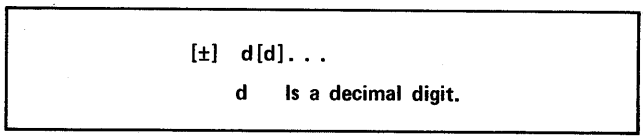


Figure 1-5. Integer Constant

Integers used in multiplication, division, and exponentiation, whether constant or variable, should be in the range $-(2^{48}-1)$ to $2^{48}-1$ ($2^{48}-1 = 281\,474\,976\,710\,655$). The result of such operations must also be in this range. For integer addition and subtraction (where both operands are integers), the full 60-bit word is used.

Examples:

```
237
-74
+136772
-0024
```

Examples of invalid integer constants:

46.	Decimal point not allowed.
23A	Letter not allowed.
7,200	Comma not allowed.

When an integer constant is used as a subscript, the maximum value is 2^{17}-1 (2^{17}-1=131071) and minimum is -(2^{17}-1) except when LCM=G is selected; the range then is -(2^{20}-8) through 2^{20}-8.

When an integer constant is used as an index in a DO statement or implied DO, the maximum value is 2^{17}-2 (2^{17}-2=131070) and the minimum value is -(2^{17}-2) except when DO=LONG is selected or a DO (LONG=1) directive is in effect; a DO index then can exceed 2^{17}-2.

DO and LCM are FORTRAN control statement parameters. They are described in section 11.

When values are converted (in an expression or assignment statement) from real to integer or from integer to real, the valid range is also from -(2^{48}-1) to 2^{48}-1. For values outside this range, the high order bits are lost and no diagnostic is provided.

REAL

A real constant consists of a string of decimal digits written with a decimal point or an exponent, or both, as shown in figure 1-6. Commas are not allowed. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

[±] **coeff**
[±] **coeff E** [±] **exp**
[±] **n E** [±] **exp**

coeff	**Is a coefficient in the form of a real constant:** **n.** **n.n** **.n**
n	**Is an unsigned integer constant.**
exp	**Is an unsigned integer exponent (base 10).**

Figure 1-6. Real Constant

The range of a real constant is 10^{-293} to 10^{+322}; if this range is exceeded, a diagnostic is printed. Precision is approximately 14 decimal digits, and the constant is stored internally in one computer word.

Examples:

7.5
-3.22
+4000.
.5

Examples of invalid real constants:

33,500. Comma not allowed.

2.5A Letter not allowed.

Optionally, a real constant can be followed by a decimal exponent, written as the letter E and an integer constant indicating the power of ten by which the number is to be multiplied. If the E is present, the integer constant following the letter E must not be omitted. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

Examples:

42.E1 Value 42. X 10^{1} = 420.

.00028E+5 Value .00028 X 10^{5} = 28.

6.205E6 Value 6.205 X 10^{6} = 6 205 000.

700.E-2 Value 700. X 10^{-2} = 7.

Example of invalid real constant:

7.2E3.4 Exponent not an integer.

DOUBLE PRECISION

A double precision constant is written in the same way as a real constant with exponent, except that the exponent is prefixed by the letter D instead of E, as shown in figure 1-7. Double precision values are represented internally by two computer words, giving additional precision. A double precision constant is accurate to approximately 29 decimal digits. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

[±] **coeff D** [±] **exp**
[±] **n D** [±] **exp**

coeff	**Is a coefficient in the form of a real constant:** **n.** **n.n** **.n**
n	**Is an unsigned integer constant.**
exp	**Is an unsigned integer exponent (base 10).**

Figure 1-7. Double Precision Constant

Examples:

5.834D2 Value 5.834 X 10^{2} = 583.4

14.D-5 Value 14. X 10^{-5} = .00014

9.2D03 Value 9.2 X 10^{3} = 9200.

3120D4 Value 3120. X 10^{4} = 31 200 000.

Examples of invalid double precision constants:

7.2D Exponent missing.

D5 Exponent alone not allowed.

2,001.3D2 Comma illegal.

3.14159265 D and exponent missing.

COMPLEX

Complex constants are written as a pair of real or integer constants or symbolic constants separated by a comma and enclosed in parentheses, as shown in figure 1-8.

(real,imag)	
real	**Is a real or integer constant for the real part.**
imag	**Is a real or integer constant for the imaginary part.**

Figure 1-8. Complex Constant

Examples:

(1, 7.54)	1. + 7.54i	$i=\sqrt{-1}$
(-2.1E1, 3.24)	-21. + 3.24i	
(4, 5)	4.0 + 5.0i	
(0., -1.)	0.0 - 1.0i	

```
PARAMETER (PAR1 = 1., PAR2 = 2.)
COMPLEX C
      .
      .
      .
C=(PAR1, PAR2)
```

The first constant represents the real part of the complex number, and the second constant represents the imaginary part. The parentheses are part of the constant and must always appear. Either constant can be preceded by a plus or minus sign. Complex values are represented internally by two consecutive computer words containing real values.

Examples of invalid complex constants:

(12.7D-4 16.1)	Comma missing and double precision not allowed.
4.7E + 2,1.942	Parentheses missing.

Real constants which form the complex constant can range from 10^{-293} to 10^{+322}. Division of complex numbers might result in underflow or overflow even when this range is not exceeded.

LOGICAL

A logical constant takes one of the two forms shown in figure 1-9. The periods are part of the constant and must appear.

.TRUE. .FALSE.	
.TRUE.	**Represents the logical value true.**
.FALSE.	**Represents the logical value false.**

Figure 1-9. Logical Constant

Examples:

.TRUE.
.FALSE.

.T. = -1 (non Ø)
.F. = Ø

Examples of invalid logical constants:

.TRUE	No terminating period.
.F.	Abbreviation not recognized.

BOOLEAN

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.

A Boolean constant is a Hollerith constant, octal constant, or hexadecimal constant. A Boolean constant is always represented in one computer word.

Hollerith

A Hollerith constant has one of the four forms shown in figure 1-10.

nHs L"s" R"s" "s"	
n	**Is an unsigned nonzero integer constant in the range $1 \leq n \leq 10$.**
s	**Is a string of 1 through 10 represented characters.**

Figure 1-10. Hollerith Constant

For the nHs form, the n specifies the number of characters in the string following the H. No more than ten characters can be specified in the string; extra characters are truncated. Blanks are significant, and characters that are not in the FORTRAN character set can be used.

The nHs form indicates left-justified with blank fill. Blank fill means that any unassigned character positions in the computer word are set to blank (display code 55_8).

Example:

2HAB Value $010255\ldots55_8$

The L"s" form indicates left-justified with binary zero fill. Binary zero fill means that any unassigned character positions are set to binary zero (display code 00_8).

Example:

L"AB" Value $010200\ldots00_8$

The R"s" form indicates right-justified with binary zero fill.

Example:

R"AB" Value $00\ldots000102_8$

The "s" form is equivalent to the nH form except the characters need not be counted. No more than 10 characters can be represented in the string. Any quote within the string is represented by two consecutive quote characters. Note that the string might be eleven characters long if one character is a quote represented by two consecutive quotes. Blanks are significant, and characters that are not in the FORTRAN character set can be used.

Examples:

"AB"	Value $010255...55_8$
"C""D"	Value $03640455...55_8$

Octal

An octal constant has the form shown in figure 1-11. An octal digit is one of the digits 0, 1, 2, 3, 4, 5, 6, or 7. The string of octal digits is interpreted as an octal number. As many as twenty octal digits can be represented in a 60-bit computer word. The octal number is right-justified with binary zero fill.

```
O"o"

    o   Is a string of 1 through 20 octal digits.
```

Figure 1-11. Octal Constant

Example:

O"77"	Value $00...0077_8$

Hexadecimal

A hexadecimal constant has the form shown in figure 1-12. A hexadecimal digit is one of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F. The string of hexadecimal (hex) digits is interpreted as a hexadecimal number. As many as fifteen hexadecimal digits can be represented in a 60-bit computer word. The hexadecimal number is right-justified with binary zero fill.

```
Z"z"

    z   Is a string of 1 through 15 hexadecimal digits.
```

Figure 1-12. Hexadecimal Constant

Example:

Z"1A"	Value $00...0032_8$

CHARACTER

A character constant has the form shown in figure 1-13. Apostrophes are used to enclose the character string. Within the character string, an apostrophe is represented by two consecutive apostrophes.

```
's'

    s   Is a string of characters.
```

Figure 1-13. Character Constant

The minimum number of characters in a character constant is one, and the maximum number of characters in a character constant is $(2^{15}-1)$ or 32 767. The length is the number of characters in the string. Blanks are significant in a character constant. Any characters in the operating system character set can be used.

Character positions in a character constant are numbered consecutively as 1, 2, 3, and so forth, up to the length of the constant. The length of the character constant is significant in all operations in which the constant is used. The length must be greater than zero.

Examples:

```
'ABC'
'123'
'YEAR''S'
```

Examples of invalid character constants:

'ABC	Terminating apostrophe is missing.
"ABC"	Not a character constant (valid Boolean constant).
'YEARS'S'	Invalid number of apostrophes.

VARIABLES

A variable represents a quantity with a value that can be changed repeatedly during program execution. Variables are identified by a symbolic name of one to six or seven letters or digits, beginning with a letter. A variable is associated with a storage location. Whenever a variable is used, it references the value currently in that location. A variable must be defined before being referenced for its value. The types of variables are integer, real, double precision, complex, Boolean, logical, and character. Variables typed by default are integer if the first letter is I, J, K, L, M, or N, and are real if the first letter is any other letter. Implicit and explicit typing of variables is described in section 2, Specification Statements.

INTEGER VARIABLES

An integer variable is a variable that is typed explicitly, implicitly, or by default as integer. An integer variable occupies one storage word. The range restrictions for integer variables are the same as for integer constants. See section 4 for restrictions on integers used in DO statements.

Examples:

```
ITEM1
NSUM
JSUM
N72
J
K2SO4
```

REAL VARIABLES

A real variable is a variable that is typed explicitly, implicitly, or by default as real. The value range is 10^{-293} through 10^{+322} with approximately 14 significant digits of precision. A real variable occupies one storage word.

Examples:

```
AVAR
SUM3
RESULT
TOTAL2
BETA
XXXX
```

DOUBLE PRECISION VARIABLES

A double precision variable is a variable that is typed explicitly or implicitly as double precision. The value of a double precision variable can range from 10^{-293} through 10^{+322} with approximately 29 significant digits of precision. Double precision variables occupy two consecutive storage words. The first word contains the more significant part of the number and the second contains the less significant part.

Example:

```
IMPLICIT DOUBLE PRECISION (A)

DOUBLE PRECISION OMEGA, X, IOTA
```

The variables OMEGA, X, IOTA, and all variables whose first letter is A are double precision.

COMPLEX VARIABLES

A complex variable is a variable that is typed explicitly or implicitly as complex. A complex variable occupies two storage words; each word contains a real number. The first word represents the real part of the number and the second represents the imaginary part.

Example:

```
COMPLEX ZETA, MU, LAMBDA
```

LOGICAL VARIABLES

A logical variable is a variable that is typed explicitly or implicitly as logical. A logical variable occupies one storage word.

Example:

```
LOGICAL L33, PRAVDA, VALUE
```

BOOLEAN VARIABLES

A Boolean variable is a variable that is typed explicitly or implicitly as Boolean. A Boolean variable occupies one storage word. Hollerith, octal, or hexadecimal values are generally assigned to Boolean variables.

Example:

```
BOOLEAN HVAL, ZZZ, R34
```

CHARACTER VARIABLES

A character variable is a variable that is typed explicitly or implicitly as character. The length of the character variable is specified when the variable is typed as character.

Example:

```
CHARACTER NAM*15, C3*3
```

ARRAYS

A FORTRAN array is a set of elements identified by a single name. The name is composed of one to seven letters and digits and begins with a letter. Each array element is referenced by the array name and a subscript. The type of the array elements is determined by the array name in the same manner as the type of a variable is determined by the variable name. The array name can be typed explicitly with a type statement, implicitly with an IMPLICIT statement, or by default typing. The array name and its dimensions must be declared in a DIMENSION, COMMON, or type statement.

When an array is declared, the declaration of array dimensions takes the form shown in figure 1-14. Arrays can have one through seven dimensions.

array (d[,d]. . .)

array	**Is the symbolic name of the array.**
d	**Specifies the bounds of an array dimension and takes the form:** **[lower:] upper**
lower	**Optionally specifies the lower bound of the dimension. The lower bound can be an integer or Boolean expression with a positive, zero, or negative value. If omitted, the lower bound is assumed to be 1.**
upper	**Specifies the upper bound of the dimension. The upper bound can be an integer or Boolean expression with a positive, zero, or negative value. The upper bound must be greater than or equal to the lower bound. In the case of an assumed-size array, the upper bound of the last dimension can be specified as ***

Figure 1-14. Declaration of Array Dimensions

The dimension bounds can be positive, negative, or zero. If the lower bound is omitted, the lower bound is assumed to be one. In this case, the upper bound must be positive. The general rule is that the upper bound must always be greater than or equal to the lower bound. The size of each dimension is indicated by the distance between the lower bound and upper bound. For example:

```
DIMENSION RX(0:5)
```

declares a 1-dimensional array of six elements such as that shown in figure 1-15.

```
DIMENSION TABLE(4,3)
```

declares a 2-dimensional array of four rows and three columns, for a total of twelve elements such as that shown in figure 1-16.

```
INTEGER STOR(6,6,3)
```

declares a 3-dimensional array of six rows, six columns and three planes, for a total of one hundred and eight elements.

The span of an array dimension is given by (u-ℓ+1) where u is the subscript upper bound and ℓ is the subscript lower bound. An array of type integer, Boolean, real, or logical occupies n words of storage, where n is the product of the spans of all dimensions. An array of type complex or double precision occupies 2*n words. An array of type character occupies (n*len+offset+9)/10 words, where len is the length in characters of an array element, and offset is the starting character position (0 to 9) of the array in the first word of the array storage.

An array in central memory must occupy less than 2^{17} words. An array in extended memory can occupy up to $2^{20}-8$ words if LCM=G is selected.

If a Boolean expression is used for the lower or upper bound of a dimension, the value of the expression is converted to integer; that is, the value is INT (lower) or INT (upper). A dimension bounds specification must not include a function reference or array element reference. Presence of a variable makes the size of the array adjustable. Presence of an asterisk as the upper bound of the last dimension makes the array an assumed-size array. An assumed-size array can only be used in a subroutine or function, as described under Procedure Communication in section 6.

ARRAY STORAGE

The elements of an array have a specific storage order, with elements of any array stored as a linear sequence of storage words. The first element of the array begins with the first storage word or character storage position, and the last element ends with the last storage word or character storage position.

The number of storage words reserved for an array is determined by the type of the array and its size. For real, integer, Boolean, and logical arrays, the number of storage words in an array equals the array size. For complex and double precision arrays, the number of storage words reserved is twice the array size. For character arrays, the number of words is calculated from the number of characters stored, at ten characters per storage word. For example, an array defined as CHARACTER*5 X(8), that is, eight 5-character elements, would require storage for 40 characters, or 4 storage words at offset zero.

Storage patterns for a 1-dimensional, 2-dimensional, and 3-dimensional array are shown in figure 1-15, figure 1-16, and figure 1-17, respectively. Arithmetic values are shown for the array elements, but an array can be any data type. Array elements are stored in ascending locations by columns. The first subscript value increases most rapidly, and the last subscript value increases least rapidly.

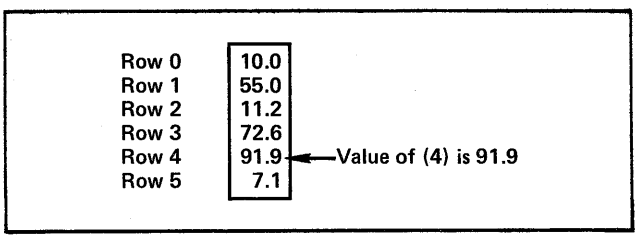


Figure 1-15. 1-Dimensional Array Storage

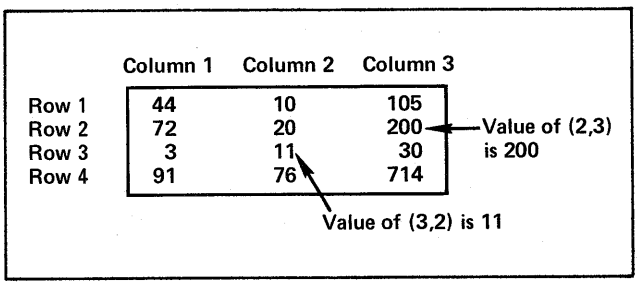


Figure 1-16. 2-Dimensional Array Storage

ARRAY REFERENCES

Array references can be references to complete arrays or to specific array elements. A reference to a complete array is simply the array name. A reference to a specific element involves the array name followed by a subscript specification. An array element reference is also called a subscripted array name.

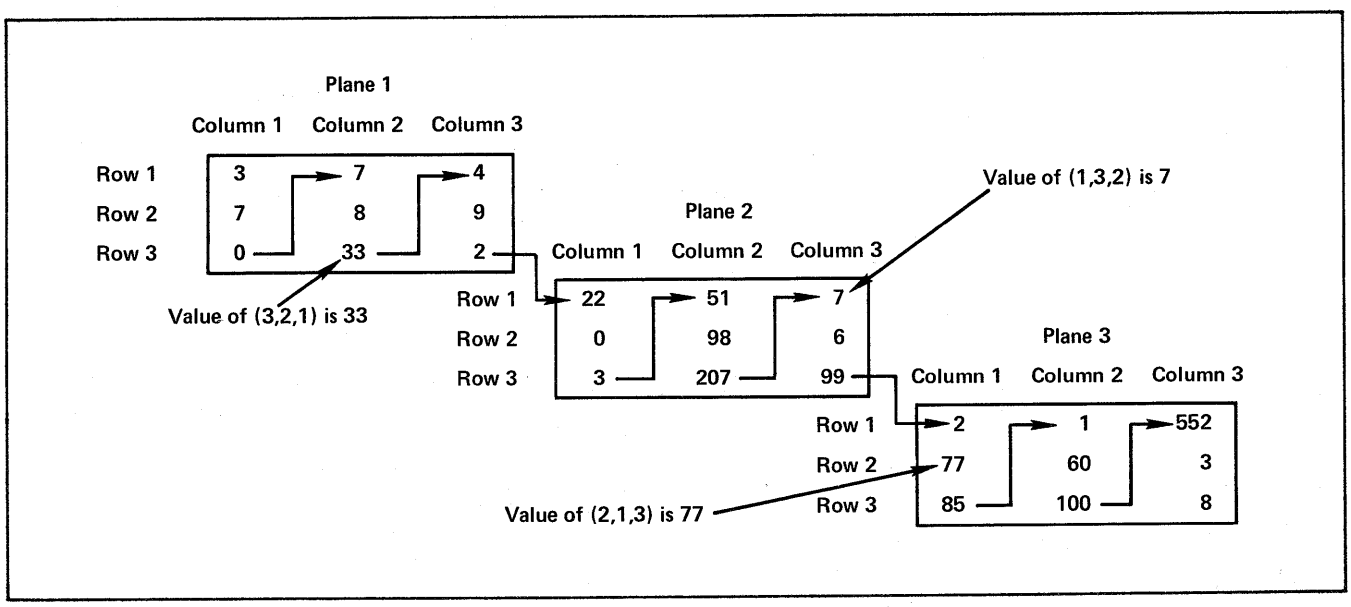


Figure 1-17. 3-Dimensional Array Storage

A reference to the complete array references all elements of the array in the order in which they are stored. For example:

```
DIMENSION XT(3)
DATA XT/1.,2.,3./
CALL CALC(XT)
```

uses the array reference XT in the DATA statement and the CALL statement.

A reference to an array element references a specific element and takes the form shown in figure 1-18.

array (e[,e]. . .)

array	**Is the symbolic name of the array.**
e	**Is a subscript expression that is an integer, real, double precision, complex, or Boolean expression. Each subscript expression has a value that is within the bounds of the corresponding dimension.**

Figure 1-18. Array Element Reference

An array element reference must specify a value for each dimension in the array. Array element references are not legal unless a value is supplied for each dimension. There can be up to seven dimensions in an array element.

An array element reference specifies the name of the array followed by a list of subscript expressions enclosed in parentheses. Each subscript expression can be an integer, real, double precision, complex, or Boolean expression. Each subscript expression is evaluated and converted as necessary to integer. A subscript expression can contain function references and array element references; however, evaluation of a function reference must not alter the value of any other subscript expression in the array element reference.

Each value after conversion to integer must not be less than the lower bound or greater than the upper bound of the dimension. If the array is an assumed-size array with the upper bound of the last dimension specified as asterisk, the value of the subscript expression must not exceed the actual size of the dimension. The results are unpredictable if an array element reference exceeds the size of an array. For each array element reference, evaluation of the subscript expressions yields a value for each dimension and a position relative to the beginning of the complete array.

The position of an array element is calculated as shown in table 1-2. The position indicates the storage location of an array element.

Example 1:

```
INTEGER DZ(12)
.
.
.
DZ(6)= 79
```

The array element reference DZ(6) refers to the element at position 6 in the array, that is, (1+(6-1)).

TABLE 1-2. ARRAY ELEMENT POSITION

Dimensions	Position of Array Element
1	$1 + (s_1-j_1)$
2	$1 + (s_1-j_1) + (s_2-j_2)*n_1$
3	$1 + (s_1-j_1) + (s_2-j_2)*n_1 + (s_3-j_3)*n_2*n_1$
. . .	
7	$1 + (s_1-j_1) + (s_2-j_2)*n_1 + (s_3-j_3)*n_2*n_1 + (s_4-j_4)*n_3*n_2*n_1 + (s_5-j_5)*n_4*n_3*n_2*n_1 + (s_6-j_6)*n_5*n_4*n_3*n_2*n_1 + (s_7-j_7)*n_6*n_5*n_4*n_3*n_2*n_1$

j_i Lower bound of dimension i.
k_i Upper bound of dimension i.
n_i Size of dimension i. If the lower bound is one, $n_i=k_i$. Otherwise, $n_i=(k_i-j_i+1)$.
s_i Value of the subscript expression specified for dimension i.

Example 2:

```
COMMON /CHAR/ CQ
CHARACTER*5 CQ(6,4)
.
.
.
CQ(6,3) = 'RUN'
```

The array element reference CQ(6,3) refers to the element at position 18, that is, (1+(6-1)+(3-1)*6). The character storage position is 86, that is, 1+(element position -1)*character length. Character position 86 indicates that storage for the element begins at the sixth character position in the ninth element of the array. (The COMMON declaration causes CQ to begin on a word boundary; in general, the compiler does not necessarily align character variables on word boundaries.)

CHARACTER SUBSTRINGS

When a character variable or character entity is declared, the entire character string can be defined and referenced. Specific parts of the character string can also be defined or referenced with character substring references. A character entity must be declared with the CHARACTER statement described in section 2. The declaration of a character entity specifies the length in characters.

SUBSTRING REFERENCES

If the name of a character entity is used in a reference, the value is the current value of the entire string.

Example:

```
CHARACTER*6 S1,S2
DATA S1/'STRING'/
S2 = S1
```

The reference to S1 is a reference to the full string 'STRING'. A reference to part of the string would be written as a character substring reference. A character substring reference has the form shown in figure 1-19.

char ([first] : [last])

char	**Is the name of a character variable or array and can be an array element reference.**
first	**Optionally specifies an integer, real, double precision, complex, or Boolean expression for the position of the first character of the substring. If first is omitted, the value is one.**
last	**Optionally specifies an integer, real, double precision, complex, or Boolean expression for the position of the last character in the substring. If last is omitted, the value is the length of the string.**

Figure 1-19. Character Substring Reference

The specification of the first character in the substring is an integer, real, double precision, complex, or Boolean expression that is evaluated and converted as necessary to integer. The expression can contain array element references and function references, but evaluation of a function reference must not alter the value of the other expression in the substring reference. If the specification of first is omitted, the value is one and all characters from one to the value of the specification of last are included in the substring.

The specification of last in the substring is an expression subject to the same rules as the specification of first. If last is omitted, the value is the length of the string and all characters from the specified first position to the end of the string are included in the substring. For a string length len, the values of first and last must be:

$$1 \leq first \leq last \leq len$$

For example, substring references to the string S1 with the value 'STRING' could be any of the following:

```
S1(1:3)     Value 'STR'
S1(3:4)     Value 'RI'
S1(4:)      Value 'ING'
S1(:4)      Value 'STRI'
S1(:)       Value 'STRING'
```

Note that the substring reference S1(:) has the same effect as the reference S1, since all characters in the string are referenced.

SUBSTRINGS AND ARRAYS

If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element followed by specification of the substring. For example:

```
CHARACTER*8 ZS(5)
CHARACTER*4 RSEN
.
.
.
ZS(4)(5:6)='FG'
RSEN=ZS(1)(:4)
```

The first reference refers to characters 5 and 6 in element 4 of array ZS. The second reference refers to the first four characters of the first element of array ZS.

STATEMENT ORDER

The order of various statements within the program unit is shown in table 1-3. Within each group, statements can be ordered as necessary, but the groups must be ordered as shown. Statements that can appear anywhere within more than one group are shown on the right in boxes that extend vertically across more than one group.

A PROGRAM statement can appear only as the first statement in a main program. The first statement of a subroutine, function, or block data subroutine is respectively a SUBROUTINE statement, FUNCTION statement, or BLOCK DATA statement. The END statement is the last statement of each of the preceding program units.

If a program is to be used as an overlay, the OVERLAY statement must precede the PROGRAM statement and any FUNCTION or SUBROUTINE statements.

Comments can appear anywhere within the program unit. Note that any comment following the END statement is considered part of the next program unit.

FORMAT statements can appear anywhere in the program unit.

ENTRY statements can appear anywhere in the program unit, subject to two restrictions. An ENTRY statement cannot appear within the range of a DO loop (between the DO statement and the terminating statement) or within a block IF construction (between the IF statement and the ENDIF statement). The ENTRY statement cannot be used in the main program unit, where an alternate entry point would have no meaning.

Specification statements in general precede the executable statements in the program unit. The nonexecutable specification statements describe characteristics of quantities known in the program unit, and the executable statements describe the actions to be taken.

All specification statements must precede all DATA statements, NAMELIST statements, statement function definitions, and executable statements. Within the specification statements, all IMPLICIT statements must precede all other specification statements except PARAMETER statements. PARAMETER statements can appear anywhere among the specification statements, but each PARAMETER statement must precede any

references to the symbolic constant defined by the PARAMETER statement.

All statement function definitions must precede all executable statements in the program unit. Statement function definitions cannot be used in block data subroutines.

DATA statements can be used anywhere among statement function definitions and executable statements.

NAMELIST statements can appear anywhere among statement function definitions and executable statements. Note that each NAMELIST statement defining a NAMELIST group must appear before the first reference to the NAMELIST group. Also note that NAMELIST statements cannot be used in block data subroutines.

Executable statements must follow all specification statements and any statement function definitions. Executable statements such as assignment, flow control, or I/O statements can appear in whatever order required in the program unit. Executable statements cannot be used in block data subroutines.

The END statement must be the last statement of each program unit.

TABLE 1-3. STATEMENT ORDER

Statement				
PROGRAM†, SUBROUTINE, FUNCTION, or BLOCK DATA				Comments and compiler directives
IMPLICIT		PARAMETER (must precede first reference)	FORMAT††	ENTRY††† (except within range of block IF or DO loop)
INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, BOOLEAN (Type specification statements)				
DIMENSION, EQUIVALENCE, COMMON, LEVEL, SAVE, EXTERNAL, INTRINSIC (Specification statements)				
Statement function definition††	DATA	NAMELIST†† (must precede first reference)		
Assignment, DO, CONTINUE, IF, ELSE, ELSEIF, ENDIF, GOTO, ASSIGN, CALL, RETURN, PAUSE, STOP (Executable statements)††				
OPEN, CLOSE, INQUIRE, READ, WRITE, PRINT, PUNCH, BUFFER IN, BUFFER OUT, ENCODE, DECODE, REWIND, BACKSPACE, ENDFILE (Executable I/O statements)††				
END				

†Can be preceded by an OVERLAY statement.
††Cannot be used in a BLOCK DATA subprogram.
†††Cannot be used in a main program or BLOCK DATA subprogram.

SPECIFICATION STATEMENTS 2

Specification statements are nonexecutable and are used to specify the characteristics of symbolic names used in the program. Specification statements must appear before all DATA statements, NAMELIST statements, statement function definitions, and executable statements in the program unit.

DATA statements are not specification statements but are described in this section.

The specification statements are:

- IMPLICIT
- DIMENSION
- PARAMETER
- EQUIVALENCE
- COMMON
- LEVEL
- SAVE
- EXTERNAL
- INTRINSIC
- Type (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, CHARACTER)

The IMPLICIT and type statements are used to specify the data type of variables. Default typing of variables takes place unless the IMPLICIT statement or the type statements are used to change the data type of specific variables. Any IMPLICIT statements must precede all other specification statements, except PARAMETER statements.

The DIMENSION statement is used to specify the number of dimensions in an array and the bounds for each dimension.

The PARAMETER statement is used to give a symbolic name to a constant. PARAMETER statements can be used anywhere among the specification statements, but each symbolic constant must be defined in a PARAMETER statement before the first reference to the symbolic constant.

The EQUIVALENCE, COMMON, and LEVEL statements are used to define the storage characteristics of variables, or to define whether storage can be shared.

The SAVE statement is used to preserve the values of variables after execution of a RETURN or END statement in a subprogram. Variables that would become undefined remain defined and can be used in any subsequent executions of the same subprogram.

The EXTERNAL and INTRINSIC statements are used to control the recognition of function names. The EXTERNAL statement specifies that a function name refers to a user-written function rather than an intrinsic function. The INTRINSIC statement specifies that a function name refers to an intrinsic function rather than a user-written function.

If any specification statement appears after the first executable statement, DATA statement, NAMELIST statement, or statement function definition, a fatal diagnostic is issued.

DATA statements are used to give initial values to variables. DATA statements must appear after all specification statements in the program unit. DATA statements can appear anywhere among the statement function definitions and executable statements. Usually, DATA statements are placed after the specification statements but before the statement function definitions and executable statements. A variable is considered undefined until a value is assigned with a DATA statement, input statement, or assignment statement. A variable must be defined before the first reference to the value of the variable.

TYPE STATEMENTS

Each variable, array, symbolic constant, statement function, or external function name has a type. Entities can be typed as integer, real, double precision, complex, Boolean, logical, or character. The name of a main program, subroutine, or block data subroutine cannot be typed.

Default typing occurs in the absence of any implicit typing or explicit typing. The type of the symbolic name is implied by the first character of the name. The letter I, J, K, L, M, or N implies type integer, and any other letter implies type real.

Implicit typing is controlled by the IMPLICIT statement. The IMPLICIT statement specifies a different typing according to the first character of each name. One or more IMPLICIT statements can be included in each program unit.

Explicit typing defines the types of individual names. The INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, or CHARACTER statements are explicit type statements. An explicit type statement can also be used to supply dimension information for an array.

Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name. Intrinsic functions are described in section 7.

INTEGER STATEMENT

The INTEGER statement shown in figure 2-1 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type integer.

Examples:

```
INTEGER ITEM1, NSUM, JSUM
INTEGER A72, H2SQ4
INTEGER M5(2)
```

INTEGER name[,name]. . .

name	**Is explicitly typed as integer. Each name is one of the forms:**
	var
	array [(d[,d]. . .)]
var	**Is a variable, function name, or function entry.**
array	**Is an array name.**
d	**Specifies the bounds of a dimension.**

Figure 2-1. INTEGER Statement

REAL STATEMENT

The REAL statement shown in figure 2-2 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type real.

Examples:

```
REAL IVAR, NSUM3, RESULT
REAL TOTAL2, BETA, XXXX
REAL TR(10, 5)
```

REAL name[,name]. . .

name	**Is explicitly typed as real. Each name is one of the forms:**
	var
	array [(d[,d]. . .)]
var	**Is a variable, function name, or function entry.**
array	**Is an array name.**
d	**Specifies the bounds of a dimension.**

Figure 2-2. REAL Statement

DOUBLE PRECISION STATEMENT

The DOUBLE PRECISION statement shown in figure 2-3 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type double precision.

Examples:

```
DOUBLE PRECISION DPROD, DEIGV
DOUBLE PRECISION RMAT(10, 10)
```

DOUBLE PRECISION name[,name]. . .

name	**Is explicitly typed as a double precision. Each name is one of the forms:**
	var
	array [(d[,d]. . .)]
var	**Is a variable, function name, or function entry.**
array	**Is an array name.**
d	**Specifies the bounds of a dimension.**

Figure 2-3. DOUBLE PRECISION Statement

COMPLEX STATEMENT

The COMPLEX statement shown in figure 2-4 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type complex.

Examples:

```
COMPLEX CPVAR
COMPLEX RES(5, 5)
```

COMPLEX name[,name]. . .

name	**Is explicitly typed as a complex. Each name is one of the forms:**
	var
	array [(d[,d]. . .)]
var	**Is a variable, function name, or function entry.**
array	**Is an array name.**
d	**Specifies the bounds of a dimension.**

Figure 2-4. COMPLEX Statement

BOOLEAN STATEMENT

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.

The BOOLEAN statement shown in figure 2-5 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type Boolean.

Examples:

```
BOOLEAN ALABEL, QMASK
BOOLEAN LABEL(14)
```

BOOLEAN name[,name]. . .

name	Is explicitly typed as Boolean. Each name is one of the forms: var array [(d[,d]. . .)]
var	Is a variable, function name, or function array.
array	Is an array name.
d	Specifies the bounds of a dimension.

Figure 2-5. BOOLEAN Statement

LOGICAL STATEMENT

The LOGICAL statement shown in figure 2-6 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type logical.

Example:

```
LOGICAL SWITCH, TEST
```

LOGICAL name[,name]. . .

name	Is explicitly typed as logical. Each name is one of the forms: var array [(d[,d]. . .)]
var	Is a variable, function name, or function entry.
array	Is an array name.
d	Specifies the bounds of a dimension.

Figure 2-6. LOGICAL Statement

CHARACTER STATEMENT

The CHARACTER statement shown in figure 2-7 can be used to define a variable, array, symbolic constant, function name, or dummy procedure name as type character.

A length specification immediately following the word CHARACTER applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. Note that for an array, the length specified is for each array element. If a length is not specified for an entity, either explicitly or by an IMPLICIT statement, the length is one. The unit of length for CHARACTER is characters.

The length specification for a variable or array declared in a CHARACTER statement must be an unsigned nonzero integer constant, or an integer constant expression.

Example:

```
CHARACTER A*3, B(10)*(12+3*2)
```

CHARACTER[*len] [,] name[,name]. . .

name	Is explicitly typed as character. Each name is one of the forms: var [*len] array [(d[,d]. . .)] [*len]
len	Specifies the length and can be: an unsigned nonzero integer constant; an integer constant expression, enclosed in parentheses, with a positive value; or an asterisk enclosed in parentheses.
var	Is a variable, function name, or function entry.
array	Is an array name.
d	Specifies the bounds of a dimension.

Figure 2-7. CHARACTER Statement

The example defines a character variable A that is 3 characters long; and a character array B that has 10 elements, each of which is 18 characters long.

If a dummy argument has the length (*) specified, the dummy argument assumes the length of the associated actual argument for each reference to the subroutine or function. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of each array element in the associated actual argument.

Example:

```
PROGRAM MN
CHARACTER *3 CC, A(4)
.
.
.
CALL TSUB (CC, A(1)(2:3))
.
.
.
SUBROUTINE TSUB (CHAR, Z)
CHARACTER *(*) CHAR, Z(4)
```

The dummy argument CHAR in subroutine TSUB will have length 3 and each element of the array Z will have length 2.

If an external function has the length (*) specified in a function subprogram, the function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function has the length specified in the referencing program unit.

The length specified for a character function, in the program unit that references the function, must be an integer constant or integer constant expression and must agree with the length specified in the function. Note that there is always agreement of length if the length (*) is specified in the function.

If a symbolic constant of type character has the length (*) specified, the constant has the length of its corresponding constant expression in a PARAMETER statement. If the length specification is a symbolic constant, it must be enclosed in parentheses.

Example:

```
PARAMETER (N=5)
CHARACTER *(N) AB
```

If the parentheses are omitted, the compiler cannot distinguish between the length specification and the variable name. (Blanks do not function as delimiters, and an error message is issued.)

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

Example:

```
CHARACTER*10 ASTR, ABC(5), XR*20
```

The variable ASTR and each element of the array ABC have the length 10. The variable XR has the specified length of 20.

Example:

```
CHARACTER AR*5, BR*8
.
.
.
CALL ZC(BR)
.
.
.
SUBROUTINE ZC(STR)
CHARACTER STR*(*)
```

In the example, the variable STR has the length 8 when subroutine ZC is called. If subroutine ZC is called with variable AR passed, the variable STR has the length 5. Note that the length is not directly known, and certain types of reference to STR cannot be used. See Procedure Communication in section 6.

Character substrings are described in section 1.

IMPLICIT STATEMENT

The IMPLICIT statement can be used to change or confirm the default typing according to the first letters of the names. The IMPLICIT statement is shown in figure 2-8.

IMPLICIT type(ac[,ac] . . .) [,type(ac[,ac]. . .)]. . .

type	Is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, CHARACTER, or CHARACTER[*len].
ac	Is a single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.
len	Specifies the length and can be an unsigned nonzero integer constant; or an integer constant expression, enclosed in parentheses, with a positive value.

Figure 2-8. IMPLICIT Statement

The statement specifies the type of variables, arrays, symbolic constants, and functions beginning with the letters ac. The IMPLICIT statements in a program unit must precede all other specification statements except PARAMETER statements. An IMPLICIT statement in a function or subroutine subprogram affects the type associated with dummy arguments and the function name, as well as other variables in the subprogram. Explicit typing of a variable name or array element in a type statement or FUNCTION statement overrides an IMPLICIT specification.

The specified single letters or ranges of letters specify the entities to be typed. A range of letters has the same effect as writing a list of the single letters within the range. The same letter can appear as a single letter, or be within a range of letters, only once in all IMPLICIT statements in a program unit.

The length can be specified implicitly for entities of type character. If length is not specified, the length is one. The length can be specified as an unsigned nonzero integer constant, or an integer constant expression, enclosed in parentheses, with a positive value. The specified length applies to all entities implicitly typed as character.

Example:

```
IMPLICIT CHARACTER*20 (M, X-Z)
```

The default typing is effective in all cases except for names beginning with the letters M, X, Y, or Z. Names beginning with M are typed as character rather than integer, and names beginning with X, Y, or Z are character rather than real.

Note that any explicit typing with a type statement is effective in overriding both the default typing and any implicit typing.

Example:

```
IMPLICIT LOGICAL (L)
INTEGER L, LX, TT
```

Names beginning with L are typed as logical rather than integer. Names L and LX are explicitly typed as integer and are not affected by the implicit typing. The name TT is explicitly typed as integer and does not take the default type real.

DIMENSION STATEMENT

The DIMENSION statement shown in figure 2-9 defines symbolic names as array names and specifies the bounds of each array. More than one array can be declared in a single DIMENSION statement. Dummy argument arrays specified within a procedure subprogram can have adjustable dimension specifications. A further explanation of adjustable dimension specifications appears under Procedure Communication in section 6.

Within the same program unit, only one definition of an array is permitted. Note that dimension information can be specified in COMMON statements and type statements. The dimension information defines the array dimensions and the bounds for each dimension.

The description of arrays is in section 1. The description covers the properties of arrays, the storage of arrays, and array references.

Example:

```
REAL NIL
DIMENSION NIL(6, 2, 2)
```

DIMENSION array(d[,d]. . .) [,array(d[,d]. . .)]. . .

array	Is an array name.
d	Specifies the bounds of a dimension in one of the forms: upper lower:upper
upper	Is the upper bound of the dimension and is a dimension bound expression in which all constants, symbolic constants, and variables are type integer or Boolean.
lower	Is the lower bound of the dimension and is a dimension bound expression in which all constants, symbolic constants, and variables are of type integer or Boolean. If only the upper bound is specified, value of the lower bound is one.

Figure 2-9. DIMENSION Statement

These statements could be combined into one statement with 24 real elements declared for array NIL:

```
REAL NIL(6, 2, 2)
```

Example:

```
COMPLEX BETA
DIMENSION BETA(2, 3)
```

BETA is an array containing six complex elements.

Example:

```
CHARACTER*8 XR
DIMENSION XR(0:4)
```

XR is an array containing five character elements, and each element has a length of eight characters. A reference to the third and fourth characters of the second element would be XR(1)(3:4).

PARAMETER STATEMENT

The PARAMETER statement shown in figure 2-10 is used to give a symbolic name to a constant.

PARAMETER (p=e [,p=e]. . .)

p	Is a symbolic name.
e	Is a constant, constant expression, or extended constant expression.

Figure 2-10. PARAMETER Statement

An extended constant expression is an expression in which each primary is one of the following:

- A constant
- A previously-defined symbolic constant
- A reference to one of the following intrinsic functions with extended constant expression arguments:

ABS	DIM	MASK
AIMAG	DINT	MAX
AINT	DNINT	MAX0
AMAX0	DMAX1	MAX1
AMAX1	DMIN1	MIN
AMIN0	DMOD	MIN0
AMIN1	DPROD	MIN1
AMOD	DSIGN	MOD
AND	EQV	NEQV
ANINT	FLOAT	NINT
BOOL (not character)	IABS	OR
CMPLX	IDIM	REAL
COMPL	IDINT	SHIFT
CONJG	IDNINT	SIGN
DABS	IFIX	SNGL
DBLE	INT	XOR
DDIM	ISIGN	

- An extended constant expression enclosed in parentheses

If a symbolic name is of type integer, real, double precision, complex, or Boolean, the corresponding expression must be an arithmetic or Boolean constant expression or an extended constant expression. If the symbolic name is of type character or logical, the corresponding expression must be a character constant expression, logical constant expression, or extended constant expression. Each symbolic name becomes defined with the value of the expression that appears to the right of the equals, according to the rules for assignment. Any symbolic constant that appears in an expression e must have been previously defined in the same or a different PARAMETER statement in the program unit.

A symbolic name of a constant can be defined only once in a program unit, and can identify only the corresponding constant. The type of a symbolic constant can be specified by an IMPLICIT statement or type statement before the first appearance of the symbolic constant in a PARAMETER statement. If the length of a symbolic character constant is not the default length of one, the length must be specified in an IMPLICIT statement or type statement before the first appearance of the symbolic constant. The easiest way to do this is to explicitly type the symbolic constant as character with length (*). The actual length of the constant is determined by the length of the string defining it in the PARAMETER statement. The length must not be changed by another IMPLICIT statement or by subsequent statements.

Once defined, a symbolic constant can appear in the program unit in the following ways:

- In an expression in any subsequent statement
- In a DATA statement as an initial value or a repeat count
- In a complex constant as the real or imaginary part
- In a LEVEL statement as the storage level
- In a C$ directive as a primary in an expression, or as a parameter value

A symbolic constant cannot appear in a format statement.

Examples:

```
PARAMETER (ITER= 20, START= 5)
CHARACTER CC*(*)
PARAMETER (CC= '(I4, F10.5)')
.
.
.
DATA COUNT /START/
.
.
.
DO 410 J= 1, ITER
.
.
.
READ CC, IX, RX
```

The symbolic constant START is used to assign an initial value to variable COUNT, the symbolic constant ITER is used to control the DO loop, and the symbolic constant CC is used to specify a character constant format specification.

COMMON STATEMENT

The COMMON statement shown in figure 2-11 provides a means of associating entities in different program units. The use of common blocks enables different program units to define and reference the same data without using arguments, and to share storage units. Within a program unit, an entity in a common block is known by a specific name. Within another program unit, the same data can be known by a different symbolic name with the scope of that program unit.

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

cb	Is a common block name identifying a named common block containing the entities in nlist. If the name is omitted, the nlist entities are in blank common.
nlist	Is a list of entities to be included in the common block. The entities are separated by commas and can take the form: var array array (d[,d]. . .)
var	Is a variable.
array	Is an array name.
d	Specifies the bounds of an array dimension.

Figure 2-11. COMMON Statement

A single variable name or array name can appear only once in any COMMON statement within the program unit. Function or entry names cannot be included in common blocks. In a subprogram, names of dummy arguments cannot be included in common blocks.

If the common block name is omitted, the common block is blank common. When the first specification in the COMMON statement is for blank common, the slashes can also be omitted. If a common block name is specified, the common block is a named common block. Within a program unit, declarations of common blocks are cumulative. The nlist following each successive appearance of the common block name (or no name for blank common) adds more entities to the common block and is treated as a continuation of the specification. Variables and arrays are stored in the order in which they appear in the specification.

If any character variable or character array is included in a common block, all entities in the common block must be type character. Note that since a common block name has the scope of the executable program, the common block name can be used within a program unit as a variable or array name, without conflict.

The maximum number of common blocks in an executable program, including blank common and all named common, is 500. The maximum size of each common block is 131 071 storage words (for character data, 1 310 710 characters). The use of ECS/LCM residence and LCM=G for any common block increases the maximum possible size to 1 048 568 storage words (for character data, 10 485 680 characters).

The actual size of any common block is the number of storage words required for the entities in the common block, plus any extensions associated with the common block by EQUIVALENCE statements. Extensions can only be made by adding storage words at the end of the common block. See the description of the EQUIVALENCE statement in this section. A blank common block can be treated as having a different size in separate program units. The length of a common block, other than blank common, must not be increased by a subprogram using the block unless the subprogram is loaded first. If a program unit does not use all locations reserved in a common block, unused variables can be inserted in the COMMON declaration to ensure proper correspondence of common areas. A common block must have the same level in all routines declaring it (see LEVEL statement).

Entities in named common blocks can be initially defined by a DATA statement in a block data subprogram, or by a DATA statement in any program unit. Entities in blank common cannot be initially defined. After an entity in a named common block has been initially defined, the value is available to any subprogram in which the named common block appears.

Entities in blank common remain defined at all times and do not become undefined on execution of a return from a subprogram. Entities in named common can become undefined on execution of a return from a subprogram, unless the SAVE statement is used. See the description of the SAVE statement in this section.

Example:

```
COMMON A, B
COMMON /XT/ C, D, E
.
.
.
SUBROUTINE P(Q, R)
COMMON /XT/ F, G, H
.
.
.
FUNCTION T(U)
COMMON Y, Z
```

The entities C, D, and E in the main program are in the common block named XT. The same storage words are known by the names F, G, and H in subroutine P. The entities A and B in the main program are in blank common. The same storage words are known by the names Y and Z in function T.

Example:

```
COMMON JCOUNT
.
.
.
JCOUNT= 6
.
.
.
FUNCTION AB(A)
COMMON /C/ STX(4)
DATA STX/1., 2., 2., 1./
```

Since an entity in blank common cannot be initially defined with a DATA statement, an assigment statement must be used to define the value of JCOUNT. In function AB, a DATA statement can be used to define initial values for the elements of array STX in the common block named C. Note that JCOUNT is not common to function AB.

Example:

```
CHARACTER*15 D, E
COMMON /CVAL/ D, E
DATA D, E/'TEST', 'PROD'/
```

The common block named CVAL contains character variables. The variables D and E are initially defined in a DATA statement.

Example:

```
COMMON /SUM/ A, B(20)
.
.
.
SUBROUTINE GR
COMPLEX FR(10)
COMMON /SUM/ X, FR
```

The common block SUM in the main program is declared to contain the variable A and the array B. In the subroutine GR, the same storage words are associated with X and the array FR. Even if X is not used in the subroutine, X holds the place so that array FR matches the placement of array B. Note also that array FR is complex. The elements B(1) and B(2) are known in GR as FR(1); B(3) and B(4) are FR(2); and so forth. Each specification of common block SUM accounts for 21 storage words.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement shown in figure 2-12 can be used to specify the sharing of storage by two or more entities in a program unit. Equivalencing causes association of the entities that share the storage. Equivalencing associates entities within a program unit, and common blocks associate entities across program units. Equivalencing and common can interact.

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

nlist	**Is a list of variable names, array names, array element names, or character substring names. The names are separated by commas.**

Figure 2-12. EQUIVALENCE Statement

If the equivalenced entities are of different data types, equivalencing does not cause type conversion. If a variable and an array are equivalenced, the variable does not acquire array properties and the array does not lose the properties of an array. An entity of type character can be equivalenced only to another entity of type character. The lengths of the equivalenced character entities can be different.

Each nlist specification must contain at least two names of entities to be equivalenced. In a subprogram, names of dummy arguments cannot appear in the list. Function and entry names cannot be included in the list. Equivalencing specifies that all entities in the list share the same first storage word. For character entities, equivalencing specifies that all entities in the list share the same first character storage position. Equivalencing can indirectly cause the association of other entities, for instance when an EQUIVALENCE statement interacts with a COMMON statement.

If an array element is included in nlist, the number of subscript expressions must match the number of dimensions declared for the array name. If an array name appears in the list, the effect is as if the first element of the array had been included in the list. Any subscript expression must be an integer or Boolean constant expression. For character entities, any substring expression must be an integer or Boolean constant expression.

Example:

```
DIMENSION Y(4), B(3, 2)
EQUIVALENCE (Y(1), B(3, 1))
EQUIVALENCE (X, Y(2))
```

Storage is shared so that 6 storage words are needed for Y, B, and X. The associations are:

```
        B(1, 1)
        B(2, 1)
Y(1)    B(3, 1)
Y(2)    B(1, 2)     X
Y(3)    B(2, 2)
Y(4)    B(3, 2)
```

Example:

```
CHARACTER A*5, C*3, D(2)*2
EQUIVALENCE (A, D(1)), (C, D(2))
```

Storage is shared so that 5 character storage positions are needed for A, C, and D. The associations are:

```
A(1:1)  D(1)(1:1)
A(2:2)  D(1)(2:2)
A(3:3)  D(2)(1:1)   C(1:1)
A(4:4)  D(2)(2:2)   C(2:2)
A(5:5)              C(3:3)
```

Variables of different data types can be equivalenced, except for character data. The equivalencing associates the first storage word of each entity. For example:

```
REAL TR(4)
COMPLEX TS(2)
EQUIVALENCE (TR, TS)
```

causes the following associations:

```
TR(1)   TS(1)-real part
TR(2)   TS(1)-imaginary part
TR(3)   TS(2)-real part
TR(4)   TS(2)-imaginary part
```

Equivalencing must not reference array elements in such a way that the storage sequence of the array would be altered. The same storage unit cannot be specified as occurring more than once in the storage sequence. For example:

```
REAL FA(3)
EQUIVALENCE (FA(1), B), (FA(3), B)
```

would be illegal. Also, the normal storage sequence of array elements cannot be interrupted to make consecutive storage words no longer consecutive. For example:

```
REAL BZ(7), CZ(5)
EQUIVALENCE (BZ, CZ), (BZ(3), CZ(4))
```

would also be illegal.

The interaction of COMMON and EQUIVALENCE statements is restricted in two ways:

- An EQUIVALENCE statement must not attempt the association of two different common blocks in the same program unit. For example:

  ```
  COMMON /LT/ A, T
  COMMON /LX/ S, R
  EQUIVALENCE (T, S)
  ```

  is not legal.

- An EQUIVALENCE statement must not cause a common block to be extended by adding storage words before the first storage word of the common block. On the other hand, a common block can be extended through equivalencing if storage words are added at the end of the common block. For example:

  ```
  COMMON /X/ A
  REAL B(5)
  EQUIVALENCE (A, B(4))
  ```

  is not legal, whereas:

  ```
  COMMON /X/ A
  REAL B(5)
  EQUIVALENCE (A, B(1))
  ```

  can be used to extend the common block.

LEVEL STATEMENT

The LEVEL statement shown in figure 2-13 provides a means for specifying the storage level of common blocks and dummy arguments. The storage level indicates the storage residence and mode of access for entities in a common block or for actual arguments associated with dummy arguments. Only common block names and dummy argument names can appear in a LEVEL statement. No dimension or type information can be included in the LEVEL statement.

LEVEL n,name[,name]. . .

n	**Is an unsigned integer constant, or symbolic constant, with the value 0, 1, 2, or 3 indicating the storage level.**
name	**Is either a common block designator of the form /[cb]/ or a dummy argument name. If n is 0, only dummy argument names can appear.**

Figure 2-13. LEVEL Statement

Storage residence is either central memory or extended memory. Central memory (CM) is also known as small central memory (SCM). Extended memory is either extended central storage (ECS) or large central memory (LCM).

Mode of access is either unrestricted or restricted. Restricted access for a common block entity means access only in a DATA statement or as an actual argument to an external procedure or LOCF. Restricted access for a dummy argument means access only as an actual argument to an external procedure or LOCF.

Storage level 1 indicates central memory residence. A common block that does not appear in any LEVEL statement in any program unit is at storage level 1 in each program unit of the program. A dummy argument that does not appear in a LEVEL statement is at storage level 1. Mode of access for level 1 entities is unrestricted.

Storage level 2 indicates the following residence:

- Central memory residence on CDC CYBER 170 Models 171, 172, 173, 174, and 175; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series computers and SCM on CYBER 176 without LCM.
- Large central memory residence on CDC CYBER 170 Model 176, CYBER 70 Model 76, and 7600 computers.

Mode of access for level 2 entities is unrestricted.

Storage level 3 indicates extended memory residence. Mode of access for level 3 entities is restricted to COMMON, DIMENSION, type, EQUIVALENCE, DATA, CALL, SUBROUTINE, and FUNCTION statements. Level 3 items cannot be used in expressions.

Storage level 0 for a dummy argument indicates that the storage level of each associated actual argument is either 1 or 2. Mode of access for level 0 entities is unrestricted. For a dummy argument of level 1, 2, or 3, each associated actual argument (except an actual argument of level 0) must have the same level as the dummy argument.

A common block designator that appears in a LEVEL statement in a program unit must appear in a LEVEL statement with the same level number in each program unit in which the common block appears.

If the storage level is multiply defined, the first declared level is used and a warning diagnostic is printed.

Example:

```
DIMENSION E(500), B(500), CM(1000)
LEVEL 3,/ECSBLK/
COMMON /ECSBLK/ E, B
.
.
.
CALL MOVLEV(CM, E, 1000)
```

The LEVEL statement allocates common block ECSBLK to extended memory. Arrays E and B are in the common block ECSBLK. The library routine MOVLEV moves 1000 words of central memory to the two arrays E and B in extended memory, starting from location CM, which is the first word address of array CM.

For more information about storage levels, see the descriptions of the MOVLEV subroutine and LOCF function in section 7, FORTRAN Supplied Procedures.

SAVE STATEMENT

The SAVE statement shown in figure 2-14 is used to retain the definition status of entities after the execution of a RETURN or END statement in a subprogram. A SAVE statement in a main program is optional and has no effect. A SAVE statement does not have the effect of retaining the definition status of an entity after the execution of a RETURN or END statement in the main program unit of an overlay.

SAVE [a[,a]. . .]

a — **Is a variable name, array name, or common block name enclosed in slashes. Redundant appearances are not permitted.**

Figure 2-14. SAVE Statement

Dummy argument names, procedure names, and names of entities in a common block must not appear in the SAVE statement. A common block name (or // indicating blank common) has the effect of specifying all of the entities in the common block. A SAVE statement with no list is treated as though it contained the names of all allowable items in the program unit. If a common block name is specified in a SAVE statement in a subprogram, the common block name must be specified by a SAVE statement in every subprogram in which the common block appears.

Execution of a RETURN statement or an END statement within a subprogram causes the entities within the subprogram to become undefined, except in the following cases:

- Entities specified by SAVE statements do not become undefined.
- Entities in blank common do not become undefined.
- Entities that have been initially defined (and not redefined) do not become undefined.
- Entities in a named common block that appears in the subprogram and in the referencing program unit do not become undefined.

If a local variable or array that is specified in a SAVE statement and is not in a common block is defined in a subprogram at the time a RETURN or END statement is executed, that variable or array remains defined with the same value at the next reference to the subprogram.

Within a subprogram, an entity in a common block can be defined or undefined, depending on the definition status of the associated storage. If a named common block is specified in a SAVE statement in a subprogram and the entities in the common block are defined, the common block storage remains defined at the time a RETURN or END statement is executed and is available to the next program unit that specifies the named common block. The common block storage can become undefined or redefined in another program unit.

Example:

```
COMMON /C1/ G, H
SAVE /C1/
CALL XYZ
.
.
.
SUBROUTINE XYZ
COMMON A, D, F
COMMON /C1/ GVAL, HVAL
SAVE
DATA JCOUNT /5/
X= 6.5
.
.
.
RETURN
END
```

The SAVE statement in subroutine XYZ has the effect of saving the value of X as 6.5 for any later invocations of the subroutine. Saving of certain other values does not depend on the presence of the SAVE statement. The three entities in blank common remain defined. The two entities in common block C1 remain defined because common block C1 appears in the referencing program unit. Finally, since JCOUNT is initially defined and not redefined in the subroutine, JCOUNT remains defined for any later invocations of the subroutine.

EXTERNAL STATEMENT

The EXTERNAL statement shown in figure 2-15 is used to identify a name as representing an external procedure and to permit such a name to be used as an actual argument.

EXTERNAL proc[,proc]. . .

proc — **Is the name of an external procedure, dummy procedure, or block data subprogram.**

Figure 2-15. EXTERNAL Statement

Only one appearance of a symbolic name in all of the EXTERNAL statements of a program unit is permitted. If an external procedure name is an actual argument in a program unit, it must appear in an EXTERNAL statement in the program unit. A statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, the name becomes the name of some external procedure. The intrinsic function with the same name cannot be referenced in the program unit.

Example:

```
SUBROUTINE ARGR
EXTERNAL SQRT
.
.
.
Y= SQRT(X)
.
.
.
FUNCTION SQRT(XVAL)
```

The name SQRT is declared as external. The function reference SQRT(X) is therefore taken to reference the user-written function SQRT rather than the intrinsic function SQRT.

Example:

```
SUBROUTINE CHECK
EXTERNAL LOW, HIGH
.
.
.
CALL AR (LOW, VAL)
.
.
.
CALL AR (HIGH, VAL)
.
.
.
SUBROUTINE AR(FUNC, VAL)
VAL= FUNC (VAL)
.
.
.
REAL FUNCTION LOW (X)
.
.
.
REAL FUNCTION HIGH (X)
```

The names LOW and HIGH are declared as external. In one call to subroutine AR, LOW is passed as an actual argument and the function reference FUNC(VAL) is equivalent to LOW(VAL). In the second call to subroutine AR, the function reference FUNC(VAL) is equivalent to HIGH(VAL).

INTRINSIC STATEMENT

The INTRINSIC statement shown in figure 2-16 is used to identify a name as representing an intrinsic function. The INTRINSIC statement also enables use of an intrinsic function name as an actual argument.

INTRINSIC fun[,fun]. . .

fun Is an intrinsic function name.

Figure 2-16. INTRINSIC Statement

Appearance of a name in an INTRINSIC statement declares the name as an intrinsic function name. If an intrinsic function name is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in the program unit. The following intrinsic function names must not be used as actual arguments:

- Type conversion functions BOOL, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SNGL
- Lexical relationship functions LGE, LGT, LLE, and LLT
- Largest/smallest value functions AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1
- Logical and masking functions AND, OR, XOR, NEQV, EQV, COMPL

The appearance of a generic intrinsic function name in an INTRINSIC statement does not remove the generic properties of the name.

An intrinsic name can appear only once in all INTRINSIC statements in a program unit. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in the program unit.

Example:

```
SUBROUTINE DC
INTRINSIC SQRT
.
.
.
CALL SUBA (X,Y, SQRT)
.
.
.
SUBROUTINE SUBA (A, B, FNC)
B= FNC(A)
```

The name SQRT is declared intrinsic in subroutine DC and passed as an argument to subroutine SUBA. Within SUBA, the reference FNC(A) references the intrinisic function SQRT.

Example:

```
SUBROUTINE CHECK
INTRINSIC SIN, COS
.
.
.
CALL AR(SIN, VAL)
.
.
.
CALL AR(COS, VAL)
.
.
.
SUBROUTINE AR(FUNC, VAL)
VAL= FUNC(VAL)
```

The names SIN and COS are declared as intrinsic and can therefore be passed as actual arguments. In the first call to subroutine AR, the reference FUNC(VAL) is equivalent to SIN(VAL); in the second call, FUNC(VAL) is equivalent to COS(VAL). In each case, the intrinsic function is referenced.

DATA STATEMENT

The DATA statement shown in figure 2-17 is used to provide initial values for variables, arrays, array elements, and substrings. The DATA statement is nonexecutable and can appear anywhere after the specification statements in a program unit.

Entities that are initially defined by DATA statements are defined when the program begins execution. Entities that are not initially defined, and not associated with an initially defined entity, are undefined at the beginning of execution of the program.

A variable, array element, or substring must not be initially defined more than once in the program. If two entities are associated, only one can be initially defined by a DATA statement.

Names of dummy arguments, functions, and entities in blank common (including any entities associated with an entity in blank common) cannot be initially defined. Entities in a named common block can be initially defined within a block data subprogram, or within any program unit in which the named common block appears.

For each list nlist, the same number of items must be specified in the corresponding list clist. A one-to-one correspondence exists between the items specified by nlist and the constants specified by clist. The first item of nlist corresponds to the first constant of clist, the second item to the second constant, and so forth. If an unsubscripted array name appears as an item in nlist, a constant in clist must be specified for each element of the array. The values of the constants are assigned according to the storage order of the array.

For arithmetic data types, the constant is converted to the type of the associated nlist item if the types differ. For all other types, the data type of each constant in clist must be compatible with the data type of the nlist item. The correspondence is shown in table 2-1.

TABLE 2-1. CORRESPONDENCE OF DATA TYPES IN DATA STATEMENTS

Data Type of nlist Item	Data Type of Corresponding clist Constant
Integer, real, double precision, complex, or Boolean	Integer, real, double precision, complex, or Boolean. The value of the nlist item is the same as would result from an assignment statement of the form nlist-item=clist-constant.
Logical	Logical
Character	Character

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

nlist Is a list of names to be initially defined. Each name in the list can take the form:

- var
- array
- element
- substring
- dolist

var Is a variable name.

array Is an array name.

element Is an array element name (that is, subscripted array name).

substring Is a substring of a character variable or array element.

dolist Is an implied-DO list of the form:

(dlist, i = init, term [,incr])

dlist Is a list of array element names and implied-DO lists. Subscript expressions must consist of integer constants and active control variables from DO list.

i Is an integer variable called the implied-DO variable.

init Is an integer constant, symbolic constant, or expression specifying the initial value, as for DO loops.

term Is an integer constant, symbolic constant, or expression specifying the terminal value, as for DO loops.

incr Is an integer constant, symbolic constant, or expression specifying the increment, as for DO loops.

clist Is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form:

- c
- r*c
- r(c[,c] . . .)

c Is a constant or symbolic constant.

r Is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant. The repeat count can repeat the value of a single constant, or can repeat the values of a list of constants enclosed in parentheses.

Figure 2-17. DATA Statement

Each subscript expression used in an array element name in nlist must be an integer constant expression, except that implied-DO variables can be used if the array element name is in dlist. Each substring expression used for an item in nlist must be an integer constant expression.

Examples:

```
INTEGER K(6)
DATA JR/4/
DATA AT/5.0/, AQ/7.5/
DATA NRX, SRX/17.0, 5.2/
DATA K/1, 2, 3, 3, 2, 1/
```

The variables JR, AT, AQ, and SRX are initially defined with the values 4, 5.0, 7.5, and 5.2, respectively. The variable NRX is initially defined with the value 17, after type conversion of the real 17.0 to the integer 17. The array K with 6 elements is initially defined with a value for each array element.

Example:

```
REAL R(10, 10)
DATA R/50*5.0, 50*75.0/
```

The array R is initially defined with the first 50 elements set to the value 5.0 and the remaining 50 elements set to the value 75.0.

Example:

```
DIMENSION TQ(2)
EQUIVALENCE (RX, TQ(2))
DATA TQ(1)/32.0/
DATA RX/47.5/
```

The first element of array TQ is initially defined with the value 32.0. The variable RX and the second element of array TQ are initially defined as 47.5, since TQ(2) is equivalenced to variable RX.

Example:

```
BOOLEAN MASK
DATA MASK/O"7777"/
```

The variable MASK is initially defined with the octal value 7777_8.

IMPLIED DO LIST

An implied DO list can be used as an item in nlist. An iteration count and the values of the implied DO variable are established from init, term, and the optional incr just as for DO loops, except that the iteration count must be positive. When the implied DO list appears in a DATA statement, the list items in dlist are specified once for each iteration of the implied DO list, with appropriate substitution of values for each occurrence of the implied DO variable i.

The appearance of a name as an implied DO variable in a DATA statement does not affect the value or definition status of a variable with the same name in the program unit. An implied DO variable has the scope of the implied DO list.

Each subscript expression used in dlist must be an integer constant expression, except that any expression can contain an implied DO variable if the subscript expression is within the corresponding implied DO list.

Example 1:

```
REAL X(5, 5)
DATA ((X(J, I), I= 1, J), J= 1, 5)/15*1.0/
```

Elements of array X are initially defined with the DATA statement. Elements in the lower diagonal part of the matrix are set to the value 1.0. The elements initialized are:

```
(1,1)
(2,1)  (2,2)
(3,1)  (3,2)  (3,3)
(4,1)  (4,2)  (4,3)  (4,4)
(5,1)  (5,2)  (5,3)  (5,4)  (5,5)
```

Example 2:

```
PARAMETER (PI=3.14159)
REAL Y(5,5)
DATA ((Y(J+1,I),J=I+1,4),I=1,3)/6*PI/
```

CHARACTER DATA INITIALIZATION

For initialization by DATA statement, a character item in nlist must correspond to a character constant in clist. The initial value is assigned according to the following rules:

- If the length of the character item in nlist is greater than the length of the corresponding character constant, the additional character positions in the item are initially defined as blanks.
- If the length of the character item in nlist is less than the length of the corresponding character constant, the additional characters in the constant are ignored.

Note that initial definition of a character item causes definition of all character positions. Each character constant initially defines exactly one character variable, array element, or substring.

Examples:

```
CHARACTER STR1*6, STR2*3
DATA STR1/'ABCDE'/
DATA STR2/'FGHJK'/
```

The character variables STR1 and STR2 are initially defined. Variable STR1 is set to 'ABCDEΔ', with the sixth character position defined as blank. Variable STR2 is set to 'FGH', with the fourth and fifth characters of the constant ignored.

This section describes the ways in which expressions are written and evaluated. Expressions can be arithmetic, character, relational, logical, or Boolean expressions. Expressions are formed from a combination of operators, operands, and parentheses.

This section also describes assignment statements, which are executable statements. The assignment statements in a program use expressions to define or redefine the values of variables.

EXPRESSIONS

Arithmetic, Boolean, character, relational, and logical expressions are described separately. The relational expressions are not fully independent and are used as parts of logical expressions.

A constant expression is an expression in which only constants (or symbolic constants) and operators are used. If an arithmetic expression is written using only constants and operators, the expression is an arithmetic constant expression. A Boolean, character, or logical expression that contains only constants and operators is, respectively, a Boolean constant expression, character constant expression, or logical constant expression.

ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of unsigned constants, symbolic constants, variables, array elements, and function references separated by operators and parentheses. For example:

```
(A-B)*F + C/D**E
```

is a valid arithmetic expression.

An arithmetic expression can be an unsigned arithmetic constant, symbolic name of an arithmetic constant, arithmetic variable reference, arithmetic array element reference, or arithmetic function reference. More complicated arithmetic expressions can be formed by using one or more arithmetic or Boolean operands together with arithmetic operators and parentheses. Arithmetic operands identify values of type integer, real, double precision, or complex.

The arithmetic operators are shown in table 3-1. Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either operates on a pair of operands and is written between the two operands, or operates on a single operand and is written preceding that operand.

The syntax for an arithmetic expression is shown in figure 3-1.

The interpretation of a division can depend on the data types of the operands. A set of rules establishes the interpretation of an arithmetic expression that contains two or more operators. A precedence among the arithmetic operators determines the order in which the operands are to be combined:

```
   **          Highest
*  and  /      Intermediate
+  and  -      Lowest
```

For example, in the expression:

```
- A ** 2
```

the exponentiating operator (**) has precedence over the negation operator (-). The operands of the exponentiation operator are combined to form an expression used as the operand of the negation operator. The above expression is the same as the expression:

```
- (A ** 2)
```

TABLE 3-1. ARITHMETIC OPERATORS

Operator	Representing	Use of Operator	Meaning
**	Exponentiation	x1 ** x2	Exponentiate x1 to the power x2.
*	Multiplication	x1 * x2	Multiply x1 and x2.
/	Division	x1 / x2	Divide x1 by x2.
+	Addition	x1 + x2	Add x1 and x2.
+	Identity	+ x2	Same as x2.
-	Subtraction	x1 - x2	Subtract x2 from x1.
-	Negation	- x2	Negate x2.

arithexp

arithexp — Is an arithmetic expression in one of the forms:

term
+ term
- term
arithexp + term
arithexp - term
+ boolprim
- boolprim
arithexp + boolprim
arithexp - boolprim
boolprim + term
boolprim - term
boolprim + boolprim
boolprim - boolprim

term — Is an arithmetic term in one of the forms:

fact
term * fact
term / fact
term * boolprim
term / boolprim
boolprim * fact
boolprim / fact
boolprim * boolprim
boolprim / boolprim

fact — Is an arithmetic factor in one of the forms:

prim
prim ** fact
boolprim ** fact
prim ** boolprim
boolprim ** boolprim

boolprim — Is a Boolean primary, as described for Boolean expressions.

prim — Is an arithmetic primary. An arithmetic primary can be an arithmetic expression enclosed in parentheses, or any of the following:

Unsigned arithmetic constant
Arithmetic symbolic constant
Arithmetic variable
Arithmetic array element reference
Arithmetic function reference

Figure 3-1. Arithmetic Expression

Successive exponentiations are combined from right to left. For example:

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

has the same interpretation as:

```
2**(3**2)
```

Two or more multiplication or division operators are combined from left to right.

Two or more addition or subtraction operators are combined from left to right. Note that arithmetic expressions containing two consecutive arithmetic operators, such as A**-B or A+-B are not permitted. However, expressions such as A**(-B) and A+(-B) are permitted.

Subexpressions containing operators of equal precedence are evaluated from left to right. The compiler may reorder individual operations that are mathematically associative and/or commutative to perform optimizations such as removal of repeated subexpressions. The mathematical results of the reordering are correct but the specific order of evaluation is indeterminate. For example, the expression A/B*C is guaranteed to algebraically equal (AC)/B, not A/(BC), but the specific order of evaluation by the compiler is indeterminate.

An arithmetic constant expression contains only arithmetic constants, symbolic names of arithmetic constants, arithmetic constant expressions enclosed in parentheses, Boolean constants, symbolic Boolean constants, or Boolean constant expressions enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer or Boolean. If the exponent e is of type Boolean, the value used is INT(e). Note that variable, array element, and function references are not allowed.

An integer constant expression is an arithmetic constant expression or a Boolean constant expression in which each constant or symbolic name of a constant is of type integer or Boolean. If a Boolean constant expression e appears, the value used is INT(e). Note that variable, array element, and function references are not allowed. The following are examples of integer constant expressions:

```
3
-3
-3+4
O"74"
R"A"
R"AB".AND. 48
```

The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands. Integer expressions, real expressions, double precision expressions, and complex expressions are arithmetic expressions whose values are of type integer, real, double precision, and complex, respectively. When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand unless the operand is of type Boolean, in which case the type of the resulting expression is integer.

When an arithmetic operator operates on a pair of arithmetic operands, the data type of the resulting expression is given in table 3-2 for exponentiation and table 3-3 for the other operators. Four entries in table 3-2 specify a value raised to a complex power. In these cases, the value of the expression is the principal value.

If two arithmetic operands are of different type, the operand that differs in type from the result of the operation is converted to the type of the result. The operator then operates on a pair of operands of the same type. The exception to this is an operand of type real, double precision, or complex raised to an integer power; the integer operand is not converted. If the value of J is negative, the interpretation of I**J is the same as the interpretation of 1/(I**ABS(J)), which is subject to the rules for integer division. For example, 2**(-3) has the value of 1/(2**3), which is zero.

Also, a Boolean operand in an exponentiation operation is converted to integer. For the + - * and / operations, if two operands are of different type and one type is Boolean, the result has the type of the other operand. If

TABLE 3-2. RESULTING DATA TYPE FOR X1**X2

Type of x1	Type of x2	x1 Value Used	x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double precision	DBLE(x1)	x2	Double precision
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	x2	Real
Real	Real	x1	x2	Real
Real	Double precision	DBLE(x1)	x2	Double precision
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double precision	Integer	x1	x2	Double precision
Double precision	Real	x1	DBLE(x2)	Double precision
Double precision	Double precision	x1	x2	Double precision
Double precision	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	x2	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double precision	x1	CMPLX(SNGL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

TABLE 3-3. RESULTING DATA TYPE FOR X1+X2, X1-X2, X1*X2 OR X1/X2

Type of x1	Type of x2	x1 Value Used	x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double precision	DBLE(x1)	x2	Double precision
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	REAL(x2)	Real
Real	Real	x1	x2	Real
Real	Double precision	DBLE(x1)	x2	Double precision
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double precision	Integer	x1	DBLE(x2)	Double precision
Double precision	Real	x1	DBLE(x2)	Double precision
Double precision	Double precision	x1	x2	Double precision
Double precision	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double precision	x1	CMPLX(SNGL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

both operands are of type Boolean, the result has type integer. The result of the operator + or the operator - operating on a single Boolean operand is of type integer. A Boolean operand is converted to the type of the result, and the operation is performed on the converted operand.

One operand of type integer can be divided by another operand of type integer to yield an integer result. The result is the signed nonfractional part of the mathematical quotient. For example, (-10)/4 is -2, formed by discarding the fractional part of the mathematical quotient -2.5.

CHARACTER EXPRESSIONS

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character. The simplest form of a character expression is a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, or character function reference. More complicated character expressions can be formed by using one or more character operands together with character operators and parentheses. The character operator is shown in table 3-4.

TABLE 3-4. CHARACTER OPERATOR

Operator	Representing	Use of Operator	Meaning
//	Concatenation	x1//x2	Concatenate x1 and x2.

The result of a concatenation operation is a character string concatenated on the right with another string and whose length is the sum of the lengths of the strings. For example, the value of 'AB' // 'CDE' is the string 'ABCDE'.

A character expression and the operands of a character expression must identify values of type character. Except in a character assignment statement, a character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is a symbolic constant.

The syntax for a character expression is shown in figure 3-2.

Two or more concatenation operators are combined from left to right to interpret the expression. For example, the interpretation of the character expression:

'AB' // 'CD' // 'EF'

is the same as the interpretation of the character expression:

('AB' // 'CD') // 'EF'

The value of the preceding expression is the same as that of the constant 'ABCDEF'. Note that parentheses have no effect on the value of a character expression. Thus, the expression:

'AB'//('CD'//'EF')

has the same value as the preceding expressions.

charexp

charexp	**Is a character expression in either form:**
	charprim **charexp // charprim**
charprim	**Is a character primary. A character primary can be a character expression enclosed in parentheses, or any of the following:**
	Character constant **Character symbolic constant** **Character variable** **Character array element reference** **Character substring reference** **Character function reference**

Figure 3-2. Character Expression

A character constant expression is a character expression in which each operand is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Note that variable, array element, substring, and function references are not allowed.

RELATIONAL EXPRESSIONS

Relational expressions can appear only within logical expressions. Evaluation of a relational expression produces a logical result with a true or false value.

A relational expression is used to compare the values of two arithmetic or Boolean expressions or two character expressions. A relational expression cannot be used to compare the value of an arithmetic expression with the value of a character expression.

The relational operators are shown in table 3-5. The syntax of a relational expression is shown in figure 3-3.

An operand of type complex is permitted only when the relational operator is .EQ. or .NE.

An arithmetic relational expression has the logical value true only if the values of the operands satisfy the relation specified by the operator. If the two arithmetic expressions are of different types, or both are Boolean, the value of the relational expression:

X1 relop X2

is the value of the expression:

((X1) - (X2)) relop 0

where 0 (zero) is of the same type as the expression. Note that the comparison of a double precision value and a complex value is not permitted.

A character relational expression has the logical value true only if the values of the operands satisfy the relation specified by the operator. The character expression X1 is considered to be less than X2 if the value of X1 precedes the value of X2 in the collating sequence; X1 is greater than X2 if the value of X1 follows the value of X2 in the collating sequence. Note that the collating sequence in use determines the result of the comparison. The default collating sequence is ASCII6 as shown in appendix A. Also refer to Collation Control in appendix E.

TABLE 3-5. RELATIONAL OPERATORS

Operator	Representing	Use of Operator	Meaning
.LT.	Less than	x1.LT.x2	Is x1 less than x2?
.LE.	Less than or equal to	x1.LE.x2	Is x1 less than or equal to x2?
.EQ.	Equal to	x1.EQ.x2	Is x1 equal to x2?
.NE.	Not equal to	x1.NE.x2	Is x1 not equal to x2?
.GT.	Greater than	x1.GT.x2	Is x1 greater than x2?
.GE.	Greater than or equal to	x1.GE.x2	Is x1 greater than or equal to x2?

relexp

relexp — **Is a relational expression used as a primary in a logical expression. A relational expression is in one of the forms:**

arithexp rop arithexp
arithexp rop boolprim
boolprim rop arithexp
boolprim rop boolprim
charexp rop charexp

rop — **Is one of the relational operators:**

.LT.
.LE.
.EQ.
.NE.
.GT.
.GE.

arithexp — **Is an arithmetic expression.**

boolprim — **Is a Boolean primary, as described for Boolean expressions.**

charexp — **Is a character expression.**

Figure 3-3. Relational Expression

Character relational expressions in PARAMETER and conditional compilation (C$IF) statements are always evaluated using the ASCII6 sequence.

If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand.

LOGICAL EXPRESSIONS

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

The simplest form of a logical expression is a logical constant, symbolic name of a logical constant, logical variable reference, logical array element reference, logical function reference, or relational expression. More complicated logical expressions can be formed by using one or more logical operands together with logical operators and parentheses.

The logical operators are shown in table 3-6. The syntax of a logical expression is shown in figure 3-4.

A set of rules establishes the interpretation of a logical expression that contains two or more logical operators. A precedence among the logical operators determines the order in which the operands are to be combined, unless the order is changed by the use of parentheses. The precedence of the logical operators is:

.NOT.	Highest
.AND.	
.OR.	
.EQV. or .NEQV. or .XOR.	Lowest

logexp

logexp — **Is a logical expression in one of the forms:**

logdis
logexp .EQV. logdis
logexp .NEQV. logdis
logexp .XOR. logdis

logdis — **Is a logical disjunction in either form:**

logterm
logdis .OR. logterm

logterm — **Is a logical term in either form:**

logfact
logterm .AND. logfact

logfact — **Is a logical factor in either form:**

logprim
.NOT. logprim

logprim — **Is a logical primary. A logical primary can be a logical expression enclosed in parentheses, a relational expression, or any of the following:**

Logical constant
Logical symbolic constant
Logical variable
Logical array element reference
Logical function reference

Figure 3-4. Logical Expression

TABLE 3-6. LOGICAL OPERATORS

Operator	Representing	Use of Operator	Meaning
.NOT.	Logical negation	.NOT.x	Complement x
.AND.	Logical conjunction	x1.AND.x2	Boolean product of x1 and x2
.OR.	Logical inclusive disjunction	x1.OR.x2	Boolean sum of x1 and x2
.EQV.	Logical equivalence	x1.EQV.x2	Is x1 logically equivalent to x2?
.NEQV.	Logical nonequivalence	x1.NEQV.x2	Is x1 not logically equivalent to x2?
.XOR.	Logical exclusive disjunction	x1.XOR.x2	Boolean difference of x1 and x2

For example, in the expression:

A .OR. B .AND. C

the .AND. operator has higher precedence than the .OR. operator; therefore, the interpretation is the same as:

A .OR. (B .AND. C)

In interpreting a logical expression containing two or more .AND. operators; two or more .OR. operators; or two or more .EQV., NEQV., or .XOR. operators, the logical quantities are combined from left to right.

The value of a logical factor involving any logical operator is shown in table 3-7.

A logical constant expression contains only logical constants, symbolic names of logical constants, relational expressions which contain only constant expressions, or logical constant expressions enclosed in parentheses. Note that variable, array element, and function references are not allowed.

BOOLEAN EXPRESSIONS

A Boolean expression is formed with logical operators and Boolean or arithmetic operands. Evaluation of a Boolean expression produces a result of type Boolean.

The syntax of a Boolean expression is shown in figure 3-5.

In interpreting a Boolean expression containing two or more .AND. operators; two or more .OR. operators; or two or more .EQV., NEQV., or .XOR. operators, the Boolean quantities are combined from left to right.

If an operand is of type integer, real, double precision, or complex, it is converted to Boolean and the operation is performed on the converted operand.

A Boolean operator determines each bit value of the result it yields independently of other bits of the result. Each bit value is determined from the corresponding bit values of the operands. At each bit position, the bit in the result is determined as shown in table 3-8.

A Boolean constant expression is a Boolean expression which contains only Boolean constants, symbolic names of Boolean constants, Boolean constant expressions enclosed in parentheses, arithmetic constants, symbolic arithmetic constants, or arithmetic constant expressions enclosed in parentheses.

GENERAL RULES FOR EXPRESSIONS

The order in which operands are combined using operators is determined by:

1. Use of parentheses
2. Precedence of the operators
3. Right-to-left interpretation of exponentiations
4. Left-to-right interpretation of multiplications and divisions
5. Left-to-right interpretation of additions and subtractions in an arithmetic expression
6. Left-to-right interpretation of concatenations in a character expression
7. Left-to-right interpretation of .AND. operators
8. Left-to-right interpretation of .OR. and .NOT. operators
9. Left-to-right interpretation of .EQV., NEQV., and .XOR. operators in a logical expression or Boolean expression

TABLE 3-7. RESULT OF LOGICAL OPERATORS

x1	x2	.NOT.x2	x1.AND.x2	x1.OR.x2	x1.EQV.x2	x1.NEQV.x2	x1.XOR.x2
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.

boolexp

boolexp Is a Boolean expression in one of the forms:

```
booldis
boolexp .EQV. booldis
boolexp .EQV. arithexp
arithexp .EQV. booldis
arithexp .EQV. arithexp
boolexp .NEQV. booldis
boolexp .NEQV. arithexp
arithexp .NEQV. booldis
arithexp .NEQV. arithexp
boolexp .XOR. booldis
boolexp .XOR. arithexp
arithexp .XOR. booldis
arithexp .XOR. arithexp
```

booldis Is a Boolean disjunct in one of the forms:

```
boolterm
booldis .OR. boolterm
booldis .OR. arithexp
arithexp .OR. boolterm
arithexp .OR. arithexp
```

boolterm Is a Boolean term in one of the forms:

```
boolfact
boolterm .AND. boolfact
boolterm .AND. arithexp
arithexp .AND. boolfact
arithexp .AND. arithexp
```

boolfact Is a Boolean factor in one of the forms:

```
boolprim
.NOT. boolprim
.NOT. arithexp
```

arithexp Is an arithmetic expression.

boolprim Is a Boolean primary. A Boolean primary can be a Boolean expression enclosed in parentheses, or any of the following:

```
Unsigned Boolean constant
Boolean symbolic constant
Boolean variable
Boolean array element reference
Boolean function reference
```

Figure 3-5. Boolean Expression

Precedences have been established among the arithmetic and logical operators. There is only one character operator. No precedence is established among the relational operators. The precedences among the operators are:

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

An expression can contain more than one kind of operator. For example, the logical expression:

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real, and L is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted as:

```
L .OR. ((A + B) .GE. C)
```

Any variable, array element, function, or character substring involved in an expression must be defined at the time the reference is made. An integer operand must be defined with an integer value rather than a statement label value. Note that if a character string or substring is referenced, all of the referenced character positions must be defined at the time the reference is executed.

Any arithmetic operation whose result is not mathematically defined is prohibited; for example, dividing by zero and raising a zero-valued primary to a zero-valued or negative-valued power.

A function reference in a statement must not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement must not alter the value of any entity in common that affects the value of any other function reference in that statement. However, execution of a function reference in the expression of a logical IF statement can affect entities in the statement that is executed when the value of the expression is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements:

```
A(I) = F(I)

Y = G(X) + X
```

are prohibited if the reference to F defines I, or the reference to G defines X.

TABLE 3-8. RESULT OF LOGICAL OPERATORS IN BOOLEAN EXPRESSIONS

Each bit in x1	Corresponding bit in x2	Corresponding bit in result of					
		.NOT.x2	x1.AND.x2	x1.OR.x2	x1.EQV.x2	x1.NEQV.x2	x1.XOR.x2
0	0	1	0	0	1	0	0
0	1	0	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	0	0

All of the operands of an expression are not necessarily evaluated if the value of the expression can be determined otherwise. For example, in the logical expression:

```
X .GT. Y .OR. L(Z)
```

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the expression causes Z to become undefined if L defines its argument.

If a statement contains more than one function reference, the functions can be evaluated in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement:

```
Y = F(G(X))
```

where F and G are functions, requires G to be evaluated before F is evaluated.

Any expression contained in parentheses is always treated as an entity. For example, in evaluating the expression A*(B*C), the product of B and C is evaluated and then multiplied by A; the mathematically equivalent expression (A*B)*C is not used.

ASSIGNMENT STATEMENTS

There are five types of assignment statements:

Arithmetic

Logical

Statement label (with the ASSIGN statement as described in section 4)

Character

Boolean

ARITHMETIC ASSIGNMENT STATEMENT

The form of an arithmetic assignment statement is shown in figure 3-6.

```
v = e
```

v Is the name of a variable or array element of type integer, real, double precision, or complex.

e Is an arithmetic or Boolean expression.

Figure 3-6. Arithmetic Assignment

After evaulation of arithmetic expression e, the result is converted to the type of v in the following way:

Integer	INT (e)
Real	REAL(e)
Double precision	DBLE (e)
Complex	CMPLX (e)

The result is then assigned to v, and v is defined or redefined with that value.

CHARACTER ASSIGNMENT STATEMENT

The form of a character assignment statement is shown in figure 3-7.

```
v = e
```

v Is the name of a character variable, character array element, or character substring.

e Is a character expression.

Figure 3-7. Character Assignment

The character expression e is evaluated, and the result is then assigned to v. None of the character positions being defined in v can be referenced in e.

The variable v and expression e can have different lengths. If the length of v is greater than the length of e, e is extended to the right with blank characters until it is the same length as v. If the length of v is less than the length of e, e is truncated from the right until it is the same length as v.

Only as much of the value of e must be defined as is needed to define v. In the example:

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

the assignment A=B requires that the substring B(1:2) be defined. It does not require that the substring B(3:4) be defined. If v is a substring, e is assigned only to the substring. The definition status of substrings not specified by v is unchanged.

LOGICAL ASSIGNMENT STATEMENT

The form of a logical assignment statement is shown in figure 3-8. The logical expression is evaluated and the result is then assigned to v. Note that e must have a value of either .TRUE. or .FALSE.

```
v = e
```

v Is the name of a logical variable or logical array element.

e Is a logical expression.

Figure 3-8. Logical Assignment

BOOLEAN ASSIGNMENT STATEMENT

The form of a Boolean assignment statement is shown in figure 3-9.

The Boolean or arithmetic expression e is evaluated. If e is an arithmetic expression, the result used is BOOL(e). The result is then assigned to v.

```
v = e
```

v Is the name of a Boolean variable or Boolean array element.

e Is a Boolean or arithmetic expression.

Figure 3-9. Boolean Assignment

MULTIPLE ASSIGNMENT

The form of a multiple assignment statement is shown in figure 3-10.

Execution of a multiple assignment statement causes the evaluation of the expression e. After any necessary conversion, the assignment and definition of the rightmost v with the value of e occurs. Assignment and definition of each additional v occurs in right-to-left order. The value assigned to each v is the value of the v immediately to its right, after any necessary conversion.

```
v = [v=] . . . e
```

v Is the name of a variable, array element, or character substring.

e Is an expression. The types of the elements v and the expression e must ensure that v = e is a valid assignment for each v specified.

Figure 3-10. Multiple Assignment

FORTRAN flow control statements provide a means of altering, interrupting, terminating, or otherwise modifying the normal sequential flow of execution. The flow control statements are as follows:

Unconditional GO TO	ELSE IF	PAUSE
Computed GO TO	ELSE	END
Assigned GO TO	END IF	CALL
Arithmetic IF	DO	RETURN
Logical IF	CONTINUE	
Block IF	STOP	

Control can be transferred only to an executable statement.

A statement can be identified by a label consisting of an integer in the range 1 through 99999, with leading zeros and embedded blanks ignored. Each statement label must be unique in the program unit (main program or subprogram) in which it appears.

GO TO STATEMENT

The three types of GO TO statements are unconditional GO TO, computed GO TO, and assigned GO TO. The ASSIGN statement is used in conjunction with the assigned GO TO and is therefore described in this subsection.

UNCONDITIONAL GO TO STATEMENT

The unconditional GO TO statement is shown in figure 4-1.

```
GO TO s1

     s1      Is the label of an executable statement.
```

Figure 4-1. Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the statement identified by the specified label. The labeled statement must appear in the same program unit as the GO TO statement. An example of an unconditional GO TO statement is shown in figure 4-2.

```
 10 A=B+Z
100 B=X+Y
    IF(A-B)20,20,30
 20 Z=A
    GO TO 10 <------ Transfers control to statement 10.
 30 Z=B
    STOP
    END
```

Figure 4-2. Example of Unconditional GO TO Statement

COMPUTED GO TO STATEMENT

The computed GO TO statement is shown in figure 4-3. This statement transfers control to the statement identified by one of the specified labels.

```
GO TO(s1 [,s1 ]. . . )[,]exp

     s1      Is the label of an executable statement that
             appears in the same program unit as the
             GO TO statement.

     exp     Is an integer, arithmetic, or Boolean expression.
```

Figure 4-3. Computed GO TO Statement

The label selected is determined by the value of the expression. If exp has a value of one, control transfers to the statement identified by the first label in the list; if exp has a value of i, control transfers to the statement identified by the i^{th} label in the list. The value of exp is truncated and converted to integer, if necessary.

If the value of exp is less than one or greater than the number of labels in the list, execution continues with the statement following the computed GO TO.

Figure 4-4 illustrates some examples of computed GO TO statements.

ASSIGN STATEMENT

The ASSIGN statement is shown in figure 4-5. This statement assigns a statement label to an integer variable. The value assigned to iv represents the label of an executable or a FORMAT statement. The labeled statement must appear in the same program unit as the ASSIGN statement. Once iv is used in an ASSIGN statement, it cannot be used in any statement other than an assigned GO TO statement, an ASSIGN statement, or an I/O statement, until it has been redefined.

The assignment must be made prior to execution of the assigned GO TO statement or the input/output statement that references assigned label s1.

Figure 4-6 illustrates some examples of ASSIGN statements.

ASSIGNED GO TO STATEMENT

The assigned GO TO statement transfers control to the executable statement last assigned to iv by the execution of a prior ASSIGN statement. The assigned GO TO statement is shown in figure 4-7.

Example 1:

```
GO TO(10,20,30,20)L
```

The next statement executed is:

10 if L = 1

20 if L = 2

30 if L = 3

20 if L = 4

Example 2:

```
K=2
GO TO(100,150,300),K
```

Statement 150 is executed next.

Example 3:

```
K=2
X=4.6
.
.
.
GO TO(10,110,11,12,13),X/K
```

Control transfers to statement 110, since the integer value of the expression X/K equals 2.

Example 4:

```
M=4
GO TO(100,200,300),M
A=B+C
```

Execution continues with the statement A=B+C, since the value of M is greater than the number of labels enclosed in the parentheses.

Figure 4-4. Examples of Computed GO TO Statements

ASSIGN sl TO iv

sl	Is the label of an executable or FORMAT statement.
iv	Is an integer variable.

Figure 4-5. ASSIGN Statement

The variable iv must not be defined by any statement other than an ASSIGN statement. The list of statement labels is optional. All labels in a statement label list must be in the same program unit as both the ASSIGN and assigned GO TO statements. Also, iv must be one of the labels in the list. Figure 4-8 illustrates an example of an assigned GO TO statement.

Example 1:

```
ASSIGN 10 TO LSWIT
GO TO LSWIT (5,10,15,20)
```

Control transfers to the statement labeled 10.

Example 2:

```
ASSIGN 24 TO IFMT
WRITE (2,IFMT)A,B
```

The variables A and B are formatted according to the FORMAT statement labeled 24.

Figure 4-6. Examples of ASSIGN Statement

GO TO iv [[,] (sl [,sl] . . .)]

iv	Is an integer variable.
sl	Is the label of an executable statement that appears in the same program unit as the assigned GO TO statement.

Figure 4-7. Assigned GO TO Statement

```
   ASSIGN 50 TO JUMP
10 GO TO JUMP,(20,30,40,50)      Statement 50 is executed
   .                             immediately after state-
20 CONTINUE                      ment 10.
   .
30 CAT=ZERO+HAT
   .
   .
   .
40 CAT=10.1-3.
   .
   .
   .
50 CAT=25.2+7.3
```

Figure 4-8. Example of Assigned GO TO Statement

IF STATEMENT

The IF statement evaluates an expression and conditionally transfers control or executes another statement, depending on the outcome of the test. The kinds of IF statements are as follows:

Arithmetic IF
Logical IF
Block IF

The ELSE, ELSE IF, and END IF statements are also discussed in this subsection since they are used in conjunction with a block IF statement.

ARITHMETIC IF STATEMENT

The arithmetic IF statement is shown in figure 4-9.

IF (exp) sl_1,sl_2,sl_3

exp	Is an integer, real, double precision, or Boolean expression.
sl_1,sl_2,sl_3	Are statement labels of executable statements that appear in the same program unit as the arithmetic IF statement.

Figure 4-9. Arithmetic IF Statement

The arithmetic IF statement transfers control to the statement labeled sl_1 if the value of exp is less than zero, to the statement labeled sl_2 if it is equal to zero, or to the statement labeled sl_3 if it is greater than zero. If exp is type Boolean, INT(exp) is used.

Figure 4-10 illustrates an example of an arithmetic IF statement.

```
    PROGRAM IF
    READ (5,100) I,J,K,N
100 FORMAT (10X,4I4)
    IF(I-N) 3,4,6
  3 ISUM=J+K
  6 CALL ERROR1
    WRITE (6,2) ISUM
  2 FORMAT (I10)
  4 STOP
    END
```

Figure 4-10. Example of Arithmetic IF Statement

LOGICAL IF STATEMENT

The logical IF statement is shown in figure 4-11.

IF (exp) stat

exp	Is a logical expression.
stat	Is any executable statement except a DO, block IF, ELSE, ELSE IF, END, END IF, or another logical IF statement.

Figure 4-11. Logical IF Statement

The logical IF statement allows for conditional execution of a statement. If the value of exp is true, statement stat is executed. If the value of exp is false, stat is not executed; execution continues with the next statement. Figure 4-12 illustrates some examples of logical IF statements.

BLOCK IF STATEMENT

The block IF statement provides for conditional execution of a block of executable statements. The block IF statement is used with the END IF and, optionally, the ELSE and ELSE IF statements to form block IF structures. The block IF statement is shown in figure 4-13.

```
    IF (P.AND.Q) RES=7.2
50 TEMP=ANS*Z
```

If P and Q are both true, the value of the variable RES is replaced by 7.2; otherwise, the value of RES is unchanged. In either case, statement 50 is executed.

```
    IF (A.LT.B) CALL SUB1
20 ZETA=TEMP+RES4
```

If A is less than B, the subroutine SUB1 is called. Upon return from this subroutine, statement 20 is executed. If A is greater than or equal to B, statement 20 is executed and SUB1 is not called.

Figure 4-12. Examples of Logical IF Statements

IF (exp) THEN

exp	Is a logical expression.

Figure 4-13. Block IF Statement

If the logical expression exp is true, execution continues with the next executable statement. If exp is false, control transfers to an ELSE or ELSE IF statement, or if none are present, to an END IF statement.

ELSE STATEMENT

The ELSE statement provides an alternate path of execution for a block IF statement or an ELSE IF statement. The ELSE statement is shown in figure 4-14.

An ELSE statement can have a statement label; however, the label cannot be referenced in any other statement.

ELSE

Figure 4-14. ELSE Statement

ELSE IF STATEMENT

The ELSE IF statement combines the functions of the ELSE and block IF statements. This statement provides an alternate path of execution for a block IF or another ELSE IF statement and performs a conditional test. The ELSE IF statement makes it possible to form a block IF structure with more than one alternative. The ELSE IF statement is shown in figure 4-15.

ELSE IF (exp) THEN

exp	Is a logical expression.

Figure 4-15. ELSE IF Statement

An ELSE IF statement can have a statement label; however, the label cannot be referenced by any other statement.

The effect of executing an ELSE IF statement is the same as for a block IF statement.

END IF STATEMENT

The END IF statement terminates a block IF structure. For each block IF statement there must be a corresponding END IF statement. A statement label for an END IF statement cannot be referenced. The END IF statement is shown in figure 4-16.

```
END IF
```

Figure 4-16. END IF Statement

BLOCK IF STRUCTURES

Block IF structures provide for alternative execution of blocks of statements. A block IF structure begins with a block IF statement, ends with an END IF statement and, optionally, includes one ELSE or one or more ELSE IF statements. Each block IF, ELSE, and ELSE IF statement is followed by an associated block of executable statements called an if-block.

The simplest form of a block IF structure is shown in figure 4-17.

```
IF (exp) THEN
   if-block
END IF
```

Figure 4-17. Simple Block IF Structure

If exp is true, execution continues with the first statement in the if-block. If exp is false, control transfers to the statement following the END IF statement. The if-block can contain any number of executable statements, including block IF statements.

Control can be transferred out of an if-block from inside the if-block. However, control cannot be transferred into an if-block from outside the if-block. It is not permissible to branch directly to an ELSE, ELSE IF, or END IF statement. However, it is permissible to branch directly to a block IF statement.

When execution of the statements in an if-block has completed, and if control has not been transferred outside an if-block, execution continues with the statement following END IF.

An example of a simple block IF structure is shown in figure 4-18. In this and subsequent examples, indentation is used to indicate levels of block IF structures.

```
IF (I.EQ.0) THEN
   X=X+DX
   Y=Y+DY
END IF
```

If I is zero, the subsequent statements are executed. If not, control transfers to the statement following END IF.

Figure 4-18. Example of Block IF Statement

A block IF structure can contain one ELSE statement to provide an alternative path of execution within the structure. Figure 4-19 shows a block IF structure containing an ELSE statement.

```
IF (exp) THEN
   if-block-1
ELSE
   if-block-2
END IF
```

Figure 4-19. Block IF Structure With ELSE Statement

In this structure, if exp is true, execution continues with the first statement in if-block-1. If the last statement of if-block-1 does not transfer control, control transfers to the statement following END IF.

If exp is false, control transfers to the first statement in if-block-2. If the last statement in if-block-2 does not transfer control, execution continues with the statement following END IF.

A block IF statement can have at most one associated ELSE statement.

An example of an ELSE statement is illustrated in figure 4-20.

```
READ (2,12) A,B
IF (XSUM.LT.XLIM) THEN
   X(I)=A/2.0+B/2.0
   XSUM=XSUM+X(I)
   WRITE (3,14) X(I),XSUM
ELSE
   Y(I)=A*B
   YSUM=Y(I)
   WRITE (3,16) YSUM,Y(I)
END IF
```

Figure 4-20. Example of Block IF Structure With ELSE Statement

An IF structure can contain one or more ELSE IF statements to provide for alternative execution of additional block IF statements. This capability allows the user to form IF structures containing a number of possible execution paths depending on the outcome of the associated IF tests. The IF structure with ELSE IF statements is shown in figure 4-21.

```
IF (exp1) THEN
   if-block-1
ELSE IF (exp2) THEN
   if-block-2
ELSE IF (exp3) THEN
   if-block-3
END IF
```

Figure 4-21. Block IF Structure With ELSE IF Statements

In this structure, the initial block IF statement and each ELSE IF or ELSE statement has an associated if-block. Only one if-block in this structure is executed (if no nested levels appear). Each logical expression is evaluated until one is found that is true. Control then transfers to the first statement of the associated if-block. When execution of the if-block has completed, and if control has not been transferred, control transfers to the statement following END IF. If none of the logical expressions are true and no ELSE statement appears, no if-blocks are executed; control transfers to the statement following END IF. In this structure, at most one if-block is executed.

If an ELSE statement appears, it must follow the last ELSE IF statement. If no logical expression is true, control transfers to the statement following ELSE.

Control can transfer out of a block IF structure from inside any if-block; however, control cannot transfer from one if-block to another if they are at the same nesting level.

An example of a block IF structure with ELSE IF statements is illustrated in figure 4-22.

NESTED BLOCK IF STRUCTURES

Block IF structures can be nested, that is, any if-block within a structure can itself contain block IF structures. Within a nesting hierarchy, control can transfer from a lower level structure into a higher level structure; however, control cannot transfer from a higher level structure into a lower level structure. Nested block IF structures are illustrated in figure 4-23. Figure 4-24 shows an additional example of a nested block IF structure.

```
      IF (N.EQ.1) THEN
         CALL ASUB(X,R)
         CALL BSUB(X,S)
      ELSE IF (N.EQ.2) THEN
         DO 6 I=1,100
6        X(I)=0.0
      ELSE IF (N.EQ.3) THEN
         GO TO 8
      ELSE
      END IF
         .
         .
         .
8     CONTINUE
```

Since no executable statements appear between ELSE and END IF, ELSE has no effect.

Figure 4-22. Example of Block IF Structure With ELSE IF Statements

```
IF (exp) THEN
   if-block-1
  IF (exp) THEN
   if-block-2
  END IF
ELSE
   if-block-1
END IF
```

Figure 4-23. Nested Block IF Structure

```
   IF (X.GT.Y) THEN
5     Y=Y+YINCR
      IF (K.EQ.J) THEN
         XT=X
         YT=Y
      ELSE
         K=K+1
         GO TO 5
      END IF
   ELSE
      X=X+XINCR
   END IF
```

Each level contains a block IF and an ELSE statement. The inner structure is executed only if X is greater than Y. The inner structure contains a legal branch to the outer structure.

Figure 4-24. Example of Nested Block IF Structure

DO STATEMENT

The DO statement is used to specify a loop, called a DO loop, that repeats a group of statements. The DO statement is shown in figure 4-25.

DO sl [,] v=e_1,e_2[,e_3]

sl — Is the label of an executable statement called the terminal statement of the DO loop.

v — Is an integer, real, or double precision control variable.

e_1 — Is an initial parameter.

e_2 — Is a terminal parameter.

e_3 — Is an optional increment parameter; default is 1.

e_1, e_2, and e_3 are called indexing parameters; they can be integer, real, double precision, or Boolean constants, symbolic constants, variables, or expressions.

Figure 4-25. DO Statement

The terminal statement of a DO loop is an executable statement that must physically follow and reside in the same program unit as its associated DO statement. The terminal statement must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF statement, it can contain any statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

DO LOOPS

The range of a DO loop consists of all the executable statements following the DO statement up to and including the terminal statement. Execution of a DO statement causes the following sequence of operations:

1. The expressions e_1, e_2, and e_3 are evaluated and, if necessary, converted to the type of the control variable v.
2. Control variable v is assigned the value of e_1.
3. The iteration count is established; this value is determined by the following expression:

   MAX(INT((m_2-m_1+m_3)/m_3),mtc)

   m_1, m_2, m_3

   are the values of the expressions e_1, e_2, and e_3, respectively, after conversion to the type of v.

   mtc

   is the minimum trip count; mtc has a value of either one or zero, and is established by the DO control statement parameter or DO loop control directive. If mtc=1 (DO=OT specified), the iteration count must be ≥ 1. A zero trip count is prohibited in DO=OT mode, and the results of such a loop are undefined.
4. If the iteration count is not zero, the range of the DO loop is executed. If the iteration count is zero, execution continues with the statement following the terminal statement of the DO loop; the control variable retains its most recent value.
5. Control variable v is incremented by the value of e_3.
6. The iteration count is decremented by one.

Steps 4 through 6 are repeated until the iteration count attains a value of zero.

If the DO=LONG control statement parameter is selected, the trip count for DO loops can exceed $2^{17}-1$. If DO=LONG is not selected, the trip count must not exceed $2^{17}-1$, and the following conditions must be satisfied:

$$|m_1 + m_3| < 2^{17} - 1$$

$$|m_2 + m_3| < 2^{17} - 1$$

If a DO loop appears within an if-block, the range of the DO loop must be entirely contained within the if-block. If a block IF statement appears within the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

ACTIVE AND INACTIVE DO LOOPS

Initially, a DO loop is inactive. A DO loop becomes active only when its DO statement is executed.

Once active, a loop becomes inactive when any of the following occur:

Its iteration count is determined to be zero.

A RETURN, STOP, or END statement is executed within the program unit containing the loop.

The control variable becomes undefined or is redefined (by a process other than loop incrementation).

It is in the range of another loop that becomes inactive.

It is in the range of another loop whose DO statement is executed.

Transfer of control out of the range of a DO loop does not deactivate the loop. When such a transfer occurs, the control variable retains its most recent value in the loop. Control can be returned to the range of the loop provided that the control variable is not redefined outside the range or the program unit containing the loop has not been exited by a RETURN, STOP, or END statement. The loop becomes inactive once the control variable is redefined and cannot be reentered except through its DO statement.

If a DO loop executes zero times, the control variable value equals m_1. Otherwise, the value is the most recent value of the control variable plus the increment parameter value.

If a DO loop becomes inactive but has not executed to completion (iteration count does not equal zero), its control variable retains its most recent value unless it has become undefined.

Transfer into the range of an inactive DO loop from outside the range is not permitted.

Figure 4-26 illustrates some examples of DO loops.

Example 1:

```
      DO 10 I=1,11,3
      IF(ALIST(I)-ALIST(I+1))15,10,10
   15 ITEMP=ALIST(I)
   10 ALIST(I)=ALIST(I+1)
  300 WRITE(6,200)ALIST
```

The statements following DO up to and including statement 10 are executed four times. The DO loop is executed with I equal to 1, 4, 7, and 10. Statement 300 is then executed. After completion of the loop, I has a value of 13.

Example 2:

```
      DO 10 I=5,1,-1
      PRINT 100, B(I)
   10 IF (X .GT. B(I) .AND. X .LT. H) Z=EQUATE
    6 A=ZERO+EXTRA
```

This example illustrates the use of a negative increment parameter. Statement 10 is executed five times, whether or not Z = EQUATE is executed. Statement 6 is executed only after the DO loop is satisfied.

Example 3:

```
      IVAR = 9
      .
      .
      .
      DO 20 I = 1,200
      IF (I .GE. IVAR) GO TO 10
   20 CONTINUE
   10 IN =11
```

An exit from the range of the DO is made to statement 10 when the value of the control variable I is equal to IVAR. The value of the integer variable IN becomes 11.

Example 4:

```
      K=3
      J=5
      DO 100 I=J,K
      RACK=2.-3.5+ANT(I)
  100 CONTINUE
```

If DO=OT is specified on the FTN5 control statement, the DO loop is executed once (with I=5) because J is larger than K. If DO=OT is not specified, the loop is not executed.

Figure 4-26. DO Loop Examples

NESTED DO LOOPS

When a DO loop entirely contains another DO loop, the grouping is called a DO nest. The range of a DO statement can include other DO statements providing the range of each inner DO is entirely within the range of the containing DO statements.

The last statement of an inner DO loop must be either the same as the last statement of the outer DO loop or must occur before it. If more than one DO loop has the same terminal statement, a transfer to that statement can be made from within the range of any loop sharing the statement (or from outside the range while the loop is active), and the label can be referenced in any GO TO or IF statement in the nest. Figure 4-27 illustrates some possible DO loop nests. Note that loops can be completely nested or can share a terminal statement.

A DO loop can be activated only by executing the DO statement. Once the DO statement has been executed, and before the loop is satisfied, control can be transferred out of the range and then transferred back into the range of the DO.

A transfer from the range of an outer DO into the range of an inner DO loop is not allowed; however, a transfer out of the range of an inner DO into the range of an outer DO is allowed because such a transfer is within the range of the outer DO loop. Subprograms can be called from within a DO loop. A transfer back into the range of an innermost DO loop is allowed if a transfer has been made from the same loop and is still active. Legal and illegal transfers are illustrated in figure 4-28.

Figure 4-29 illustrates some examples of nested DO loops.

A terminal statement that is shared by more than one DO loop can be referenced in a GO TO or IF statement in the range of any of the loops, provided the referencing loop is active, as illustrated in figure 4-30. If the terminal statement is referenced in an inactive loop, results are undefined.

When an IF or GO TO statement is used to bypass several inner loops, different terminal statements are required for each loop. Figure 4-31 illustrates nested DO loops with different terminal statements.

CONTINUE STATEMENT

The CONTINUE statement is shown in figure 4-32.

The CONTINUE statement performs no operation. It is an executable statement that can be placed anywhere in the executable statement portion of a source program without affecting the sequence of execution. The CONTINUE statement is most frequently used as the last statement of a DO loop. It can provide loop termination when a GO TO or IF would normally be the last statement of the loop. If the CONTINUE statement does not have a label, an informative diagnostic is issued. Figure 4-33 shows an example using a CONTINUE statement.

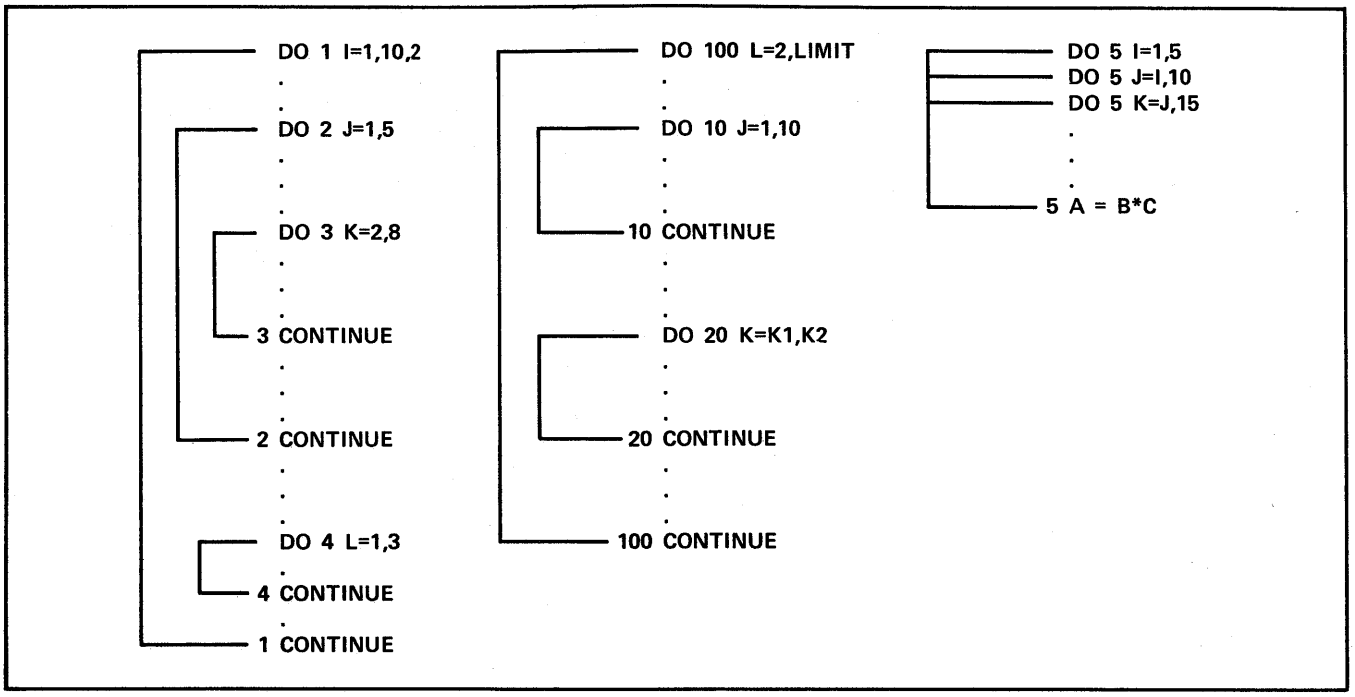


Figure 4-27. Nested DO Loops

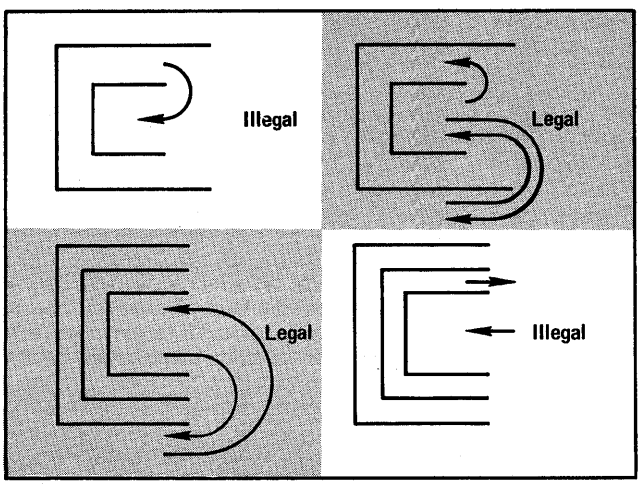


Figure 4-28. Nested DO Loop Transfers

Example 1:

```
      N=0
      DO 100 I=1,10
      J=I
      DO 100 K=1,5
      L=K
  100 N=N+1
  101 CONTINUE
```

After execution of these DO loops and at the execution of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.

Figure 4-29. Nested DO Loop Examples (Sheet 1 of 2)

Example 2:

```
      N=0
      DO 200 I=1,10
      J=I
      DO 200 K=5,1
      L=K
  200 N=N+1
  201 CONTINUE
```

If DO=OT is not specified on the FTN5 control statement, the inner loop will not be executed. If DO=OT is specified, the inner loop is illegal because DO=OT implies that the minimum trip count must be greater than or equal to 1.

Example 3:

```
      DIMENSION A(5,4,4), B(4,4)
      DO 2 I = 1,4
      DO 2 J = 1,4
      DO 1 K = 1,5
    1 A(K,J,I) = 0.0
    2 B(J,I) = 0.0
```

This example sets arrays A and B to zero.

Figure 4-29. Nested DO Loop Examples (Sheet 2 of 2)

```
      DO 10 J=1,50
      DO 10 I=1,50
      DO 10 M=1,100
      .
      .
      .
      GO TO 10
      .
      .
      .
   10 CONTINUE
```

Figure 4-30. Branch to Shared Terminal Statement

```
      DO 10 K=1,100
      IF(DATA(K).NE.10) GO TO 10 †
   20 DO 30 L=1,20
      IF(DATA(L).NE.FACT*K-10.) GO TO 30†
   40 DO 50 J=1,5
      .
      .
      .
      GO TO (101,102,50),INDEX
  101 TEST=TEST+1
      GO TO 104
  103 TEST=TEST-1
      DATA(K)=DATA(K)*2.0
      .
      .
      .
   50 CONTINUE
   30 CONTINUE
   10 CONTINUE
      .
      .
      .
      GO TO 104
  102 DO 109 M=1,3
      .
      .
      .
  109 CONTINUE
      .
  104 CONTINUE
```

†Transfer bypasses inner loops.

Figure 4-31. Nested DO Loops With Different Terminal Statements

```
CONTINUE
```

Figure 4-32. CONTINUE Statement

Example:

```
      DO 10 I = 1,11
      IF(A(I).GE.A(I+1)) GO TO 10

      A (I) = A (I+1)
   10 CONTINUE
```

Figure 4-33. CONTINUE Statement Examples

PAUSE STATEMENT

The PAUSE statement is shown in figure 4-34. When a PAUSE statement is encountered during execution, the program halts and PAUSE n appears as a dayfile message on the operator console. If the job is executing interactively, PAUSE n appears as a dayfile message at the user terminal (does not apply to SCOPE 2). For batch originated programs, the console operator can continue or terminate the program with an entry from the console.

For programs executing interactively through INTERCOM under NOS/BE and SCOPE 2, the user types GO to continue execution or DROP to terminate. For any other type-in, a diagnostic message is issued and INTERCOM waits for a correct type-in.

For programs executing interactively through IAF under NOS, the user types the user break 2 sequence to terminate program execution. Any other type-in causes execution to continue.

For programs executing interactively through the NOS Time-Sharing System, the user types STOP to terminate execution. Any other type-in causes execution to continue.

Examples:

```
PAUSE 45321

PAUSE 'EXAMPLE TWO'
```

PAUSE[n]

n Is a string of 1 through 5 decimal digits, or a character constant of at most 70 characters.

Figure 4-34. PAUSE Statement

STOP STATEMENT

The STOP statement is shown in figure 4-35.

STOP[n]

n **Is a string of 1 through 5 decimal digits, or a character constant of at most 70 characters.**

Figure 4-35. STOP Statement

A STOP statement terminates program execution. When a STOP statement is encountered during execution, STOP n is displayed in the dayfile (or at the terminal if executing interactively), the program terminates, and control returns to the operating system. If n is omitted, blanks are implied. A program unit can contain more than one STOP statement.

Example:

```
STOP 'PROGRAM HAS ENDED'
```

END STATEMENT

The END statement is shown in figure 4-36.

END

Figure 4-36. END Statement

The END statement indicates the end of the program unit to the compiler. Every program unit must physically terminate with an END statement. The END statement can be labeled. If control flows into or branches to an END statement in a main program, execution terminates. If control flows into or branches to an END statement in a function or subroutine, it is treated as if a RETURN statement had preceded the END statement.

An END statement cannot be continued; it must be completely contained on an initial line. A line following an END statement is considered to be the first line of the next program unit, even if it has a continuation character in column 6.

When the END statement is used in an overlay, it terminates that overlay and causes a return to the previous overlay.

RETURN STATEMENT

The RETURN statement is described in section 6, Program Units and Procedures.

CALL STATEMENT

The CALL statement is described in section 6, Program Units and Procedures.

Processing that results from input/output statements depends on the type of statement used. For each category, there are one or more input statements and corresponding output statements. The categories are:

Formatted (READ, WRITE, PRINT and PUNCH statements with format specifier)

Unformatted (READ and WRITE without format specifier)

NAMELIST (READ, WRITE, PRINT and PUNCH with the NAMELIST group name replacing the format specifier)

List directed (READ, WRITE, PRINT, and PUNCH with an * replacing the format specifier)

Buffered (BUFFER IN and BUFFER OUT)

Mass storage input/output (Subroutines READMS, WRITMS, OPENMS, CLOSMS, and STINDX; see section 7)

CYBER Record Manager interface routines (See section 8)

In addition, there are file status statements OPEN, INQUIRE, and CLOSE, the memory-to-memory transfer statements ENCODE and DECODE, and the file positioning statements REWIND, BACKSPACE, and ENDFILE, all discussed in this section. Format specifications, input/output lists, and internal files which provide for memory-to-memory transfer of data, are also discussed in this section.

Subprograms used in connection with input/output, besides the mass storage routines and the CYBER Record Manager routines, include EOF, IOCHEC, UNIT, LENGTH, and LENGTHX. These subprograms are discussed in section 7.

FILE USAGE

Input and output involve reading records from files and writing records to files. Every file must have a logical file name of one to seven letters and digits beginning with a letter. The logical file name is defined only for the current job, and is the name by which the file is referred to in control statements.

For batch jobs (jobs not executed interactively at a terminal), certain file names have a predefined origin or destination. These file names are:

INPUT	Data from user's source deck
OUTPUT	Printed at job termination
PUNCH	Punched in Hollerith format at job termination
PUNCHB	Punched in binary format at job termination

The files INPUT, OUTPUT, and PUNCH should be processed only by formatted, list directed, or namelist input/output statements.

The predefined meaning of any file name except INPUT can be overridden by control statements.

Sequential files need not be declared on the PROGRAM or an OPEN statement. If a file is not declared on the PROGRAM or OPEN statement, a buffer is created automatically on the first reference to the file. Files processed by CYBER Record Manager interface routines, however, must not be declared on the PROGRAM statement. The PROGRAM and OPEN statements also allow the user to specify maximum record length and buffer size for a file. In the absence of user specification, default values are provided.

Mixing types of operations on the same file can sometimes lead to destruction of file integrity. In particular, files processed by mass storage or CYBER Record Manager interface subroutines should be processed only by these routines. Files processed by buffer statements should be processed only by the buffer statements in a given program (REWIND, ENDFILE, and BACKSPACE are permitted for files processed by CYBER Record Manager subroutines or buffer statements).

A file should not be processed both by unformatted operations and by formatted, namelist, or list directed operations without an intervening rewind. If rewound, it can then be rewritten in a different mode.

If formatted, list directed, or NAMELIST input/output is performed on a 7-track S or L tape, a FILE control statement that specifies CM=NO (appendix F) must be included in the job.

For every formatted, list directed, namelist, or unformatted READ, end-of-file status can be checked by use of the END= or IOSTAT= parameter in the READ statement. If end-of-file is encountered and a test is not included, the program terminates with a fatal error.

Record length on card files should not exceed 80 characters. Record length on print files should not exceed 137 characters; the first character is always used as carriage control and is not printed. The second character appears in the first print position. Carriage control characters are listed in this section under Format Processing.

The following keyword=value forms are used in input/output statements.

UNIT= u — Specifies the FORTRAN unit or internal file to be used. The unit name is derived from u depending on its value. The u can be one of the following:

An asterisk implying unit INPUT in a READ statement and unit OUTPUT in a WRITE statement.

The name of a character variable, array, array element, or substring identifying an internal file.

An integer or Boolean expression having the following characteristics:

INT(u) has a value in the range 0 through 999. The compiler associates these numbers with unit names of the type TAPEu.

or

BOOL(u) is a display code name in L format (left-justified with binary zero fill). This is the unit name. If this value is of the form TAPEk, where k is an integer in the range 0 through 999, with no leading zero, it is equivalent to the integer k for the purpose of identifying external units. A valid unit name consists of one through seven letters or digits beginning with a letter.

The characters UNIT= can be omitted, in which case u must be the first item in the control information list.

File names default to the unit name unless a different file name has been specified using execution control statement substitution, PROGRAM statement equivalencing, or an OPEN statement.

FMT=fn Specifies a format to be used for formatted input/output; fn can be one of the following:

A statement label identifying a FORMAT statement in the program unit containing the input/output statement.

The name of a character array, variable, expression or array element containing the format specification.

A noncharacter array containing the format specification.

An integer variable that has been assigned the statement number of a FORMAT statement by an ASSIGN statement.

An asterisk, indicating list directed I/O.

A NAMELIST group name

The characters FMT= can be omitted, in which case the format designator must be the second item in the control information list, and the first item must be the unit specifier without the characters UNIT=.

REC=rn Specifies the number of the record to be read or written in the file; must be a positive nonzero integer. Valid for files opened for direct access only.

END=sl Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation.

ERR=sl Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

IOSTAT=ios Specifies an integer variable into which one of the following values is placed after the input/output operation is complete:

<0	End-of-file
=0	Operation completed normally
>0	Number of error condition detected (see appendix B, table B-4).
>1000	CRM error; the rightmost 3 digits correspond to an octal error code in the CYBER Record Manager reference manual. For example, error number 1355 corresponds to CRM error number 355.

iolist Input/output list specifying items to be transmitted (described under Input/Output Lists).

FORMATTED INPUT/OUTPUT

For formatted input/output, a format specifier must be present in the input/output statement. The input/output list is optional. Each formatted input/output statement transfers one or more records. The formatted input/output statements are READ, WRITE, PRINT, and PUNCH.

INPUT/OUTPUT LISTS

The list portion of an input/output statement specifies the items to be read or written and the order of transmission. The input/output list can contain any number of items. List items are read or written sequentially from left to right.

If no list appears on input, one or more records are skipped. Only information completely contained within the FORMAT statement, such as character strings, can be output with a null (empty) output list.

A list item consists of a variable name, an array name, an array element name, a character substring name, or an implied DO list. On output the list items can also include character, Boolean, logical, or arithmetic expressions. No expression in an input/output list can reference a function if such reference would cause any input/output operations to be executed or would cause the value of any element of the input/output statement to be changed. List items are separated by commas.

An array name without subscripts in an input/output list specifies the entire array in the order in which it is stored. The entire array (not just the first word of the array) is read or written. Assumed-size array names are illegal in input/output lists.

Subscripts in an input/output list can be any valid subscript (section 1).

Example of input/output lists:

```
READ (2,100) A,B,C,D
READ (3,200) A,B,C(I),D(3,4),E(I,J,7),H
READ (4,101) J,A(J),I,B(I,J)
WRITE (2,202) DELTA
WRITE (4,102) DELTA(5*J+2,5*I-3,5*K),C,D(I+7)
```

On formatted input or output, the I/O list is scanned and each item in the list is paired with the field specification provided by the FORMAT statement. After one item has been input or output, the next format specification is taken together with the next element of the list, and so on until the end of the list.

Example:

```
    READ (5,20) L,M,N
20  FORMAT (I3,I2,I7)
```

Input record:

```
100223456712
```

100 is read into the variable L under the specification I3. 22 is read into M under the specification I2, and 3456712 is read into N under specification I7.

IMPLIED DO LOOP IN I/O LIST

An implied DO specification has the following form:

(dlist,i=e_1,e_2[,e_3])

The elements i, e_1,e_2, and e_3 have the same meaning as in the DO statement, and dlist is an input/output list. The range of an implied DO specification is that of dlist. The value of i must not be changed within the range of the implied DO list by a READ statement. Changes to the values of e_1, e_2, and e_3 have no effect upon the execution of the implied DO. However, their values can be changed in a READ statement if they are outside the range of the implied DO, and the change does have effect. For example:

```
READ 100, K, (A(I),I=1,K)
```

reads a value into K and uses that value as the terminal parameter of the implied DO.

The statements:

```
    K=2
    READ 100, (A(I),I=1,K)
100 FORMAT (F10.3)
```

read two records, each containing a value for A.

An implied DO loop can be used to transmit a simple variable more than one time. For example, the list (A(K),B,K=1,5) causes the variable B to be transmitted five times.

Input/output of array elements can be accomplished by using an implied DO loop. The list of variables followed by the DO loop index is enclosed in parentheses to form a single element of the input/output list.

Example:

```
READ (5,100) (A(I),I=1,3)
```

has the same effect as the statement:

```
READ (5,100) A(1),A(2),A(3)
```

Example:

```
WRITE (3,20) (CAT,DOG,RAT,I=1,10)
```

CAT, DOG, and RAT are written 10 times each.

A variable cannot be used as a control variable more than once in the same implied DO nest, but iolist items can appear more than once. The value of a control variable within an implied DO specification is defined within that specification. On exit from the implied DO specification the control variable retains the first value to exceed the upper limit (e_2).

Implied DO loops can be nested, that is, the iolist in an implied DO can itself contain implied DO loops. The first (innermost) control variable varies most rapidly, and the last (outermost) control variable varies least rapidly. For example, a nested implied DO with two levels has the form:

((list,v_1=e_1,e_2,e_3),v_2=ee_1,ee_2,ee_3)

Nested implied DO loops are executed in the same manner as nested DO statements.

The nested form can be used to read into and write from arrays.

Example:

```
READ (2,100) ((A(JV,JX),JV=2,20,2),JX=1,30)
READ (2,300) (((ITMLIST(I,J+1,K-2),I=1,25),J=2,N),
*K=IVAR,IVMAX,4)
```

Example:

```
    DIMENSION VECTOR(3,4,7)
    READ (3,100) VECTOR
100 FORMAT (I6)
```

is equivalent to the following:

```
DIMENSION VECTOR(3,4,7)
READ (3,100) (((VECTOR(I,J,K),I=1,3),J=1,4),K=1,7)
```

The following statement transmits nine elements into the array E in the order: E(1,1), E(1,2), E(1,3), E(2,1), E(2,2), E(2,3), E(3,1), E(3,2), E(3,3):

```
READ (1,100) ((E(I,J),J=1,3),I=1,3)
```

Each execution of an input or output statement transmits at least one record regardless of the FORMAT statement. Once a READ is initiated, the FORMAT statement determines when a new record will be transmitted. For example:

```
      READ (5,100) (VECTOR (I),I=1,10)
  100 FORMAT (F7.2)
```

reads data (consisting of one constant per record) into the first 10 elements of the array VECTOR. The following statements have the same effect:

```
      DO 40 I = 1,10
   40 READ (5,100) VECTOR (I)
  100 FORMAT (F7.2)
```

In this example, numbers are read, one from each record, into the elements VECTOR(1) through VECTOR(10) of the array VECTOR. The READ statement is encountered each time the DO loop is executed; and a new record is read for each element of the array.

If statement 100 FORMAT (F7.2) had been 100 FORMAT (4F7.2), only three records would be read by the first example; the second example would still read ten records. Both examples would read ten values.

FORMATTED READ

The formatted READ statement is shown in figure 5-1.

```
READ ([UNIT=]u,[FMT=]fn[,IOSTAT=ios]
      [,ERR=sl] [,END=sl])[iolist]

READ fn[,iolist]
```

Figure 5-1. Formatted READ Statement

These statements transmit data from unit u, or the unit INPUT (the second form of read), to storage locations named in iolist according to FORMAT specification fn. The number of items in the list and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, one or more input records will be bypassed. The number of records bypassed is one plus the number of slashes interpreted in the FORMAT statement.

The user should specify the END= or IOSTAT= parameter to avoid termination when an end-of-file is encountered. If an attempt is made to read on unit u and an end-of-file was encountered on the previous read operation on this unit, execution terminates and an error message is printed. Records following an end-of-file can be read by issuing a CLOSE followed by an OPEN on the file or by using the EOF function (section 7). CLOSE/OPEN, described later in this section, is the preferred method.

Examples of formatted READ statements are shown in figure 5-2.

Example 1:

```
      PROGRAM IN
      OPEN (4, FILE='INPUT')
      OPEN (7, FILE='OUTPUT')
      READ (4,200)A,B,C
  200 FORMAT (3F7.3)
      A=B*C+A
      WRITE (7,50) A
   50 FORMAT (50X,F7.4)
      STOP
      END
```

The READ statement transfers data from logical unit 4 (externally, the file INPUT) to the variables A, B, and C, according to the specifications in the FORMAT statement labeled 200.

Example 2:

```
      PROGRAM RLIST
      READ 5,X,Y,Z
    5 FORMAT (3G20.2)
      RESULT = X-Y+Z
      PRINT 100, RESULT
  100 FORMAT (10X,G10.2)
      STOP
      END
```

The READ statement transfers data from file INPUT to the variables X, Y, and Z, according to the specifications in the FORMAT statement labeled 5. Result is printed on file OUTPUT.

Example 3:

```
      PROGRAM READ (INPUT,OUTPUT,TAPE2=INPUT,
     +TAPE3=OUTPUT)
      READ (2,100,ERR=16,END=18) A,B
  100 FORMAT (2F10.4)
      C=A+B
      PRINT *,A,B,C
      STOP
   16 PRINT 101
  101 FORMAT ('ΔI/O ERROR')
      STOP
   18 PRINT 102
  102 FORMAT ('ΔEND OF FILE')
      STOP
      END
```

Variables are read according to the FORMAT statement labeled 100. If an error occurs during the read, control transfers to statement 16; if an end-of-file is encountered, control transfers to statement 18.

Example 4:

In example 3, the READ and FORMAT statements can be combined as follows:

```
READ (2,'(2F10.4)',ERR=16,END=18)A,B
```

Figure 5-2. READ Statement Examples

FORMATTED WRITE

The formatted WRITE statement is shown in figure 5-3.

```
WRITE ([UNIT=]u,[FMT=]fn[,IOSTAT=ios]
      [,ERR=sl])[iolist]
```

Figure 5-3. Formatted WRITE Statement

The formatted WRITE statement transfers information from the storage locations named in the input/output list to the unit specified by u, according to the FORMAT specification, fn.

Examples:

```
      WRITE (4,50)
   50 FORMAT ('THE IOLIST CAN BE OMITTED')

      WRITE (*,FMT=12) L,M,S(3)
   12 FORMAT (3E16.5)
```

In the following example, the format specification appears in the WRITE statement:

```
      WRITE (2,'(2E16.5)',ERR=12) X,Y
```

Figure 5-4 shows a program segment containing a WRITE statement.

```
      PROGRAM RITE
      X=2.1
      Y=3.
      M=7
      WRITE (6,100,ERR=200) X,Y,M
  100 FORMAT (2F6.2,I4)
  200 STOP
      END
```

Figure 5-4. WRITE Statement Example

FORMATTED PRINT

The PRINT statement is shown in figure 5-5.

```
PRINT fn [,iolist]
```

Figure 5-5. PRINT Statement

This statement transfers information from the storage locations named in the input/output list to the file named OUTPUT according to the specified format. At the end of a batch job, file OUTPUT is normally sent to the printer.

Example:

```
      PROGRAM PRINT
      CHARACTER B*3
      A=1.2
      B='YES'
      N=19
      PRINT 4,A,B,N
    4 FORMAT (G20.6,A,I5)
      STOP
      END
```

FORMATTED PUNCH

The PUNCH statement is shown in figure 5-6.

```
PUNCH fn [,iolist]
```

Figure 5-6. PUNCH Statement

Data is transferred from the storage locations specified by iolist to the file PUNCH. At the end of a batch job the file PUNCH is output on the standard punch unit as Hollerith codes, 80 characters or fewer per card in accordance with format specification fn. If the card image is longer than 80 characters, additional cards are punched with the remaining characters.

Examples:

```
      PUNCH 5,A,B,C,ANSWER
    5 FORMAT (3G12.6,G20.6)

      PUNCH 30
   30 FORMAT ('LAST CARD')
```

FORMAT SPECIFICATION

Format specifications are used in conjunction with formatted input/output statements to produce output or read input that consists of strings of display code characters. On input, data is converted from a specified format to its internal binary representation. On output, data is converted from its internal binary representation to the specified format before it is transmitted. Formats can be specified by:

The statement label of a FORMAT statement.

An integer variable which has been assigned the statement label of a FORMAT statement (see ASSIGN Statement, section 4).

A character array name or any character expression, except one involving assumed-length character entities.

A noncharacter array name.

FORMAT STATEMENT

The FORMAT statement is shown in figure 5-7.

```
sl  FORMAT (flist)
```

sl Is a statement label.

flist Is a list of items, separated by commas, having the following forms:

```
[r]ed
ned
[r](flist)
```

ed Is a repeatable edit descriptor.

ned Is a nonrepeatable edit descriptor.

r Is a nonzero unsigned integer constant repeat specification.

Figure 5-7. FORMAT Statement

FORMAT is a nonexecutable statement which specifies the formatting of data to be read or written with formatted I/O. It is used in conjunction with formatted input and output statements, and it can appear anywhere in the program after the PROGRAM, FUNCTION or SUBROUTINE statement. An example of a READ statement and its associated FORMAT statement is as follows:

```
      READ (5,100) INK,NAME,AREA
  100 FORMAT (10X,I4,I2,F7.2)
```

The format specification consists of edit descriptors in parentheses. Blanks are not significant except in H, quote, and apostrophe descriptors.

Generally, each item in an input/output list is associated with a corresponding edit descriptor in a FORMAT statement. The FORMAT statement specifies the external format of the data and the type of conversion to be used. Complex variables always correspond to two edit descriptors. Double precision variables correspond to one floating-point edit descriptor (D,E,F,G). The D edit descriptor corresponds to exactly one list item. Complex editing requires two (D,E,F,G) descriptors; the two descriptors can be different.

The type of conversion should correspond to the type of the variable in the input/output list. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete. For example, the statements:

```
      INTEGER N
      READ (5,100) N
  100 FORMAT (F10.2)
```

assign a floating point number to the variable N which could cause unpredictable results if N is referenced later as an integer.

CHARACTER FORMAT SPECIFICATIONS

A format specification can also be specified as a character expression or as the name of a character variable or array containing a format specification. The form of these format specifications is the same as for FORMAT statements without the keyword FORMAT. Any character information beyond the terminating parenthesis is ignored. The initial left parenthesis can be preceded by blanks.

Example:

```
      CHARACTER FORM*11
      DATA FORM/'(I3,2E14.4)'/
      READ (2, FMT=FORM,END=50) N,A,B
```

is equivalent to:

```
      READ (2,FMT=100,END=50) N,A,B
  100 FORMAT (I3,2E14.4)
```

The preceding examples can also be expressed as:

```
      READ (2, FMT='(I3, 2E14.4)',END=50) N,A,B
```

or

```
      CHARACTER FORM*(*)
      PARAMETER (FORM='(I3,2E14.4)')
      READ (2,FMT=FORM,END=50)N,A,B
```

If a format specification is contained in a character array, the specification may cross element boundaries. Only the array name need be specified in the input/output statement; all information up to the closing parenthesis is considered to be part of the format specification. For example:

```
      CHARACTER AR(2)*10
      DATA AR/'(10X,2I2,1','0X,F6.2)'/
      READ (5,AR) I,J,X
```

is equivalent to:

```
      READ (5,000) I,J,X
  100 FORMAT (10X,2I2,10X,F6.2)
```

NONCHARACTER FORMAT SPECIFICATION

Format specifications can be contained in a noncharacter array. The rules for noncharacter format specifications are the same as for character format specifications.

EDIT DESCRIPTORS

Format specifications are composed of edit descriptors which specify the data conversions to be performed. Tables 5-1 and 5-2 list the edit descriptors and give a brief description of each. The descriptors listed in table 5-1 can be preceded by an unsigned nonzero decimal integer indicating the number of times the descriptor is to be repeated (as described later in this section under Repeated Edit Descriptors). Uppercase letters indicate the type of conversion. Lowercase letters indicate user-supplied information that has the following meaning:

- w Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading blanks, + or - signs, decimal point, and exponent.
- d Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded.
- e Nonzero unsigned integer constant specifying the number of digits in the exponent; the value of e cannot exceed six.
- m Unsigned integer constant specifying the minimum number of digits to be output.
- k Integer constant scale factor.
- n Positive nonzero decimal integer.

The field width w must be specified for all conversion codes except A.

Field separators are used to separate descriptors and groups of descriptors. The format field separators are the slash (/), the comma, and the colon. The slash is also used to specify demarcation of formatted records.

Leading blanks are not significant in numeric input conversions; other blanks in numeric conversions are ignored unless BLANK='ZERO' was specified for the file on an OPEN statement or a BZ edit descriptor is in effect. Plus signs can be omitted. An all-blank field is considered to be zero, except for logical input, where an all-blank field is considered to be FALSE.

TABLE 5-1. REPEATABLE EDIT DESCRIPTORS

Descriptor	Descriptor Type	Description
Ew.d	Numeric	Single precision floating-point with exponent
Ew.dEe		Single precision floating-point with explicitly specified exponent length
Fw.d		Single precision floating-point without exponent
Dw.d		Double precision floating-point with exponent
Gw.d		Single precision floating-point with or without exponent
Gw.dEe		Single precision floating-point with or without explicitly specified exponent length
Iw		Decimal integer
Iw.m		Decimal integer with minimum number of digits
Lw	Logical	Logical
A	Character	Character with data-dependent length
Aw	Character or Boolean	Character or Boolean with specified length
Rw	Boolean	Boolean conversion
Ow		Octal integer
Ow.m		Octal integer with leading zeros and minimum number of digits
Zw		Hexadecimal integer
Zw.m		Hexadecimal with leading zeros and minimum number of digits

For the E, F, G, and D input conversions, a decimal point in the input field overrides the decimal point specification of the field descriptor.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field unless w.m is specified, in which case leading zeros are produced as necessary. The number of characters produced by an output conversion must not be greater than the field width. If the field width is exceeded, asterisks are inserted throughout the field.

TABLE 5-2. NONREPEATABLE EDIT DESCRIPTORS

Descriptor	Descriptor Type	Description
SP	Numeric output control	Plus signs (+) produced.
SS		Plus signs (+) suppressed.
S		Plus signs (+) suppressed.
nX	Tabulation control	Position forward.
Tn		Position forward or backward.
TRn		Position forward.
TLn		Position backward.
nH	Character output	Output character string.
"		
'		
:	Format control	Terminate format control.
/	End of record	Indicates end of current input or output record.
kP	Scale factor	Scaling for numeric editing.
BN	Numeric input control	Blanks ignored.
BZ		Blanks treated as zeros

Complex data items are converted on input/output as two independent floating-point quantities. The format specification uses two conversion elements.

Example:

```
      COMPLEX A,B,C,D
      WRITE (6,10)A
10    FORMAT (F7.2,E8.2)
      READ (5,11) B,C,D
11    FORMAT (2E10.3,2(F8.3,F4.1))
```

Different types of data can be read by the same FORMAT specification. For example:

```
10 FORMAT (I5,F15.2)
```

specifies two numbers, the first of type integer, the second of type real.

Example:

```
      CHARACTER R*4
      READ (5,15) NO,NONE,INK,A,B,R
   15 FORMAT (3I5,2F7.2,A4)
```

reads three integer values, two real values, and one character string.

Following are descriptions of the edit descriptors.

I Descriptor

The I descriptor specifies integer conversion. This descriptor has the forms:

Iw Iw.m

Input

The plus sign can be omitted for positive integers. When a sign appears, it must precede the first digit in the field. An Iw.m specification has no effect on input. An all blank field is considered to be zero. Decimal points are not permitted. The value is stored in the specified variable. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will cause an error.

Example:

```
      OPEN (2,BLANK='NULL')
      READ (2,10) I,J,K,L,M,N
   10 FORMAT (I3,I7,I2,I3,I2,I4)
```

Input Record:

```
  139  -15  18  7   1 4
```

In memory:

I contains 139	L contains 7
J contains -15	M contains 0
K contains 18	N contains 14

If BLANK='ZERO' were specified on the OPEN statement, J would contain -1500 and N would contain 104. Other values would not be affected. (The OPEN statement is described later in this section.)

Output

If the integer is positive, the plus sign is suppressed unless an SP specification is in effect. Leading zeros are suppressed.

If Iw.m is used and the output value occupies fewer than m positions, leading zeros are generated to fill up to m digits. If m=0, a zero value will produce all blanks. If m=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. If the field is too short, asterisks occupy the field.

Figure 5-8 shows some examples of I output. Note that the first character of a printer output record is used for carriage control and is not printed. More information on carriage control appears later in this section.

Example 1:

```
      PRINT 10,I,J,K
   10 FORMAT (I9,I10,I5)
```

I contains -3762
J contains +4762937

K contains +13

Printed result:

```
ΔΔΔ-3762|ΔΔΔ4762937|ΔΔ013|
   8         10        5
```

First blank taken as printer control character

Example 2:

```
      WRITE (6,100)N,M,I
  100 FORMAT (I5,I6,I9)
```

N contains +20
M contains -731450
I contains +205

Printed result:

```
ΔΔ20|******|ΔΔΔΔΔ205|
 4    6      9
```

First blank taken as printer control character

Specification too small, * indicates field is too short

Figure 5-8. I Output Examples

E Descriptor

The E descriptor specifies conversion between an internal real or double precision value and an external number written with an exponent. This descriptor has the forms:

Ew.d Ew.dEe

Input

The width w includes plus or minus signs, digits, decimal point, E, and exponent. If an external decimal point is not provided, d acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is:

(integer subfield) $\times 10^{-d} \times 10$ (exponent subfield)

For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as: $3267 \times 10^{-8} \times 10^{5} = 3.267$.

If an external decimal point is provided, it overrides d; e, if specified, has no effect on input. An input field consisting entirely of blanks is interpreted as zero.

The diagram in figure 5-9 illustrates the structure of the input field. It shows the characters allowed to start a subfield.

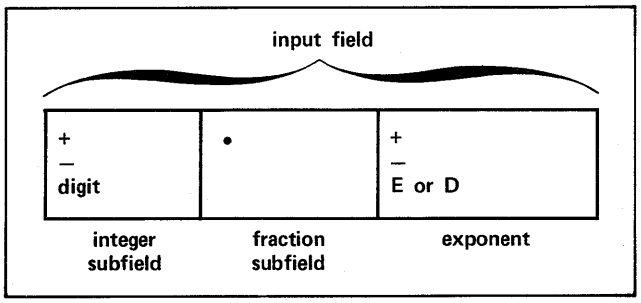


Figure 5-9. E Input Field

The integer subfield begins with a + or - sign, a digit, or a blank; and it can contain a string of digits. The integer field is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, - or the end of the input field. It can contain a string of digits.

The exponent subfield can begin with E, + or -. When it begins with E, the + is optional between E and the string of digits in the subfield. For example, the following are valid equivalent forms for the exponent 3:

E+03 E03 E03 E3 +3

The range, in absolute value, of permissible values is approximately 10^{-293} to 10^{322}. Numbers below the range are treated as zero; numbers above the range cause a fatal error message.

Valid subfield combinations are as follows:

+1.6327E-04	Integer-fraction-exponent
-32.7216	Integer-fraction
+328+5	Integer-exponent
.629E-1	Fraction-exponent
+136	Integer only
136	Integer only
.07628431	Fraction only
E-06 (interpreted as zero)	Exponent only

If the field length specified by w in Ew.d is not the same as the length of the field containing the input number, incorrect numbers might be read, converted, and stored. The example in figure 5-10 illustrates a situation where numbers are read incorrectly, converted, and stored; yet there is no immediate indication that an error has occurred. First, +647E-01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number -2.36+5 is read instead of -2.36. The specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input number. Since the second specification incorrectly took two digits from the third number, the specification for the third number is now incorrect. The field .321E+02ΔΔ is read. The OPEN statement specifies that trailing blanks are to be treated as zeros; therefore the number .321E+0200 is read converted and placed in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

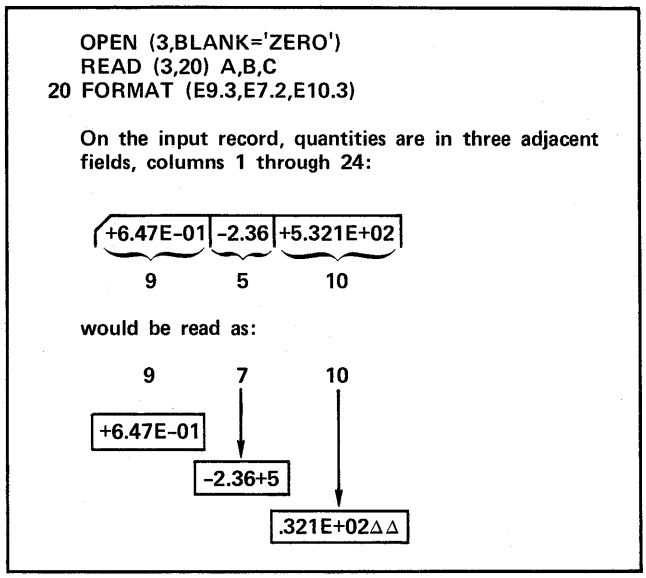


Figure 5-10. Example Showing E Input Incorrectly Read

Some additional examples of Ew.d input specifications are shown in figure 5-11.

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	0.14326	All subfields present.
327.625	E7.3	327.625	No exponent subfield.
-.0003627+5	E11.7	-36.27	Integer subfield only a minus sign and a plus sign appears instead of E.
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only.
blanks	E4.1	0.	All subfields empty.
E+06	E10.6	0.	No integer or fraction subfield: zero stored regardless of exponent field contents.

Figure 5-11. Ew.d Input Examples

Output

The width w, must be sufficient to contain digits, plus or minus signs, decimal point, E, the exponent, and blanks. Generally, $w \geq d+6$ or $w \geq d+e+4$ for negative numbers and $w \geq d+5$ or $w \geq d+e+3$ for positive numbers. Positive numbers need not reserve a space for the sign of the number unless an SP specification is in effect. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

The Ew.d specification produces output in the following formats:

s.a...aE ± ee	For values where the magnitude of the exponent is less than one hundred
s.a...a ± eee	For values where the magnitude of the exponent exceeds one hundred
s	Is a minus sign if the number is negative, and omitted if the number is positive.
a...a	Are the most significant digits of the value correctly rounded.

When the specification Ew.dEe is used, the exponent is preceded by E, and the number of digits used for the exponent field not counting the letter and sign is determined by e. If e is specified too small for the value being output, the entire field width as specified by w will be filled with asterisks.

If an integer variable is output under the Ew.d specification, results are unpredictable since the internal formats of real and integer values differ. An integer value normally does not have an exponent and will be printed, therefore, as a very small value or 0.0.

Example:

```
      WRITE (2,10)A     A contains -67.32 or +67.32
   10 FORMAT (E9.3)

      Result:           -.673E+02 or Δ.673E+02
```

Example:

```
      WRITE (2,10)A
   10 FORMAT (E12.3)

      Result:           ΔΔΔ-.673E+02 or ΔΔΔΔ.673E+02
```

F Descriptor

The F descriptor specifies conversion between an internal real or double precision number and an external floating-point number without an exponent. This descriptor has the form:

Fw.d

Input

On input the F specification is treated identically to the E specification. Some examples are shown in figure 5-12.

Output

The F descriptor outputs a real number without a decimal exponent.

The plus sign is suppressed for positive numbers. If the field is too short, all asterisks appear in the output field. If the field is longer than required, the number is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range (exceeds the capacity of the machine), an R is printed.

The specification Fw.d outputs a number in the following format:

sn.n

n	Is a field of decimal digits.
s	Is a minus sign if the number is negative, or omitted if the number is positive.

Some examples of F output are shown in figure 5-13.

G Descriptor

The G descriptor specifies conversion between an internal real or double precision number and an external floating-point number written either with or without an exponent, depending on the magnitude of the number. This descriptor has the forms:

Gw.d Gw.dEe

Input

Input under control of G specification is the same as for the E specification. The rules which apply to the E specification also apply to the G specification. For example:

```
      READ (5,11) A,B,C
   11 FORMAT (G13.6,2G12.4)
```

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field.
.62543	F6.5	.62543	No integer subfield.
.62543	F6.2	.62543	Decimal point overrides d of specification.
+144.15E-03	F11.2	.14415	Exponents are allowed in F input.
50000	F5.2	500.00	No fraction subfield; input number converted as $50\,000 \times 10^{-2}$.
ΔΔΔΔΔ	F5.2	0	Blanks in input field interpreted as 0.

Figure 5-12. F Input Examples

Value of A	FORMAT Statement	PRINT Statement	Printed Result
+32.694	10 FORMAT (1H ,F6.3)	PRINT 10,A	32.694
+32.694	11 FORMAT (1H ,F10.3)	PRINT 11,A	ΔΔΔΔ32.694
-32.694	12 FORMAT (1H ,F6.3)	PRINT 12,A	******
.32694	13 FORMAT (1H ,F4.3,F6.3)	PRINT 13,A,A	.327ΔΔ.327
32.694	14 FORMAT (1H ,F6.0)	PRINT 14,A	ΔΔΔ33.

The specification 1H is the carriage control character.

Figure 5-13. F Output Examples

Output

Output under control of the G descriptor depends on the size of the floating-point number being edited. For values in the range greater than or equal to .1 and less than 10**d the number is output under F format. For values outside this range, Gw.d output is identical to Ew.d and Gw.dEe is identical to Ew.dEe.

If a number is output under the Gw.d specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field E±ee). Therefore, for output under G conversion, w must be greater than or equal to d+6. The six extra spaces are required for sign and decimal point plus four spaces for the exponent field. If the Gw.dEe form is used for a number output without an exponent, then e+2 spaces are inserted to the right of the field. For example:

```
      Y=77.132
      WRITE (7,200)Y
  200 FORMAT (G10.3)
```

writes the following:

```
ΔΔ77.1ΔΔΔΔ
```

```
      EXIT=1214635.1
      WRITE (4,100) EXIT
  100 FORMAT (G10.3)
```

writes the following:

```
.121E+07
```

Additional examples of G output are shown in figure 5-14.

```
      READ (5,50) SAMPLE
   50 FORMAT (E20.5)
      :
      WRITE (6,60) SAMPLE
   60 FORMAT (1X,G14.8)
```

Data Read By READ Statement	Data Printed	Format Option
.1415926535Δ E-10	Δ.14159265E-10	E conversion
ΔΔΔ.8979323846	Δ.89793238	F conversion
ΔΔΔ2643383279.	Δ.26433833E+10	E conversion
ΔΔΔ-693.9937510	-693.99375	F conversion

Figure 5-14. G Output Examples

D Descriptor

The D descriptor specifies conversion between an internal double precision real number and an external floating-point number written with an exponent. This descriptor has the form:

Dw.d

NOTE

The E descriptor is preferred over the D descriptor.

Input

D editing corresponds to E editing and can be used to input all the same forms as E.

The diagram in figure 5-15 illustrates the structure of the input field. It shows the characters allowed to start a subfield.

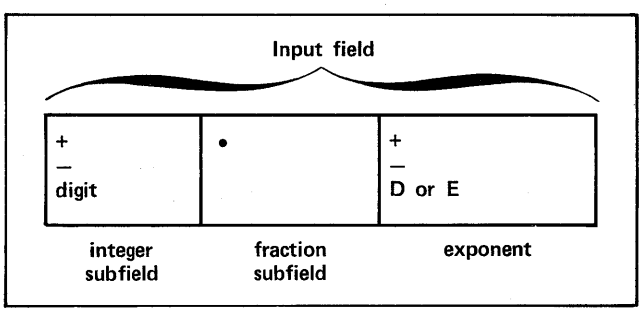


Figure 5-15. D Input Field

Output

Type D conversion is used to output double precision values. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

Examples of type D output:

```
      DOUBLE PRECISION A,B,C
      A = 111111.11111D0
      B = 222222.22222D0
      C = A + B
      WRITE (2,10) A,B,C
   10 FORMAT (3D23.11)

      .11111111111D+06     .22222222222D+06
      .33333333333D+06
```

The specification Dw.d produces output in the following format:

s.a±eee	For values where the magnitude of the exponent exceeds one hundred
s.aD±ee	For values where the magnitude of the exponent is less than one hundred
s	Minus sign if the number is negative, or blank if the number is positive
a	One or more most significant digits
ee	Digits in the exponent

P Descriptor

The P descriptor has the form:

kP

where k is a signed or unsigned integer constant called the scale factor.

The P descriptor is used to change the position of a decimal point of a real number when it is input or output. Scale factors can precede D, E, F, and G format specifications or appear independently. Forms are as follows:

kPDw.d

kPEw.dEe

kPEw.d

kPFw.d

kPGw.d

kP

A scale factor of zero is established when each FORMAT specification is first referenced; it holds for all F, E, G, and D field descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G descriptors in that FORMAT specification until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G descriptors a zero scale factor (0P) must be specified.

Example:

```
   15 FORMAT(2P,E14.3,F10.2,G16.2,0P,4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specifications. The 0P scale factor restores normal scaling ($10^0 = 1$) for the subsequent specification 4F13.2.

Example:

```
   20 FORMAT(3P,5X,E12.6,F10.3,0PD18.7,-1P,F5.2)
```

E12.6 and F10.3 specifications are scaled by 10^3. The D18.7 specification is not scaled, and the F5.2 specification is scaled by 10^{-1}.

The specification (3P,3I9,F10.2) is the same as the specification (3I9,3PF10.2).

Input

For F, E, D, and G editing, provided that the number in the input field does not have an exponent, the number is divided by 10^k and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is $314.1592 \times 10^{-2} = 3.141592$. However, if an exponent is read the scale factor is ignored.

Output

For F editing, the number in the output field is the internal number multiplied by 10^k. In the output representation, the decimal point is fixed; the number is adjusted to the left or right, depending on whether the scale factor is plus or minus. For example, the internal number-3.1415926536 can be represented on output under scaled F specifications as shown in figure 5-16.

```
.............................
(-1PF13.6)      -.314159
(   F13.6)     -3.141593
( 1PF13.6)    -31.415927
( 3PF13.6)  -3141.592654
.............................
```

Figure 5-16. Scaled F Output

For E and D editing, the effect of the scale factor kP is to shift the output coefficient left k places and reduce the exponent by k. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent such that: if $k \leq 0$, there will be exactly -k leading zeros and d + k significant digits after the decimal point; if $k > 0$, there will be exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point. For example, the number -3.1415926536 is represented on output under the indicated Ew.d scaling as shown in figure 5-17.

```
.............................
(-3PE20.4)          -.0003E+04
(-1PE20.4)          -.0314E+02
(   E20.4)          -.3142E+01
( 1PE20.4)         -3.1416E+00
( 3PE20.4)         -314.16E-02
.............................
```

Figure 5-17. Scaled E Output

For G editing, the effect of the scale factor is nullified unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number N<10-1 or N≥10d). In these cases, the scale factor has the same effect as described for Ew.d and Dw.d scaling. For example, the numbers -3.1415926536 and -.00031415926536 are represented on output under the indicated Gw.d scaling as shown in figure 5-18.

```
.............................
(-3PG20.6)          -3.14159
(-1PG20.6)          -3.14159
(   G20.6)          -3.14159
( 1PG20.6)          -3.14159
( 3PG20.6)          -3.14159
( 5PG20.6)          -3.14159
( 7PG20.6)          -3.14159
.............................

.............................
(-3PG20.6)          -.000314E+00
(-1PG20.6)          -.031416E-02
(   G20.6)          -.314159E-03
( 1PG20.6)          -3.141593E-04
( 3PG20.6)          -314.1593E-06
( 5PG20.6)          -31415.93E-08
( 7PG20.6)          -3141593.E-10
.............................
```

Figure 5-18. Scaled G Output

BN and BZ Blank Interpretation

The BN and BZ descriptors can be used with the I, F, E, D, and G edit descriptors, on input, to specify the interpretation of blanks (other than leading blanks). In the absence of a BN or BZ descriptor, blanks in input fields are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier currently in effect for the input/output unit. BLANK='NULL' is the default for input. If a BN descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are ignored; that is, the field is treated as if blanks had been removed, the remaining portion of the field right-justified, and the field padded with leading blanks. A field of all blanks has a value of zero.

If a BZ descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are interpreted as zeros.

For example, assuming BLANK = 'NULL', if the statement:

```
READ (6,'(I3, BZ, I3, BN, I3)')I,J,K
```

reads the input record:

```
1ΔΔ2ΔΔ3ΔΔ
```

then the I, J, and K have the following values:

```
I = 1   J = 200   K = 3
```

S,SP,SS Plus Sign Control

The S, SP and SS descriptors can be used on output with the I,F,E,D, and G descriptors to control the printing of plus (+) characters. S, SP and SS have no effect on input.

Normally, FORTRAN does not precede positive numbers by a plus sign on output. If an SP descriptor is encountered in a format specification, all succeeding positive numeric fields will contain the plus sign (w must be of sufficient length to include the sign). If an SS or S descriptor is encountered, the optional plus signs will not appear.

S, SP, and SS have no effect on plus signs preceding exponents, since those signs are always provided. For example:

```
A = 10.5
B = 7.3
C = 26.0
WRITE (2,'(1X,F6.2,SP,F6.2,SS,F6.2)')A,B,C
```

prints the following:

```
ΔΔ10.50Δ+7.30Δ26.00
```

A Descriptor

The A descriptor can be used with an input/output list item of type character or noncharacter. This descriptor has the forms:

```
A    Aw
```

Input

If w is less than the length of the list item, the input quantity is stored left-justified in the item; the remainder of the item is filled with blanks. If w is greater than the length of the item, the rightmost characters are stored and the remaining characters are ignored. If w is omitted, the length of the field is equal to the length of the list item. Examples of A input are shown in figure 5-19.

Output

If w is less than the length of the list item, the leftmost characters in the item are output. For example, if a variable A, declared CHARACTER A*8, contains:

```
SAMPLEΔΔ
```

and A is output with the following statement:

```
WRITE (6,'(1X,A4)')A
```

then the characters SAMP are output.

If w is greater than the length of the list item, the characters are output right-justified in the field, with blanks on the left. For example, if A in the previous example is output with the following statements:

```
      WRITE (6,400)A
400   FORMAT (1X,A12)
```

output is as follows:

```
ΔΔΔΔSAMPLEΔΔ
```

If w is omitted, the length of the character list item determines the length of the output field.

Example 1 (character list item):

```
      CHARACTER A*9
      READ (5,100) A
  100 FORMAT (A7)
```

Input record:

```
      EXAMPLE
```

In location A:

```
      EXAMPLEΔΔ
```

Example 2:

```
      CHARACTER B*10
      READ (5,200)B
  200 FORMAT (A13)
```

Input record:

```
1             13
SPECIFICATION
```

In location B:

```
1         10
CIFICATION
```

Example 3:

```
      CHARACTER Q*8,P*12,R*9
      READ (5,10) Q,P,R
   10 FORMAT (A8,A12,A5)
```

Input record:

```
THIS IS AN EXAMPLE I KNOW
   8          12         5
```

In storage:

```
P THISΔISΔ
Q ANΔEXAMPLEΔI
R ΔKNOWΔΔΔΔ
```

Example 4:

```
      CHARACTER NAME*30,PHONE*7
      READ (5,'(A,A)') NAME,PHONE
```

Note that if no length is specified for an A edit descriptor, the length of the list item is used.

Figure 5-19. A Input Examples

A Descriptor for Noncharacter List Items

The form of the A descriptor, when used for noncharacter list items, is:

Aw

When the A descriptor is used with an input/output list item of noncharacter type, character code conversion (appendix A) is performed. The field width specifier, w, must appear; w characters are converted.

On input, if w is less than or equal to 10 (there are 10 characters per word), the w characters of the input item are converted to character code and stored left-justified in the word with blank fill on the right. If w is greater than 10, the rightmost 10 characters of the input item are converted and stored.

On output, if w is less than or equal to 10, the leftmost w characters of the output item are written to the output record. If w is greater than 10, the output item is right-justified in the field and preceded by blanks.

R Descriptor

The R descriptor is used with noncharacter list items. This descriptor is used to transmit the rightmost characters of a word. The R descriptor has the form:

Rw

On both input and output, the R specification is identical to the A specification, unless w is less than 10.

On input, if w is less than 10, the rightmost w characters are read and stored right-justified with upper binary zero fill.

On output, if w is less than 10, the rightmost w characters of the output item are written to the output record.

An example of R input is shown in figure 5-20.

```
      BOOLEAN HOO,RAY
      READ (5,600) HOO,RAY
  600 FORMAT (R10,R5)
```

Input record:

```
                RESULTS OF TEST
                    10      5
```

In storage:

```
HOO: R E S U L T S Δ O F
RAY: 0 0 0 0 0 Δ T E S T
```

Figure 5-20. R Input Example

L Descriptor

The L descriptor is used to input or output logical items. This descriptor has the form:

Lw

Input

If the first nonblank characters in the field are T or .T, the logical value .TRUE. is stored in the corresponding list item, which should be of type logical. If the first nonblank characters are F or .F, the value .FALSE. is stored. If the first nonblank characters are not T, .T, F, or .F, a diagnostic is printed. An all blank field has the value .FALSE.

Output

Variables output under the L specification should be of type logical. A value of .TRUE. or .FALSE. in memory is output as a right-justified T or F with blanks on the left.

Example:

```
      LOGICAL I,J,K
      I = .TRUE.
      J = .FALSE.
      K = .TRUE.
      WRITE (4,5) I,J,K
    5 FORMAT (3L3)
```

Printed output:

```
ΔTΔΔFΔΔT
```

O Descriptor

The O descriptor is used to input or output items in octal format. This descriptor has the forms:

```
Ow  Ow.m
```

The form Ow.m means the same thing as Ow on input. The octal digits include the numbers 0 through 7.

Input

The input field can contain a maximum of 20 octal digits. Blanks are allowed and a plus or minus sign can precede the first octal digit. Blanks are interpreted as zeros and an all blank field is interpreted as zero. A decimal point is not allowed. An example is shown in figure 5-21.

```
      BOOLEAN P,Q,R
      READ 10,P,Q,R
   10 FORMAT (BZ,O10,O12,O2)

Input record:

   3737373737666Δ6644Δ444 -0
   10         12        2

In storage (octal representation):

   P: 0000000003737373737
   Q: 0000000666066440444
   R: 7777777777777777777
```

Figure 5-21. O Input Example

Output

If w is less than 20, the rightmost digits are output. For example, if location P contains:

```
0000000003737373737
```

and the output statements are:

```
      WRITE (6,100)P
  100 FORMAT (1X,O4)
```

the digits 3737 are output.

If w is greater than 20, the 20 total digits (20 octal digits = a 60- bit word) are right-justified with blanks on the left.

For example, if the contents of location P are output with the following statements:

```
      WRITE (6,200)P
  200 FORMAT (1X,O22)
```

output would appear as follows:

```
ΔΔ0000000003737373737
```

A negative number is output in one's complement internal form. For example:

```
      I = -11
      WRITE (6,200)I
  200 FORMAT (1X,O22)
```

Output:

```
ΔΔ7777777777777777764
```

If m is specified, the number is printed with leading zeros so that at least m digits are printed, and with a minus sign for negative numbers. If the number cannot be output in w octal digits, all asterisks will fill the field.

The specification Ow produces a string of up to 20 octal digits. Two octal specifications must be used for variables whose type is complex or double precision.

Z Descriptor

The Z descriptor is used for hexadecimal conversion. This descriptor has the forms:

```
Zw  Zw.m
```

The form Zw.m is meaningful for output only. Hexadecimal digits include the digits 0 through 9 and the letters A through F. A hexadecimal digit is represented by 4 bits.

Input

The input string can contain up to 15 hexadecimal digits. Embedded blanks are interpreted as zero and an all blank field is equivalent to zero. A plus or minus sign can precede the first digit. The string is stored right-justified with zeros on the left.

An example is illustrated in figure 5-22.

```
      INTEGER R,S
      READ (10,'(BZ,Z10,Z4)')R,S

Input record:

   A309FFFFCC4ΔD1
   10         4

In storage (hexadecimal representation):

   R: 00000A309FFFFCC
   S: 0000000000040D1
```

Figure 5-22. Z Input Example

Output

If w is less than 15, the rightmost w*4 bits are converted to hexadecimal and written. For example, if location I contains:

0000000000FB26C (hexadecimal format)

then the output statement:

```
WRITE(6,'(1X,Z3)')I
```

writes the digits 26C.

If w is greater than 15, the 15 hexadecimal digits are right-justified with blanks on the left.

If m is specified, the number is printed with leading zeros so that at least m digits are output. If the number of hexadecimal digits exceeds w, a field of asterisks is written.

H Descriptor

The H descriptor is used to output strings of characters. This descriptor is not associated with a variable in the output list. The H descriptor has the form:

nHstring

- n — Is the number of characters in the string including blanks.
- string — Is a string of characters.

The H descriptor cannot be used on input.

Note that although using apostrophes to designate a character string precludes the need to count characters, the H descriptor may be more convenient if the string contains apostrophes.

Example:

Source statements:

```
    A = 1.5
    WRITE (2,30)A
 30 FORMAT (6HΔLMAX=,F5.2)
```

Output:

```
LMAX = 1.50
```

Replacing the H descriptor in the preceding example with 'ΔLMAX=' would produce the same output.

Apostrophe and Quote Descriptors

Character strings delimited by a pair of apostrophe (') or quote (") symbols can be used as alternate forms of the H specification for output. The paired symbols delineate the string. If the string is empty or invalidly delimited, a fatal compilation error occurs and an error message is printed. The apostrophe and quote descriptors must not be used on input.

NOTE

The apostrophe descriptor is preferred usage over the quote descriptor.

Examples:

Source statements:

```
    PRINT 10
 10 FORMAT ('ΔSUBTOTALS')
```

Printed output:

```
 SUBTOTALS
```

Source statements:

```
    WRITE (6,20)
 20 FORMAT ('ΔRESULT OF CALCULATIONS ISΔ'
   *'AS FOLLOWS')
```

Output:

```
 RESULT OF CALCULATIONS IS AS FOLLOWS
```

An apostrophe or quote within a string delimited by the same symbol can be represented by two consecutive occurrences of the symbol. Alternatively, if a quote or apostrophe appears within a string, the other symbol can be used as the delimiter.

Examples:

Source statements:

```
    PRINT 1
  1 FORMAT ("ΔABC'DE")
```

Output:

```
 ABC'DE
```

Source statements:

```
    PRINT 3
  3 FORMAT('ΔDON''T')
```

Output:

```
 DON'T
```

Note that on some printers " is output as ≠ and ' is output as ↑.

X Descriptor

The X descriptor is used to skip character positions in an input line or output line. X is not associated with a variable in the input/output list. The X descriptor has the form:

nX

- n — Is the number of character positions to be skipped from the current character position; n is a nonzero unsigned integer.

The specification nX indicates that transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

Examples:

Source statements:

```
A = -342.743
B = 1.53190
J = 22
WRITE (6,'(1X,F9.4,4X,F7.5,4X,I3)') A,B,J
```

Output:

```
-342.7430ΔΔΔΔ1.53190ΔΔΔΔΔ22
```

Source statement:

```
READ (3,'(F5.2,3X,F5.2,6X,F5.2)') R,S,T
```

Input:

```
14.62ΔΔ$13.78ΔCOSTΔ15.97
```

In storage:

```
R   14.62
S   13.78
T   15.97
```

T,TL,TR Descriptors

The T, TL, and TR descriptors provide for tabulation control. These descriptors have the forms:

Tn TLn TRn

n Is a nonzero unsigned decimal integer.

When a Tn descriptor is encountered in a format specification, input or output control skips right or left to column n; the next edit descriptor is then processed.

When a TLn descriptor is encountered, control skips backward (left) n columns. If n is greater than or equal to the current character position, control skips to the first character position.

When a TRn descriptor is encountered, control skips forward (right) n characters.

On card input, control can be positioned beyond column 80, but a succeeding descriptor would read only blanks.

Example:

```
      READ 40, A,B,C
   40 FORMAT (T2,F5.2,TR5,F6.1,TR3,F5.2)
```

Input:

```
Δ684.73ΔΔΔΔΔ2436.2ΔΔΔΔ89.14
```

A is set to 684.7, B to 2436.0, and C to 89.0.

Example:

```
      WRITE (31,10)
   10 FORMAT (T20,'LABELS')
```

Positions to column 20 of the output record and writes the characters LABELS.

With a T, TR, or TL specification, the order of a list need not be the same as that of the input or output record, and the same information can be read more than once.

Example:

```
READ (2,'(F5.2,TL5,F5.2)')A,B
```

Input record:

```
76.05
```

Both A and B contain 76.05.

When a T, TR, TL specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; those already filled are left unchanged. An example is shown in figure 5-23.

The following example shows that it is possible to destroy a previously formed field:

```
      WRITE (2,8)
    8 FORMAT ('DISASTERS',T5,3H123)
```

Output record before printing:

```
DISA123RS
```

If the output record is printed, the first character is not printed. See Carriage Control Character in this section.

End-of-Record Slash

The slash indicates the end of a record anywhere in the FORMAT specification. When a slash is used to separate edit descriptors, a comma is allowed but not required. Consecutive slashes can be used and need not be separated from other elements by commas. When a slash is the last format specification to be processed, it causes a blank record to be written on output or an input record to be skipped. Normally, the slash indicates the end of a record during output and specifies that further data comes from the next record during input.

Example:

```
      WRITE (2,10)
   10 FORMAT (6X,7HHEADING///1X,5HINPUT,
     *7HΔOUTPUT)
```

Printed output:

```
ΔΔΔΔΔHEADING
(blank line)
(blank line)
INPUT OUTPUT
```

Each line corresponds to a formatted record. The second and third records are blank and produce the line spacing illustrated.

```
      PROGRAM TEST
1     FORMAT(12(' 123456789'))
      PRINT 1
      PRINT 60
60    FORMAT(T80,'COMMENTS',T60,'HEADING4',T40,
     *       'HEADING3',T20,'HEADING2',T2,'HEADING1')
      PRINT 10
10    FORMAT(20X, 'THIS IS THE END OF THIS RUN',T52,'HONEST')
      PRINT 1
      STOP
      END
```

```
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
HEADING1          HEADING2            HEADING3            HEADING4            COMMENTS
                    THIS IS THE END OF THIS RUN    HONEST
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

For the FORMAT statement labeled 60, control passes over the first print position (the one used for carriage control); therefore, it is automatically set to a blank, which causes the line to be single spaced.

Figure 5-23. T Output Example

Example:

```
      I=5
      J=6
      K=7
      WRITE (2,1)I,J,K
    1 FORMAT (3I5/F10.4)
      WRITE (2,2)
    2 FORMAT ('ΔA BLANK LINE SHOULD PRECEDEΔ',
     *THIS LINE')
```

Printed output:

```
ΔΔΔ5ΔΔΔΔ6ΔΔΔΔ7

A BLANK LINE SHOULD PRECEDE THIS LINE
```

The variable list (I, J, K) is exhausted and processing continues until a variable conversion is encountered (F10.4). Since the slash has been processed, it causes a blank line to be printed, and F10.4 is ignored because there is nothing to be converted.

Example:

```
      DIMENSION B(3)
      READ (5,100)IA,B
  100 FORMAT (I5/3E7.2)
```

These statements read two records; the first contains an integer number, and the second contains three real numbers.

Example:

```
      WRITE (3,11)A,B,C,D
   11 FORMAT (1X,2E10.2/1X,2F7.3)
```

In storage:

```
A   -11.6
B   .325
C   46.327
D   -14.261
```

Printed output:

```
ΔΔ-.12E+02ΔΔΔ.33E+00
Δ46.327-14.261
```

Example:

```
      WRITE (1,11)A,B,C,D
   11 FORMAT (1X,2E10.2//1X,2F7.3)
```

Printed output:

```
ΔΔ-.12E+02ΔΔΔ.33E+00
(blank line)
Δ46.327-14.261
```

The second slash causes the blank line.

Repeated Edit Descriptors

Certain edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. The repeatable edit descriptors are D, E, F, G, I, A, L, R, Z, and O. The other edit descriptors cannot be repeated.

Examples:

```
  100 FORMAT (3I4,2E7.3)
```

is equivalent to:

```
  100 FORMAT (I4,I4,I4,E7.3,E7.3)

   50 FORMAT (4G12.6)
```

is equivalent to:

```
   50 FORMAT (G12.6,G12.6,G12.6,G12.6)
```

A group of descriptors can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor. If no integer precedes the left parenthesis, the repetition factor is 1.

Example:

```
    1 FORMAT (I3,2(E15.3,F6.1,2I4))
```

is equivalent to the following specification if the number of items in the input/output list does not exceed the number of format conversion codes:

```
    1 FORMAT (I3,E15.3,F6.1,I4,I4,E15.3,F6.1,I4,I4)
```

A maximum of nine levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If there are fewer items in the input/output list than indicated by the format conversions in the FORMAT specification, the excess conversions are ignored.

If the number of items in the input/output list exceeds the number of format conversions when the final right parenthesis in the FORMAT statement is reached, the line formed internally is output. The format control then scans to the left looking for a right parenthesis within the FORMAT statement. If none is found, the scan stops when it reaches the beginning of the format specification. If a right parenthesis is found, however, the scan continues to the left until it reaches the field separator which precedes the left parenthesis pairing the right parenthesis. Output resumes with the format control moving right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered.

If n slashes are indicated, a repetition factor can be used to indicate multiple slashes; n-1 lines are skipped on output.

Example:

```
      DIMENSION A(9)
      DATA A/3.62,-4.03,-9.78,-6.33,7.12,3.49,6.21,
     *-6.74,-1.18/
          .
          .
          .
      WRITE (3,15)(A(I),I=1,9)
   15 FORMAT (8HΔRESULTS,4(/),(1X,3F8.2))
```

Format statement 15 is equivalent to:

```
15 FORMAT (8HΔRESULTS,//// (1X,3F8.2))
```

Output:

```
RESULTS
(blank line)
(blank line)
(blank line)
ΔΔΔΔ3.62ΔΔΔ-4.03ΔΔΔ-9.78
ΔΔΔ-6.33ΔΔΔΔ7.12ΔΔΔΔ3.49
ΔΔΔΔ6.21ΔΔΔ-6.74ΔΔΔ-1.18
```

Example:

```
      READ (5,300) I,J,E,K,F,L,M,G,N,R
300 FORMAT (I3,2(I4,F7.3),I7)
```

Data is stored in I with format I3, J with I4, E with F7.3, K with I4, F with F7.3, and L with I7. A new record is then read; data is stored in M with the format I4, G with F7.3, N with I4, and R with F7.3.

Example:

```
      READ (5,100) NEXT,DAY,KAT,WAY,NAT,
     *RAY,MAT
100 FORMAT (I7,(F12.7,I3))
```

NEXT is input with format I7, DAY is input with F12.7, KAT is input with I3. The FORMAT statement is exhausted (the right parenthesis has been reached), a new record is read, and the statement is rescanned from the group (F12.7,I3). WAY is input with the format F12.7, NAT with I3, and from a third record, RAY with F12.7, and MAT with I3.

Termination of Format Control

A colon (:) in a format specification terminates format control if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list. This descriptor is useful in forms where nonlist item edit descriptors follow list item edit descriptors; when the iolist is exhausted, the subsequent edit descriptors are not processed. For example:

```
    A = 1.0
    B = 2.2
    C = 3.1
    D = 5.7
    PRINT 10, A, B, C, D
 10 FORMAT (4(F4.1,:,','))
```

Output:

```
1.0,Δ2.2,Δ3.1,Δ5.7
```

In this example, format control terminates after the value of D is printed, and the last comma is not printed.

Carriage Control Character

The first character of a printer output record is used for carriage control and is not printed. It appears in other forms of output as data. Carriage control also applies to records listed at a terminal under INTERCOM; the meaning of carriage control characters depends on the type of terminal. (See the INTERCOM reference manual.) Carriage control does not apply to records listed at a terminal under the NOS Time-Sharing System; the first character is listed as data.

The carriage control characters are shown in table 5-3.

TABLE 5-3. PRINTER CONTROL CHARACTERS

Character	Action
Blank	Space vertically one line, then print.
0	Space vertically two lines, then print.
1	Eject to the first line of the next page before printing.
+	No advance before printing; allows overprinting.
Any other character	Refer to the operating system reference manual.

(This table applies to NOS/BE and SCOPE 2 only. For corresponding information under NOS, refer to the reference manual for the subsystem under which the program is executed.)

For output directed to the card punch or any device other than the line printer or terminal, control characters are not required. If carriage control characters are transmitted to the card punch, they are punched in column one.

Carriage control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash. Carriage control characters can be generated by any means.

Examples:

```
10 FORMAT (1H0,F7.3,I2,G12.6)

20 FORMAT (' ',I5,'RESULT=',F8.4)

30 FORMAT ('1',I4,2(F7.3))

40 FORMAT (1X,I4,G16.8)
```

A program using carriage control characters, and resulting output, is shown in figure 5-24. The program constructs a tic tac toe diagram. A '1' specification causes the first output line to appear at the top of a page. FORMAT statement 20 causes three lines to be skipped. In FORMAT statements 30 and 40, a slash skips to the next output record and a plus character causes the record to begin on the same line as the previous record, resulting in overprinting of a row of X characters and = characters. FORMAT statement 60 uses a '0' specification to skip two lines before writing the last output line.

EXECUTION TIME FORMAT SPECIFICATION

Variable format specifications can be read in as part of the data at execution time and used wherever a normal format can be used. The format can be read in under the A specification and stored in a character array, variable,

Example:

```
      PROGRAM CHARCON
      PRINT 10
10    FORMAT('1', 5X, 'HERE WE ARE AT THE TOP OF A NEW PAGE')
      PRINT 20
20    FORMAT(3(/))
C
      DO 50 I=2, 8
      IF (I .EQ. 4 .OR. I .EQ. 6) THEN
      PRINT 30
30    FORMAT(20X, ' XXXXXXXXXX '/'+', 19X, ' ========== ')
      ELSE
      PRINT 40
40    FORMAT(21X, '    X  X  ',/'+',20X, '    =  =  ')
      ENDIF
50    CONTINUE
C
      PRINT 60
60    FORMAT('0', 5X, 'BEGIN TIC TAC TOE ')
      STOP
      END
```

Output:

```
HERE WE ARE AT THE TOP OF A NEW PAGE


          *  *
          *  *
       **********
          *  *
       **********
          *  *
          *  *

BEGIN TIC TAC TOE
```

Figure 5-24. Carriage Control Example

or array element; or it can be included in a DATA statement. Formats can also be generated by the program at execution time.

If an array or array element is used, its type can be other than character, although character is the preferred type. In either case, the format must consist of a list of descriptors and editing characters enclosed in parentheses, but without the keyword FORMAT and the statement label.

The name of the entity containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The name specifies the location of the first word of the format information.

Example:

Input record:

```
(E7.2,G20.5,F7.4,I3)
```

This specification can be read and subsequently referenced as follows:

```
CHARACTER F*30
READ (2,'(A)') F
WRITE (3,F) A,B,C,N
```

Example:

Input record:

```
(E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

This specification can be read by the statements:

```
CHARACTER VAR*40
READ (2,'(A)') VAR
```

A subsequent output statement in the same program can refer to these format specifications as:

```
WRITE (2,VAR) A,B,I,C,D,E,J
```

If PRTFLG is zero, the program produces the same result as WRITE (2,'(3I10)') I,J,K.

UNFORMATTED INPUT/OUTPUT

Unformatted READ and WRITE statements do not use format specifications and do not convert data in any way on input or output. Instead, data is transferred as is between memory and the external device. Each unformatted input/output statement transfers exactly one record. If data is written by an unformatted WRITE and subsequently read by an unformatted READ, exactly what was written is read; no precision is lost since no conversion is performed.

UNFORMATTED WRITE

The unformatted WRITE statement is shown in figure 5-25.

```
WRITE ([UNIT=]u[,IOSTAT=ios] [,ERR=sl])[iolist]
```

Figure 5-25. Unformatted WRITE Statement

This statement is used to output binary records. Information is transferred from the items iolist to the specified output unit u with no format conversion. One record is created by an unformatted WRITE statement. If the list is omitted, the statement writes a null record on the output device. A null record has no data but contains all other properties of a legitimate record.

Example:

```
PROGRAM OUT
DIMENSION A(260), B(4000)
.
.
.
WRITE (10,ERR=16) A,B
END
```

The 4260 words of arrays A and B are written as one record on unit 10.

UNFORMATTED READ

The unformatted READ statement is shown in figure 5-26.

```
READ ([UNIT=]u[,IOSTAT=ios] [,ERR=sl] [,END=sl])
     [iolist]
```

Figure 5-26. Unformatted READ Statement

One record is transmitted from the specified unit u to the storage locations named in iolist. Records are not converted; no FORMAT statement is used. The information is transmitted from the designated file in the form in which it exists on the file without any conversion. If the number of words in the list exceeds the number of words in the record, an execution diagnostic results. If the number of locations specified in iolist is less than the number of words in the record, the excess data is ignored. If iolist is omitted, the unformatted READ skips one record.

The user should specify the END= or IOSTAT= parameter to avoid termination when an end-of-file is encountered. If an attempt is made to read on unit u and an end-of-file was encountered on the previous read operation on this unit, execution terminates and an error message is printed. Records following an end-of-file can be read by issuing a CLOSE followed by an OPEN on the file or by using the EOF function (section 7). CLOSE/OPEN, described later in this section, is the preferred method.

Example:

```
PROGRAM AREAD
READ (2,END=30,ERR=40) X,Y,Z
SUM = X+Y+Z/2.
.
.
.
END
```

LIST DIRECTED INPUT/OUTPUT

List directed input/output involves the processing of coded records without a FORMAT statement. Each record consists of a list of values in a freer format than is used for formatted input/output. This type of input/output is particularly convenient when the exact form of data is not important.

LIST DIRECTED INPUT

The list directed READ statement is shown in figure 5-27.

```
READ ([UNIT=]u,[FMT=]*[,IOSTAT=ios] [,ERR=sl]
     [,END=sl])[iolist]

READ *[,iolist]
```

Figure 5-27. List Directed READ Statement

Data is transmitted from unit u or the file INPUT (if u is omitted or unit=* specified) to the storage locations named in iolist. The input data items are free-form with separators rather than in fixed-size fields.

A list directed READ following a list directed READ that terminated in the middle of a record starts with the next data record.

The user should specify the END= or IOSTAT= parameter to avoid termination when an end-of-file is encountered. If an attempt is made to read on unit u and an end-of-file was encountered on the previous read operation on this unit, execution terminates and an error message is printed. Records following an end-of-file can be read by issuing a CLOSE followed by an OPEN on the file or by using the EOF function (section 7). CLOSE/OPEN, described later in this section, is the preferred method.

Input data consists of a string of values separated by one or more blanks, or by a comma or slash, either of which can be preceded or followed by any number of blanks. Also, a line boundary, such as end-of-record or end-of-card, serves as a value separator; however, a separator adjacent to a line boundary does not indicate a null value.

Embedded blanks are not allowed in input values, except character values and complex numbers. The format of values in the input record is as follows:

Integers	Same format as for integer constants.

Real numbers	Any valid FORTRAN format for real or double precision numbers. In addition, the decimal point can be omitted; it is assumed to be to the right of the mantissa.
Complex numbers	Two real values, separated by a comma, and enclosed by parentheses. The parentheses are not considered to be a separator. The decimal point can be omitted from either of the real constants. Each of the real values can be preceded or followed by blanks.
Character values	A string of characters (which can include blanks) enclosed by apostrophes. A delimiting apostrophe can be represented within a string by two successive occurrences. Character values can only be read into character arrays, variables and substrings. If the string length exceeds the length of the list item, the string is truncated. If the string is shorter than the list item, the string is left-justified and remaining character positions are blank filled.
Logical values	An optional period, followed by a T or F, followed by optional characters which do not include separators (slashes or commas).

A Boolean constant can be input only if the corresponding list item is of type Boolean. These include:

Octal constants.

Hexadecimal constants.

Hollerith constants containing one through 10 characters and delimited by quotes. Constants of less than 10 characters are left-justified with blank fill on the right. Strings of greater than 10 characters are truncated to 10 characters.

In addition, real and integer values can be read into Boolean variables.

To repeat a value, an integer repeat constant is followed by an asterisk and the constant to be repeated. Blanks cannot be embedded in the repeat part of the specification.

A null can be input in place of a constant when the value of the corresponding list entity is not to be changed. A null is indicated by the first character in the input string being a comma or by two commas separated by an arbitrary number of blanks. Nulls can be repeated by specifying an integer repeat count followed by an asterisk and any value separator. The next value begins immediately after a repeated null. A null cannot be used for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, remaining list elements are treated as nulls and the remainder of the current record is discarded.

Input values must correspond in type to variables in the input/output list. Note that the form of a real value can be the same as that of an integer value.

Some examples of list directed input are illustrated in figure 5-28.

LIST DIRECTED OUTPUT

The list directed output statements consist of a WRITE, a PRINT, and a PUNCH statement. These statements are shown in figures 5-29, 5-30, and 5-31, respectively.

Data is transferred from storage locations specified by the iolist to the designated unit in a manner consistent with list directed input.

PRINT outputs data to the unit OUTPUT. PUNCH outputs to the unit PUNCH.

List directed output is consistent with the input; however, null values, slashes, repeated constants, and the apostrophes used to indicate character values are not produced. For real or double precision variables with absolute values in the range of 10^{-6} to 10^{9}, an F format type of conversion is used; otherwise, output is of the 1PE type. Trailing zeros in the mantissa and leading zeros in the exponent are suppressed. Values are separated by blanks.

List directed output statements always produce a blank for carriage control as the first character of the output record.

Logical values are output as T or F. Complex values are enclosed in parentheses with a comma separating the real and imaginary parts.

Boolean values are output in the form O"n n...", where n is an octal digit. Leading zeros are suppressed.

On a connected file under NOS, if the iolist of a list directed output statement ends with a comma, no carriage control or line feed takes place after the line is output. Under NOS/BE and SCOPE 2, a comma as the last character of an iolist is ignored.

Some examples of list directed output are shown in figure 5-32.

NAMELIST INPUT/OUTPUT

The NAMELIST statement permits input and output of groups of variables and arrays with an identifying name. No format specification is used. The NAMELIST statement is a nonexecutable statement that appears in the declarative portion of the program following any PARAMETER statements. The NAMELIST statement is shown in figure 5-33.

The NAMELIST group name identifies the succeeding list of variables or array names.

A NAMELIST group name must be declared in a NAMELIST statement before it is used in an input/output statement. The group name can be declared only once, and it can not be used for any purpose other than a NAMELIST name in the program unit. It can appear in READ, WRITE, PRINT, and PUNCH statements in place of the format specifier. When a NAMELIST group name is used, the list must be omitted from the input/output statement.

Example 1:

```
            PROGRAM LDR
            READ *, CAT, BIRD, DOG
            PRINT 10, CAT, BIRD, DOG
      10    FORMAT(' CAT = ', E9.3,/' BIRD = ', E9.3,/' DOG = ',E9.3)
            STOP
            END
```

Input:

```
            13.3, -5.2, .01
```

Output:

```
CAT =  .133E+02
BIRD = -.520E+01
DOG =  .100E-01
```

Example 2:

```
            PROGRAM LDIN
      100   READ(*,*,END=99) J,K
            PRINT 1, J,K
      1     FORMAT (T20,2I5)
            GO TO 100
      99    STOP
            END
```

Input:

```
            1 2
            ,3
            ,,
            2*8
            /9
```

Output:

```
1    2
1    3
1    3
8    8
8    8
```

Figure 5-28. List Directed Input Examples

WRITE ([UNIT=]u,[FMT=]*[,IOSTAT=ios] [,ERR=sl])[iolist]

Figure 5-29. List Directed WRITE Statement

PRINT*[,iolist]

Figure 5-30. List Directed PRINT Statement

PUNCH*[,iolist]

Figure 5-31. List Directed PUNCH Statement

A variable or array name can belong to one or more NAMELIST groups.

Data read by a single NAMELIST name READ statement must contain only names listed in the referenced NAMELIST group. All items in the NAMELIST group, or any subset of the group, can be input. Values are unchanged for items not input. Variables need not be in the order in which they appear in the defining NAMELIST statement.

A sample program segment showing NAMELIST input and output is illustrated in figure 5-34.

NAMELIST INPUT

The NAMELIST READ statement is shown in figure 5-35.

When a READ statement references a NAMELIST group name, input data in the format described below is read from the designated file. If the specified group name is not found before end-of-file, a fatal error occurs. If the file is empty an end-of-file condition results. This must be detected by an END= or IOSTAT= specifier or a fatal error will result. A subsequent read on the same file without an intervening positioning, CLOSE/OPEN, or EOF function test, results in a fatal error. The format of a NAMELIST input group is shown in figure 5-36.

In each record of a NAMELIST group, column one is reserved for carriage control and must be left blank. Data items following $name (or &name) are read until another $ (or &) is encountered.

Blanks must not appear:

- Between $ (or &) and NAMELIST group name
- Between $ (or &) and END
- Within array names and variable names

Blanks can be used freely elsewhere.

More than one record can be used as input data in a NAMELIST group. The first column of each input record is ignored. All input records containing data should end with a constant followed by a comma; however, the last record can be terminated by a $ (or &) without the final comma. Constants can be preceded by a repetition factor followed by an asterisk. Omitting a constant constitutes a fatal error.

Example 1:

```
PROGRAM LDW
INTEGER J(4)
COMPLEX Z(2)
DOUBLE PRECISION Q
DATA J,Z,Q /1,-2,3,-4,(7.,-1.),(-3.,2.),1.D-5/
PRINT *, J
PRINT *, Z,Q
STOP
END
```

Output:

```
1 -2 3 -4
(7.,-1.) (-3.,2.) .00001
```

Example 2:

```
PROGRAM K
PRINT *, 'TYPE IN X'
READ *, X
PRINT *, 'TYPE IN Y'
READ *, Y
END
```

Terminal listing under NOS:

```
 TYPE IN X
? 1.234
 TYPE IN Y
? 5.678
```

Figure 5-32. List Directed Output Examples

NAMELIST/name/a[,a]. . .[/name/a[,a]. . .]. . .

name	Symbolic group name which must be enclosed in slashes and must be unique within the program unit.
a	Variable or array name.

Figure 5-33. NAMELIST Statement

Constants can be integer, real, double precision, complex, logical, Boolean, or character. Each constant must agree with the type of the corresponding input list item as follows:

A logical, character, or complex constant must be of the same type as the corresponding input list item. 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 element, or substring.

An integer, real, or double precision constant can be used for an integer, real, double precision, or Boolean input list item. The constant is converted to the type of the list item. A Boolean constant cannot be used for a non-Boolean list item.

Logical constants have the following forms:

.TRUE.	.FALSE.
.T.	.F.
T	F

A character constant must have delimiting apostrophes. If a character constant occupies more than one record, each continuation of the constant must begin in column two; a complex constant has the form (real constant, real constant). A character constant must extend to the end of a record preceding a continuation record. A Boolean constant must be an octal constant, a hexadecimal constant, or a Hollerith constant delimited by quotes.

Blank characters appearing within noncharacter constants are ignored. The BLANK= specifier in an OPEN statement has no effect on NAMELIST. If a constant, other than a character constant, contains no characters other than blanks, a fatal error results.

Example:

```
ΔΔ$AGRP                    Group name
ΔXVAL=5.0,
ΔARR=5*(1.7,-2.4),         Five complex numbers
ΔCHAR='HI THERE',
ΔΔ$END                     Group terminator
```

NAMELIST OUTPUT

The NAMELIST output statements consist of a WRITE statement, a PRINT statement, and a PUNCH statement. These statements are shown in figures 5-37, 5-38, and 5-39.

Example:

```
      PROGRAM NMLIST
      NAMELIST /SHIP/ A,B,C,I1,I2
      READ(*, SHIP,END=10)
      IF(C .GT. 0.0) THEN
      A=B+C
      I1=I1+I2
      WRITE(*, SHIP)
      ENDIF
      STOP
10    PRINT *, ' NO DATA FOUND'
      STOP
      END
```

Input record:

```
$SHIP A=15.7,B=12.3,C=3.4,I1=58,I2=8$
```

(beginning in Column 2)

Output:

```
$SHIP

A       = .157E+02,

B       = .123E+02,

C       = .34E+01,

I1      = 58,

I2      = 8,

$END
```

Figure 5-34. NAMELIST Example

READ ([UNIT=]u,[FMT=]name[,IOSTAT=ios] [,ERR=sl] [,END=sl])

READ name

Figure 5-35. NAMELIST READ Statement

All variables and arrays and their values in the list associated with the NAMELIST group name are output on the file associated with unit u, OUTPUT, or PUNCH. They are output in the order of specification in the NAMELIST statement. Output consists of at least three records. The first record is a $ in column 2 followed by the group name; the last record is a $ in column 2 followed by the characters END. Each group begins with triple spacing (a - is inserted in the carriage control position of each record).

Example:

```
PROGRAM NAME (OUTPUT, TAPE6=OUTPUT)
NAMELIST /VALUES/ TOTAL,QUANT,COST
DATA QUANT,COST /15.,3.02/
TOTAL = QUANT*COST*1.3
WRITE (6,VALUES)
STOP
END
```

Output:

```
$VALUES

TOTAL  =  .58889999999999E+02,

QUANT  =  .15E+02,

COST   =  .302E+01,

$END
```

$\left\{\begin{matrix}\$\\ \&\end{matrix}\right\}$ name $\left(\begin{matrix}v=c\\ v(i1:i2)=c\\ array[(s)]=[r*]c[,[r*]c]\ \ldots\\ array\ (s)(i1:i2)=[r*]c[,[r*]c]\end{matrix}\right)$ $\left[,\left(\begin{matrix}v=c\\ v(i1:i2)=c\\ array[(s)]=[r*]c[,[r*]c]\ \ldots\\ array\ (s)(i1:i2)=[r*]c[,[r*]c]\end{matrix}\right)\right]$ ... [,] $\left\{\begin{matrix}\$\\ \&\end{matrix}\right\}$ [END]

name	Is the name of the namelist group.
v	Is a variable name.
c	Is a constant.
array	Is an array name.
s	Is an array subscript in which each subscript expression is an integer constant. The number of subscript expressions in s must be equal to the number of dimensions of the array.
r	Is an unsigned, nonzero, integer repetition factor.
i1, i2	Are integer constants.

The form r*c is equivalent to r successive appearances of the constant c.

Figure 5-36. NAMELIST Group Format

```
WRITE ([UNIT=]u,[FMT=]name[,IOSTAT=ios] [,ERR=sl])
```

name Is a NAMELIST group name.

Figure 5-37. NAMELIST WRITE Statement

```
PRINT name
```

name Is the name of a NAMELIST group.

Figure 5-38. NAMELIST PRINT Statement

```
PUNCH name
```

name Is the name of a NAMELIST group.

Figure 5-39. NAMELIST PUNCH Statement

No data appears in column 1 of any record. If a noncharacter constant would cross column 80, the columns up to and including 80 are filled with blanks instead and the constant begins in column 82; therefore, card boundaries will not be crossed if data is punched. The maximum length of any record is 136 characters (unless a smaller maximum record length has been specified in the PROGRAM or OPEN statement). Logical constants appear as T or F. Elements of an array are output in the order in which they are stored.

Character constants are written with delimiting apostrophes. Boolean constants are written in the form O"n[n]...", where n is an octal digit; leading zeros are suppressed.

If a character constant crosses a record boundary and the file is punched, a record length of 80 must be specified to correctly read the cards with NAMELIST input.

Records output by a NAMELIST WRITE statement can be read later in the same program by a NAMELIST READ statement specifying the same group name.

Example:

```
NAMELIST /ITEMS/ X,Y,Z
.
.
.
WRITE (6, ITEMS)
```

Output record:

```
$ITEMS
X     =    .7342E+03,
Y     =    .23749E+04,
Z     =    .2225E+02,
$END
```

Subsequent READ statement:

```
READ (5,ITEMS)
```

ARRAYS IN NAMELIST

In input data the number of constants, including repetitions, given for an array name should not exceed the number of elements in the array.

Example:

```
INTEGER BAT(10)
NAMELIST /HAT/ BAT,DOT
READ (5,HAT)
```

Input record:

```
Δ$HAT     BAT=2,3,8*4,DOT=1.05$END
```

The value of DOT becomes 1.05; the array BAT is as follows:

```
BAT(1)    2
BAT(2)    3
BAT(3)    4
BAT(4)    4
BAT(5)    4

BAT(6)    4
BAT(7)    4
BAT(8)    4
BAT(9)    4
BAT(10)   4
```

Example:

```
DIMENSION GAY(5)
NAMELIST /DAY/ GAY,BAY,RAY
READ (5,DAY)
```

Input record:

```
Δ$DAY GAY(3)=7.2,GAY(5)=3.0,BAY=2.3,RAY=77.2$
```

```
array element=constant,...,constant
```

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants need not equal, but must not exceed, the remaining number of elements in the array.

Example:

```
DIMENSION ALPHA (6)
NAMELIST /BETA/ ALPHA,DELTA,X,Y
READ (5,BETA)
```

Input record:

```
Δ$BETA ALPHA(3)=7.,8.,9.,DELTA=2.$
```

In memory:

```
ALPHA(3)    7.
ALPHA(4)    8.
ALPHA(5)    9.
DELTA       2.
```

Example:

```
DIMENSION Y(3,5)
LOGICAL L
COMPLEX Z
NAMELIST /HURRY/ I1,I2,I3,K,M,Y,Z,L
READ(5, HURRY)
```

Input record:

```
Δ$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,
    12.0E1,13,4*14,Z=(1.,2.),K=16,M=17$
```

Values stored:

I1=1	Y(1,2)=14.0
I2=2	Y(2,2)=14.0
I3=3	Y(3,2)=14.0
Y(3,5)=26.0	Y(1,3)=14.0
Y(1,1)=11.0	K=16
Y(2,1)=120.0	M=17
Y(3,1)=13.0	Z=(1.,2.)
	L=.TRUE.

The rest of Y is unchanged.

BUFFER INPUT/OUTPUT STATEMENTS

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.

Buffer input/output statements (BUFFER IN and BUFFER OUT) allow input/output operations to occur simultaneously with other processing. They differ from formatted and unformatted READ and WRITE statements in the following ways:

A buffer statement initiates data transmission and then returns control to the program so that it can perform other tasks while data transmission is in progress. A READ or WRITE completes data transmission before returning control to the program.

In a buffer statement, parity must be specified by a parity indicator. In a READ or WRITE statement, parity is implied by the form of the statement: an unformatted READ or WRITE implies binary mode, and a formatted READ or WRITE implies coded mode.

READ and WRITE are associated with an input/output list. Buffer statements are not associated with a list; data is transmitted to or from a block of storage.

ENDFILE, REWIND, and BACKSPACE are valid for files processed by buffer statements. However, a file processed by buffer statements cannot be processed in the same program by formatted or unformatted input/output statements, or by mass storage or CYBER Record Manager subroutines unless the file has been rewound before changing the type of input/output used.

Each buffer statement defines the location of the first and last words of the block of memory to or from which data is to be transmitted. The address of the last word must be greater than or equal to the address of the first word. The relative locations of the first and last word are defined only if they are the same variable or are in the same array, common block, or equivalence class. If the first and last words do not satisfy one of these relationships, their relative position is undefined and a fatal error might result at execution time.

If the first word and the last word are in the same common block but not in the same array or equivalence class, optimization might be degraded.

After execution of a buffer statement has been initiated, and before referencing the same file or any of the contents of the block of memory to or from which data is transferred, the status of the buffer operation must be checked by a reference to the UNIT function (section 7). This status check ensures that the data has actually been transferred and the buffer parameters for the file have been restored. If a second buffer operation is attempted on the same file without an intervening reference to UNIT, the results are undefined.

On a CYBER 170 Model 176, a FILE control statement (appendix F) specifying SBF=NO must be provided if a level 2 or 3 extended memory variable is used in a buffer statement.

BUFFER IN

The BUFFER IN statement is shown in figure 5-40.

BUFFER IN (u,p) (a,b)

u	Is a unit specifier.
p	Is an integer constant or simple integer variable. Designates parity on 7-track magnetic tape; 0 designates even parity (coded); 1 designates odd parity (binary). This parameter is irrelevant for mass storage and 9-track SI tapes. For 9-track S and L tapes, 0 indicates conversion from display code to ASCII or EBCDIC (depending on the REQUEST control statement specification), while 1 indicates no conversion. The parameter does not affect parity on 9-track tapes. Under SCOPE 2, p does not apply since files are always read and written in odd parity; however, p must be specified.
a	Is the first variable or array element of the block of memory to which data is to be transmitted; cannot be type character.
b	Is the last variable or array element of the block of memory to which data is to be transmitted; cannot be type character. If u is a unit specifier of a tape or mass storage device, the block of memory to which data is to be transmitted should be one word larger than logically required. The additional word is needed to receive an error status from the operating system if a tape error occurs. Under SCOPE 2, the additional word is not needed because no error status word is written.

Figure 5-40. BUFFER IN Statement

BUFFER IN transfers one record from the file indicated by the unit designator u to the block of memory beginning at a and ending at b. If the record is shorter than the block of memory, excess locations are not changed. If the record is longer than the block of memory, excess words in the record are ignored, except when the record type is fixed (RT=F on FILE statement), in which case an error occurs.

The UNIT function can be used to test for end-of-file after BUFFER IN. After UNIT has been referenced, the number of words transferred to memory can be obtained by a call to the function LENGTH (section 7). If records do not terminate on a word boundary (in a file not written by BUFFER OUT), the exact length of the record is returned by LENGTHX in terms of words and excess bits.

If the end of a system-logical-record (end-of-section) is encountered on a file other than INPUT, no data is transferred and the length returned by LENGTH is zero. The next BUFFER IN begins reading after the end-of-section. (On INPUT, end-of-section is treated as end-of-file).

The UNIT function should be used to test for end-of-file after BUFFER IN.

Example:

```
DIMENSION CALC(51)
BUFFER IN (1,0) (CALC(1),CALC(51))
IF(UNIT(1).GE.0) GO TO 20
```

Coded information is transferred from logical unit 1 into storage beginning at the first word of the array, CALC(1), and extending through CALC(50). An error or end-of-file will transfer control to statement 20.

Figure 5-41 illustrates a program that uses a BUFFER IN statement. In this example, binary information is transferred from logical unit 1 into storage beginning at the first word of the array, REC(1), and extending through REC(512). The UNIT function tests the status of the buffer operation. If the buffer operation is completed without error, statement 3 is executed. If an end-of-file or a parity error is encountered, control transfers to statement 5 and the program stops.

BUFFER OUT

The BUFFER OUT statement is shown in figure 5-42. BUFFER OUT writes one record by transferring the contents of the block of memory beginning at a and ending at b to the file indicated by the unit designator u at the parity (even or odd) indicated by p. The length of the record:

```
LWA - FWA + 1
```

where LWA is the terminal address of the record and FWA is the starting address. For fixed-length records (RT=F on FILE statement), the record length is the length (characters) specified on the FILE statement (FL parameter). If FL is greater than (LWA - FWA + 1) X 10, an error occurs.

The UNIT function must be referenced before another reference is made to the file or to the contents of the block of memory.

BUFFER OUT (u,p) (a,b)

u	Is a unit specifier.
p	Is an integer constant or simple integer variable. Designates parity on 7-track magnetic tape; 0 designates even parity (coded); 1 designates odd parity (binary). This parameter is irrelevant for mass storage and 9-track SI tapes. For 9-track S and L tapes, zero indicates conversion from display code to ASCII or EBCDIC (depending on the REQUEST control statement specification), while 1 indicates no conversion. The parameter does not affect parity on 9-track tapes. Under SCOPE 2, p does not apply since files are always read and written in odd parity; however, p must be specified.
a	Is the first variable or array element of the block of memory from which data is to be transmitted; cannot be type character.
b	Is the last variable or array element of the block of memory from which data is to be transmitted; cannot be type character. If u is a unit specifier of a tape or mass storage device, the block of memory from which data is to be transmitted should be one word larger than logically required. The additional word is needed to receive an error status from the operating system if a tape error occurs. Under SCOPE 2, the additional word is not needed because no error status word is written.

Figure 5-42. BUFFER OUT Statement

DIRECT ACCESS FILES

Direct access file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written, and can normally be read back only in the same order. This can be slow and inconvenient in applications where the order of writing and of retrieving records differs and, in addition, it requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, a direct access file capability is provided by the FORTRAN input/output statements.

In a direct access file, any record can be read, written, or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random access mass storage device that can be positioned to any portion of a file. Thus, the entire

```
      PROGRAM TP
      INTEGER REC(513),RNUMB
      REWIND 1
      DO 4 RNUMB = 1,1000
1     BUFFER IN (1,1) (REC(1),REC(512))
2     IF(UNIT(1)) 3,5,5
3     K = LENGTH(1)
C  LENGTH RETURNS NUMBER OF WORDS TRANSFERRED BY BUFFER IN.
4     PRINT 100, RNUMB, (REC(I), I = 1,K)
100   FORMAT ('0RECORD', I5, /, (1X,A10))
5     STOP
      END
```

Figure 5-41. BUFFER IN Example

concept of file position does not apply to a direct access file. The notion of rewinding a direct access file is, for instance, without meaning.

To create a direct access file the user must specify an OPEN statement with ACCESS='DIRECT' and include the RECL (record length) specifier. For example:

```
OPEN(2,FILE='DAFL',ACCESS='DIRECT',RECL=120)
```

opens an unformatted file DAFL for direct access. The file is associated with unit 2 and has a record length of 120 words.

The record length of a direct access file must be specified in the OPEN statement, and all records have the same length.

The record length for a formatted direct access file is specified in characters. The record length for an unformatted direct access file is specified in words. If the iolist for an unformatted WRITE contains character data, the record length to be written is still specified in words and can be determined by the following rules:

1. Each noncharacter item counts as one word except for double precision and complex items, which count as two words.

2. The length in words of each contiguous group of character items is determined by adding 9 to the combined length of the items in characters and dividing this result by 10, discarding the fractional part.

3. The lengths calculated in steps 1 and 2 are added to determine the record length in words.

Example:

```
      CHARACTER A*7,B*9,C*10,D*20,E*15,F*12
      INTEGER IA,IB,IC,ID(5)
      OPEN (5,ACCESS='DIRECT',
     *FORM='UNFORMATTED',RECL=100)
      WRITE (5,REC=1) A,B,IA,C,IB,E,D,ID,F
```

The length of the output record is determined by the following calculation:

(length of A + length of B + 9)/10	= 2 words
IA	= 1 words
(length of C + 9)/10	= 1 words
IB	= 1 words
(length of E + length of D + 9)/10	= 4 words
ID	= 5 words
(length of F + 9)/10	= 2 words

Record length = 2+1+1+1+4+5+2=16 words

Records in a direct access file are identified by a record number. The record number is a positive decimal integer that is assigned when the record is written. Once a record is written with a record number, it can always be accessed with the same number. The order of records on a direct access file is the order of their record numbers. Records can be written, rewritten, or read by specifying the record number in a READ or WRITE statement. Records can be read or written in any order; they need not be referenced in the order of their record numbers. The number of the record to be read or written is specified in a READ or WRITE statement with the REC=rn specifier.

If the length of the iolist in a direct access formatted WRITE statement is less than the record length of the direct access file, the unused portion of the record is blank filled. A direct access WRITE statement must not write a record longer than the record length.

A direct access file can be opened for formatted or unformatted input/output. However, list directed input/output cannot be used with direct access files.

An internal file cannot be opened for direct access. A discussion of internal files follows in this section.

Example:

```
      WRITE(2,'(3E10.4)',REC=6)A,B,C
      WRITE(2,'(2I4,G20.10)',REC=1)I,J,X
```

Variables A, B, and C are written to record number 6, and variables I, J, and X are written to record number 1 of the direct access file associated with unit 2.

Example:

```
        OPEN(2,FILE='DARG',ACCESS='DIRECT',
     *FORM='FORMATTED',RECL=72)
        DO 14 I=10,2,-2
        READ(2,99,REC=I,ERR=20) (A(J),J=1,6)
   99 FORMAT (6E12.6)
        .
        .
        .
   14 CONTINUE
```

Records 10, 8, 6, 4, and 2 are read from the direct access file DARG.

INPUT/OUTPUT STATUS STATEMENTS

FORTRAN provides three statements that can be used to establish, examine, or alter certain attributes of files used for input or output. These are the OPEN, INQUIRE, and CLOSE statements.

OPEN

The OPEN statement can be used to associate an existing file with a unit number, to create a new file and associate it with a unit number, or to change certain attributes of an existing file. The OPEN statement is shown in figure 5-43.

The UNIT= parameter is required; all other parameters are optional except that the RECL parameter must be specified if a file is being opened for direct access. If a STATUS of OLD or NEW is specified, a FILE=specifier must be given.

If the FILE= parameter is omitted, the file is assumed to be the one associated with the specified unit in the PROGRAM statement (described in section 6). If the file is not specified on the PROGRAM statement, the file name is derived from the unit number. For unit numbers in the range 0 through 999, the file name is TAPEn where n is the unit number; for unit numbers having the form of a logical file name, the file name will be the same as the unit number.

A declaration in an OPEN statement overrides a declaration in a preceding PROGRAM statement provided no input/output operations have been performed on the file.

OPEN ([UNIT=]u[,IOSTAT=ios] [,ERR=sl] [,FILE=fin] [,STATUS=sta] [,ACCESS=acc] [,FORM=fm] [,RECL=rl] [,BLANK=blnk] [BUFL=bl])

u — Specifies the unit number of the file to be opened. (See File Usage.)

ios — Is an integer variable that contains an error number if an error occurs during the open, or zero if no errors occur.

sl — Is the label of an executable statement to which control transfers if an error occurs during the open.

fin — Is a character expression (seven characters or fewer; first character must be a letter) whose value is the name of the file to be opened. Trailing blanks are removed. This file becomes associated with unit u.

sta — Is a character expression specifying file status. Valid values are:

'OLD'	File fl currently exists.
'NEW'	File fl does not currently exist.
'SCRATCH'	Delete the file associated with unit u on program termination or execution of CLOSE that specifies unit u; must not appear if FILE parameter is specified.
'UNKNOWN'	File status is unknown.

Default is STATUS='UNKNOWN'.

acc — Is a character expression specifying the access method of the file. Valid values are:

'SEQUENTIAL'	File is to be opened for sequential access.
'DIRECT'	File is to be opened for direct access.

Default is ACCESS='SEQUENTIAL'.

If the file exists, the access method must be valid for the existing file.

fm — Is a character expression having one of the following values:

'FORMATTED'	File is being opened for formatted input/output.
'UNFORMATTED'	File is being opened for unformatted input/output.

Default is FORM='FORMATTED' for sequential access files, FORM='UNFORMATTED' for direct access files.

For an existing file, the specified form must be valid for that file.

rl — Is an integer variable or positive integer constant specifying the record length for a direct or sequential access file. RECL is required for a direct access file; if omitted for a sequential access file, it defaults to 150 characters or 15 words.

blnk — Is a character expression having one of the following values:

'NULL'	Blank values in numeric formatted input fields are ignored, except that a field of all blanks is treated as zeros.
'ZERO'	Blanks, other than leading blanks, are treated as zeros.

Default is BLANK='NULL'.

bl — Is a nonnegative integer or Boolean expression specifying the file buffer length. Default is system selected based on device type.

Figure 5-43. OPEN Statement

Example:

```
PROGRAM XX (TAPE2=500)
.
.
.
OPEN (2,BUFL=1000,FILE='FILEY')
READ(2,100)A,B,C
```

The PROGRAM statement declares a 500 word buffer for unit 2. The OPEN statement specifies a 1000 word buffer for unit 2, which is used. The READ statement reads data from FILEY.

Declarations of file properties specified on a FILE control statement override any conflicting OPEN statement parameters for a unit associated with that file; this applies to all OPEN statements for that unit. For example, an MRL or FL specification on a FILE control statement always overrides the RECL parameter value specified in an OPEN statement.

Once properties of a file have been established in an OPEN statement, only the BLANK= parameter can be changed in a subsequent OPEN statement for that file, unless the file is first closed in a CLOSE statement.

Once a file has been associated with a particular unit, the file can be associated with another unit in a subsequent OPEN statement. The file is then associated with more than one unit. In this case the unit numbers refer to the same file. Actions taken on one unit also affect the other unit. For example, closing a unit closes all other units associated with the same file.

Example:

```
OPEN (2,FILE='INFIL')
.
.
.
OPEN (3,FILE='INFIL')
READ (2,100) A,B
READ (3,100) X,Y
```

Both READ statements read from file INFIL.

Example:

```
OPEN (3,FILE='XXX'STATUS='OLD',BLANK='ZERO')
```

When data is read from the existing file XXX, blanks will be interpreted as zeros.

Example:

```
OPEN (2,STATUS='NEW',ERR=12,FILE='NEWFL',
*ACCESS='SEQUENTIAL')
```

A new file, NEWFL, is associated with unit 2 and is to be a sequential access file.

If a file is associated with a unit and a succeeding OPEN statement associates a different file with the same unit, the effect is the same as performing a CLOSE without a STATUS= specifier on the currently associated file before associating the new file with the unit. For example:

```
OPEN (2,FILE='MYFILE')
WRITE (2,'(A)')A,B,C
OPEN (2,FILE='PART2')
```

In this example, the second OPEN statement closes MYFILE before opening PART2.

CLOSE

The CLOSE statement disconnects a file from a specified unit and specifies whether the file connected to that unit is to be kept or released. The CLOSE statement is shown in figure 5-44.

CLOSE ([UNIT=] u[,IOSTAT=ios] [,ERR=sl] [,STATUS=sta])

u Is the unit designator of the file to be closed.

ios Is an integer variable which, upon completion of the CLOSE, contains the error number; a value of 0 indicates no errors occurred.

sl Is the label of an executable statement to which control transfers if an error occurs during the close.

sta Is a character expression that determines the disposition of the file associated with the specified unit. Valid values are:

'KEEP'	The file is kept after execution of the CLOSE statement.
'DELETE'	The file is unloaded after execution of the CLOSE statement.

Default is STATUS='DELETE' if file status is 'SCRATCH'; otherwise, the default is STATUS='KEEP'.

'KEEP' is not valid for a file whose status is 'SCRATCH'.

Figure 5-44. CLOSE Statement

A CLOSE statement can appear in any program unit in the program; it need not appear in the same program unit as the OPEN statement specifying the same unit.

A CLOSE statement that references a unit that does not have a file connected to it has no effect.

After a unit has been disconnected by a CLOSE statement, it can be connected again within the same program to the same file or to a different file. A file connected to a unit specified in a CLOSE statement can be connected again to the same or to another unit, provided the file still exists.

File equivalence established on the PROGRAM statement or on the execution control statement is no longer in effect after the CLOSE statement is executed.

When a program terminates normally, an implicit CLOSE (u,STATUS='KEEP') occurs for each connected unit unless the status of the file was SCRATCH; in this case, a CLOSE (u,STATUS='DELETE') occurs.

Example:

```
CLOSE (2,ERR=25,STATUS='DELETE')
```

INQUIRE

There are two forms of the INQUIRE statement: inquire by unit is used to obtain information about the current status of a specified unit; inquire by file is used to obtain information about the current status of a file. The INQUIRE statement is shown in figure 5-45.

Either a file name (inquire by file) or a unit specifier (inquire by unit), but not both, must be specified in an INQUIRE statement. The file or unit need not exist when INQUIRE is executed. Following execution of an INQUIRE statement, the specified parameters contain values that are current at the time the statement is executed. If a unit number is specified and the unit is opened, the

INQUIRE ({ [UNIT=] u / FILE=fin } [,IOSTAT=ios] [,ERR=sl] [,EXIST=ex] [,OPENED=od] [,NUMBER=num] [,NAMED=nmd] [,NAME=fn]
[,ACCESS=acc] [,SEQUENTIAL=seq] [,DIRECT=dir] [,FORM=fm] [,FORMATTED=fmt] [,UNFORMATTED=unf]
[,RECL=rcl] [,NEXTREC=nr] [,BLANK=blnk])

u — Is the external unit for which information is to be returned; if the unit is associated with a file, information about the file is returned. (The format of this parameter is described under File Usage.)

fin — Is a character expression specifying the name of the file for which information is to be returned.

ios — Is an integer variable which, upon completion of the INQUIRE, contains an error number; the value is 0 if no errors occurred.

sl — Is a user-specified statement label of an executable statement to which control passes if an error occurs during an inquire.

ex — Is a logical variable:

- .TRUE. The file (unit) exists.
- .FALSE. The file (unit) does not exist.

od — Is a logical variable:

- .TRUE. The file (unit) is connected to a unit (file).
- .FALSE. The file (unit) is not connected to a unit (file).

num — Is an integer variable containing the external unit number of the unit currently associated with the file; undefined if the file is not associated with a unit.

nmd — Is a logical variable:

- .TRUE. The file has a name.
- .FALSE. The file does not have a name.

fn — Is a character variable containing the name of the file associated with unit u.

acc — Is a character variable indicating the access method of the file:

- 'SEQUENTIAL' The file is opened for sequential access input/output.
- 'DIRECT' The file is opened for direct access input/output.

If the file is not opened, acc is undefined.

seq — Is a character variable indicating whether the file can be opened for sequential access input/output:

- 'YES' The file can be opened for sequential access input/output.
- 'NO' The file cannot be opened for sequential access input/output.
- 'UNKNOWN' Cannot be determined.

dir — Is a character variable indicating whether the file can be opened for direct access input/output:

- 'YES' The file can be opened for direct access input/output.
- 'NO' The file cannot be opened for direct access input/output.
- 'UNKNOWN' Cannot be determined.

Figure 5-45. INQUIRE Statement (Sheet 1 of 2)

fm	Is a character variable indicating formatted or unformatted input/output:
	'FORMATTED' The file is opened for formatted input/output.
	'UNFORMATTED' The file is opened for unformatted input/output.
	If the file has not been opened, fm is undefined.
fmt	Is a character variable specifying whether the file can be opened for formatted input/output:
	'YES' The file can be opened for formatted input/output.
	'NO' The file cannot be opened for formatted input/output.
	'UNKNOWN' It cannot be determined if the file can be opened for formatted input/output.
unf	Is a character variable specifying whether the file can be opened for unformatted input/output:
	'YES' The file can be opened for unformatted input/output.
	'NO' The file cannot be opened for unformatted input/output.
	'UNKNOWN' It cannot be determined if the file can be opened for unformatted input/output.
rcl	Is an integer variable containing the record length of a file opened for direct access. If the file is 'FORMATTED', rcl contains the record length in characters; if 'UNFORMATTED', the record length is in words; undefined if the file is not opened for direct access.
nr	Is an integer variable; for a direct access file, nr contains the record number of the next record to be read or written. If no records have been read or written, nr contains 1. Undefined for sequential files.
blnk	Is a character variable:
	'NULL' Null blank control is in effect for a file opened for formatted input/output.
	'ZERO' Zero blank control is in effect for a file opened for formatted input/output.
	Undefined if the file is not opened for formatted input/output.

Figure 5-45. INQUIRE Statement (Sheet 2 of 2)

NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, OPENED, EXIST, NUMBER, ACCESS, and BLANK specifiers will contain information about the file associated with the unit. If a file name is specified, the NAMED, NAME, SEQUENTIAL, DIRECT, FORMATTED, UNFORMATTED, OPENED, EXIST, NUMBER, ACCESS, FORM, RECL, NEXTREC, and BLANK specifiers will contain information about the file and the unit it is associated with. If a file is specified that is associated with more than one unit, the NUMBER specifier will contain one of the unit numbers or names.

If a nonexistent file or unit is specified, no error results but certain parameters are not assigned values. Note that if a unit that is not associated with a file is specified, only the IOSTAT and EXIST parameters contain values.

If an error occurs during an INQUIRE, only IOSTAT contains a value.

Example:

```
LOGICAL EX
CHARACTER*10 AC
.
.
.
INQUIRE (FILE='AFILE', ERR=100, EXIST=EX,
*ACCESS=AC)
```

INTERNAL FILES

Internal files provide a means of reformatting and transferring data from one area of memory to another. Input and output on internal files are performed by formatted READ and WRITE statements and the ENCODE and DECODE statements. However, no input/output devices are involved. Internal files allow data to be reformatted without the necessity of writing it and rereading it under a different format specification. Internal files also allow numeric conversion to or from character data type. The two types of internal files are standard internal files and extended internal files.

STANDARD INTERNAL FILES

A standard internal file can be any character variable, array, or substring. If the file is a variable or substring, it consists of a single record whose length is the length of the variable or substring. If the file is an array, each array element constitutes a single record. For example:

```
CHARACTER *20 A(100)
```

The internal file A contains 100 records of 20 characters each.

LCM resident internal files are restricted to a maximum length of 150 characters when used in READ or WRITE statements.

Records of an internal file are defined by storing data into the records, either with an output statement or an assignment statement.

It is not necessary to declare internal files in the same manner as external files. Only formatted input/output can be used; unformatted, list directed, NAMELIST, and buffer input/output are not valid for internal files. In addition, file manipulation and file status statements cannot be used with internal files. Some sample programs using internal files are included in section 12.

Output

Data is written to standard internal files using a formatted WRITE statement (figure 5-3) in which the internal unit specifier u is a character variable, array, or substring name. The WRITE statement transmits data from the variables specified in iolist to consecutive locations starting with the leftmost character of the location specified by u; data is converted from internal to character format according to the format specification. The number of characters transmitted is determined by the record length.

Figure 5-46 shows some examples of internal files used for output.

Example 1:

```
INTEGER A,B,C,D
CHARACTER*4 AR(4)
      .
      .
      .
A=123
B=-27
C=104
D=1234
WRITE (AR,'(I4)')A,B,C,D
```

In memory:

Δ123	Δ-27	Δ104	1234

The WRITE statement defines an internal file, AR, and writes four records to the file.

Example 2:

```
CHARACTER*8 BIRD(3),A*1,B,C
      .
      .
      .
A='Z'
B='ABCDE'
C='12345678'
WRITE (BIRD,'(A1/A5/A8)')A,B,C
```

In memory:

ZΔΔΔΔΔΔΔ	ABCDEΔΔΔ	12345678

BIRD(1) BIRD (2) BIRD(3)

The WRITE statement defines an internal file, BIRD, which contains three records (array elements).

Figure 5-46. Internal File Output Examples

Input

Data is read from a standard internal file using a formatted READ statement (figure 5-1) in which the internal unit identifier is a character variable, array, or substring. Data is transferred from consecutive locations starting at the first character position of u, converted under format specification, and stored in the variables specified in iolist.

Some examples of internal files used for input are shown in figure 5-47.

Example 1:

```
CHARACTER*3 ZT(6),A,B,C
      .
      .
      .
READ (ZT,'(A3)')A,B,C
```

Contents of ZT:

CAT	DOG	RUN

ZT(1) ZT(2) ZT(3)

Stored in A, B, C:

A CAT

B DOG

C RUN

Example 2:

```
CHARACTER CN*12
      .
      .
      .
READ (CN,'(4I3)')I,J,K,L
```

Contents of CN:

2ΔΔΔ56Δ4ΔΔΔ8

Stored in I,J,K,L (internal integer format):

I 2

J 56

K 4

L 8

Figure 5-47. Internal File Input Examples

EXTENDED INTERNAL FILES

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.

An extended internal file can be any noncharacter variable, array, or array element. A record of an extended internal file is defined by writing the record. The record length is measured in characters. Since one word contains 10 characters, the record length of an extended internal file is given by:

```
10*a
```

where a is the number of words in the record.

An extended internal file residing in LCM cannot consist of more than 15 words.

ENCODE

The ENCODE statement, shown in figure 5-48, is the extended internal file output statement.

ENCODE (c, fn, u) iolist

c	**Is an unsigned integer constant or variable having a value greater than zero; c specifies the number of characters to be transferred per record. The record length is calculated from c. Must not exceed 150 if the file resides in level 2 storage on a CYBER 170/Model 176, CYBER 70/Model 76, or 7600 computer.**
fn	**Is a statement label of a FORMAT statement, or a character expression whose value is a format specification; fn must not specify NAMELIST or list directed formatting.**
u	**Is an extended internal file (noncharacter variable, array element or array name) in which the record is to be encoded.**
iolist	**Is a list of noncharacter variables, arrays or array elements to be transmitted to the location specified by u.**

Figure 5-48. ENCODE Statement

ENCODE is similar to an internal file formatted WRITE. Values are transferred to the receiving storage area from the variables specified in iolist under the specified format. The first record starts with the leftmost character of the location specified by u. The length in characters of each record is:

```
INT((c+9)/10)*10
```

where INT(a) is the largest integer less than or equal to a. If c is less than the record length, the remainder of the word is blank filled.

The internal file must be large enough to contain the total number of characters transmitted by the ENCODE statement. For example, if 70 characters are generated by the ENCODE statement, the array starting at location v must be at least 70 characters (7 words) in length. If A is the receiving array the declaration BOOLEAN A(7) would be sufficient.

If 27 characters are generated, the declaration BOOLEAN A(3) is sufficient.

If the list and the format specification transmit more than the number of characters specified per record, an execution error message is printed. If the number of characters transmitted is less than the record length, remaining characters in the record are blank filled.

ENCODE can be used to calculate a field definition in a format specification at execution time.

Example:

```
      BOOLEAN SPECMAT(1)
      .
      .
      .
      IF(M.GE.10.OR.M.LE.1) GO TO 2
      ENCODE (10,100,SPECMAT)M
  100 FORMAT ('(2A10,I',I1,1H))
      .
      .
      .
      PRINT SPECMAT,A,B,J
```

In this example, the programmer wishes to specify m in the statement FORMAT (2A10,Im) at some point in the program. The variable M is permitted to vary in the range 2 through 9. M is tested to ensure it is within limits; if it is not, control goes to statement 2 which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters (2A10,I). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A10,Im).

A and B will be printed under specification A10, and the quantity J under specifications I2 through I9, according to the value of m.

Example:

```
      PROGRAM IGEN
      INTEGER FMT(1)
      DO 9 J=1,50
      ENCODE (10,7,FMT)J
    7 FORMAT ('(I',I2,')')
    9 WRITE (6,FMT)J
      STOP
      END
```

In memory, FMT is first (I1), then (I2), then (I3), and so forth.

An area in memory should not be encoded or decoded upon itself, as this gives unpredictable results.

DECODE

The DECODE statement, shown in figure 5-49, is the extended internal file input statement.

DECODE (c, fn, u) iolist

c, fn, and u are as described for ENCODE.

iolist	**Is a list of noncharacter variables, arrays, or array elements to receive data from the extended internal file specified by u.**

Figure 5-49. DECODE Statement

DECODE processing of an illegal character for a given conversion specification produces a fatal error. If DECODE is processing an A or R format specification and encounters a zero character (6 bits of binary zero), the character is treated as a colon under 64-character set or as a blank under 63-character set.

DECODE can be used to pack the partial contents of two words into one. For example:

In memory:

```
LOC1    SSSSSxxxxx
LOC2    xxxxxDDDDD
```

The following statements store SSSSSDDDDD in location NAME:

```
      BOOLEAN LOC1,LOC2,TEMP,NAME
      .
      .
      .
      DECODE (10,1,LOC2)TEMP
    1 FORMAT (5X,A5)
      ENCODE (10,2,NAME)LOC1,TEMP
    2 FORMAT (2A5)
```

The DECODE statement places the last 5 display code characters of LOC2 into the first 5 characters of TEMP. The ENCODE statement packs the first 5 characters of LOC1 and TEMP into NAME.

Some additional DECODE examples are illustrated in figure 5-50.

FILE POSITIONING STATEMENTS

Three statements can be used to position files connected for sequential access: REWIND, BACKSPACE, and ENDFILE.

REWIND

The REWIND statement, shown in figure 5-51 positions a file at beginning-of-information so that the next input/output operation references the first record in the file, even though several ENDFILE statements may have been issued to that unit since the last REWIND. If the file is already at beginning-of-information, no action is taken. (Refer to BACKSPACE/REWIND, appendix F, for further information.)

Example:

```
REWIND 3
```

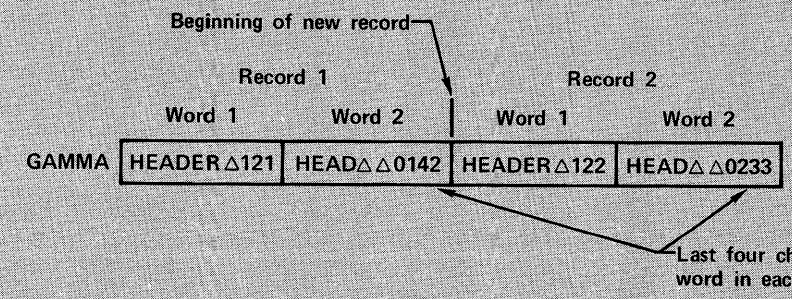
REWIND ([UNIT=]u[,IOSTAT=ios] [,ERR=sl])

REWIND u

u	**Is an external unit specifier.**
ios	**Is an integer variable which, after execution of REWIND, contains an error number; a value of 0 indicates no errors occurred.**
sl	**Is a statement label of an executable statement to which control transfers if an error occurs during the rewind.**

Figure 5-51. REWIND Statement

```
      INTEGER GAMMA (4)
      DECODE (16,1,GAMMA) X,B,C,D
    1 FORMAT (2A8)
```



Data transmitted under this DECODE specification would appear in storage as follows:

```
X=HEADER 1
B=21HEAD
C=HEADER 1
D=22HEAD
```

Figure 5-50. DECODE Example

BACKSPACE

The BACKSPACE statement, shown in figure 5-52, backspaces unit u one record. When the file is positioned at beginning-of-information, this statement acts as a do-nothing statement. Backspace operations should not be used on direct access files or on records created by list directed or NAMELIST output.

BACKSPACE ([UNIT=]u[,IOSTAT=ios] [,ERR=sl])

BACKSPACE u

u, ios, and sl are as described for REWIND.

Figure 5-52. BACKSPACE Statement

Example:

```
      DO 1 LUN = 1,4
    1 BACKSPACE LUN
```

The files associated with units 1 through 4 are backspaced one record.

ENDFILE

The ENDFILE statement, shown in figure 5-53 writes an end-of-partition (end-of-file) on the designated unit. ENDFILE is not permitted on units opened for direct access. The end-of-partition boundary can be detected by the END= and IOSTAT= specifiers.

ENDFILE ([UNIT=]u[,IOSTAT=ios] [,ERR=sl])

ENDFILE u

u, ios, and sl are as described for REWIND.

Figure 5-53. ENDFILE Statement

Because the file mode (formatted or unformatted) cannot be detected, ENDFILE should not be the first operation on a file.

Meaningful results are not guaranteed if ENDFILE is used on a file processed by mass storage subroutines.

Example:

```
      IOUT = 7
      ENDFILE (UNIT=IOUT, ERR=100)
```

End-of-partition is written on unit 7.

PROGRAM UNITS AND PROCEDURES 6

An executable program contains one main program unit and zero or more subprograms. Each subprogram is a program unit. A program unit is a group of FORTRAN statements, with optional comments, terminated by an END statement.

A main program is a program unit that does not begin with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. Usually, a main program begins with a PROGRAM statement, but this statement can be omitted. Execution of any program begins with the main program unit.

A subprogram is a program unit that begins with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. A subprogram is defined separately and can be compiled independently of a main program. A subprogram that begins with a SUBROUTINE or FUNCTION statement is a procedure subprogram and can accept one or more values through a list of arguments, through common blocks, or both. A subprogram that begins with a BLOCK DATA statement is a specification subprogram.

A procedure can be a function subprogram (external or intrinsic), a subroutine subprogram, or a statement function. Intrinsic functions are FORTRAN-supplied procedures and are available to any programmer (section 7). External functions, subroutines, and statement functions are provided by the programmer.

Functions return single values through the function names. Function subprograms defined by the programmer can also return values through a list of arguments, through common blocks, or both.

This section discusses programmer-written procedures, which include statement functions, function subprograms, and subroutine subprograms. FORTRAN-supplied procedures, which include intrinsic functions and utility subprograms, are discussed in section 7. The only subprogram that is not a procedure is the block data subprogram, which is not executable.

Table 6-1 summarizes the characteristics of procedures and subprograms.

MAIN PROGRAMS

A main program can contain any FORTRAN statements except FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY. The main program should have a PROGRAM statement and at least one executable statement followed by an END statement. No executable program can have more than one main program unit, except an overlay-structured program, which has one main program unit in each overlay.

TABLE 6-1. CHARACTERISTICS OF PROCEDURES AND SUBPROGRAMS

Main Program	Subroutine	External Function	Intrinsic Function	Statement Function	Block Data Subprogram
-	Procedure	Procedure	Procedure	Procedure	-
-	Subprogram	Subprogram	-	-	Subprogram
-	-	Function	Function	Function	-
User-written	User-written	User-written	Supplied	User-written	User-written
Separate program unit	Separate program unit	Separate program unit	In the FORTRAN library	Within a program unit	Separate program unit
Not typed	Not typed	Typed implicitly or explicitly	Typed by intrinsic function name, or generic	Typed implicitly or explicitly	Not typed
-	Alternate RETURN allowed	RETURN allowed	Single RETURN, effectively	Immediate RETURN, effectively	-
-	Accepts values through arguments or common blocks	Accepts values through arguments or common blocks	Accepts values through arguments	Accepts values through arguments	-
-	-	Returns a value for the function name	Returns a value for the function name	Returns a value for the function name	-

The main program can be compiled independently of any subprograms. When a main program is loaded into memory for execution, all the required subprograms must also be loaded and ready for execution.

PROGRAM STATEMENT

The PROGRAM statement defines the program name that is used as the entry point name and as the object deck name for the loader. Figure 6-1 shows the syntax for the PROGRAM statement.

PROGRAM name[(fpar[,fpar]. . .)]

name	**Is the program name that cannot be used elsewhere in the program as a user-defined name.**
fpar	**Can declare the file in any of the following forms:** **file** **file=n** **file=/r** **file=n/r** **altunit=file**
file	**Is the file name of a file required by the main program or its subprograms; the maximum number of file names is 49.**
file=n	**Specifies an integer or octal constant for the buffer length; default length is 1003 octal words. n is ignored if specified in a program run under SCOPE 2.**
file=/r	**Specifies the maximum length in characters for list directed, formatted, and NAMELIST records; default length is 150 characters.**
file=n/r	**Specifies both the buffer length and record length for the file.**
altunit=file	**Specifies that two file names are equivalent. The buffer length and record length for altunit are the same as previously specified (or defaulted) for file. The two files share the same FIT. Usually, altunit has the form TAPEu where u is an integer in the range 0 through 999.**

Figure 6-1. PROGRAM Statement

In a program structured for overlays, the fpar list is used only in the PROGRAM statement for the main overlay. The fpar list cannot be used for the PROGRAM statements of primary or secondary overlays.

PROGRAM STATEMENT USAGE

The PROGRAM statement can declare files that are used in the program and in any subprograms that are called. If this statement is omitted from the main program, the program is assumed to have the name START. and two files named INPUT and OUTPUT.

Files referenced in input/output statements need not be declared in a PROGRAM statement. If a file is not declared on the PROGRAM statement, a buffer will be created on the first reference to the file.

FORTRAN input/output routines add the characters TAPE as a prefix to the unit number to form the file name. TAPE3 is the file name assigned to unit number 3 and TAPE5 is the file name assigned to unit number 5. TAPE5 and TAPE05 do not specify the same file name. TAPEu refers to a file located on rotating mass storage unless specified otherwise in the job deck before the program is executed. The file is temporary unless made permanent by the user.

If a buffer length is specified on the program statement, FORTRAN input/output statements will use a buffer with the specified length. The buffer length can appear only with the first reference to the file in the PROGRAM statement. A buffer length of zero should be specified for a file referenced by a buffer statement, unless the file is a connected file or the file description has been changed by a FILE control statement. Since buffered records are transmitted directly into and out of central memory, field length of the program is reduced for each file declared with zero buffer length in the PROGRAM statement. The following values of n are minimal if default record and block types are used:

- For terminals, n is ignored. For efficiency, set the buffer length to zero.
- For mass storage input/output files, $n \geq 64$. Large records and sequential reading/writing execute faster with a larger buffer.
- For sequential files, the format controls the minimum value of n in the following way:

SI tape	128 for formatted, 512 for unformatted
I,X tape	512 for unformatted (NOS only)
S tape	512 for formatted or unformatted
L tape	$\geq$ maximum block length
mass storage	64 for formatted, 512 for unformatted

The appearance of a symbolic file name in the PROGRAM statement has the same effect as the execution of an OPEN statement (section 5) with that file name. Any later attempt to use the OPEN statement to change the buffer length results in an error, unless the file has been closed first.

Record length should always be specified for files referenced in list directed input/output statements. This specification creates a separate working storage area for the file, which is different from the default area. If the default area is used, input/output to other files destroys any data remaining after a list directed read.

When file names are made equivalent, the buffer length and record length specified for the first file also apply to the specified altunit. Therefore, any attempt to specify buffer length or record length for altunit results in an error. An example of equivalent file names is shown in figure 6-2.

```
      PROGRAM SAMPLE (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      READ(5,100) A,B,C
100   FORMAT(3F7.3)
      WRITE(6,200) A,B,C
200   FORMAT(1H1,3F7.3)
      STOP
      END
```

Figure 6-2. File Equivalencing Example

The READ statement reads from logical unit 5. Since the PROGRAM statement declares file TAPE5 as equivalent to file INPUT, input is taken from file INPUT. The WRITE statement writes to logical unit 6, and the PROGRAM statement declares file TAPE6 as equivalent to file OUTPUT. Records are therefore written to file OUTPUT.

PROCEDURES

The main program unit is a procedure. Other procedures can be subroutines, function subprograms, intrinsic functions, and statement functions. The use of additional procedures depends on the needs of the program. If the program requires the evaluation of a standard function, then a FORTRAN intrinsic function can be used. If a single computation is needed repeatedly, a user-written statement function can be included in the program. If several statements are required to obtain a single value, a function subprogram can be written. If several statements are required to obtain more than one value, a subroutine can be written.

Procedures enable multiple executions of the same routine. Communication can be controlled through the use of common blocks or through passing actual arguments. Procedures (except statement functions) can be compiled independently of the main programs or other procedures.

BLOCK DATA SUBPROGRAM

A block data subprogram is the only subprogram that is not a procedure. The block data subprogram is a nonexecutable specification subprogram that can be used to enter initial values for variables and array elements in named common blocks. A program can have more than one block data subprogram. Only one block data subprogram can be unnamed; the name BLKDAT. is assigned to the unnamed block data subprogram.

The BLOCK DATA statement must appear as the first statement of the block data subprogram. The name used for the block data subprogram must not be the same as any local variables in the subprogram. The name is global and must not be the same as any other program unit or entry name in the program. The BLOCK DATA statement is shown in figure 6-3.

BLOCK DATA [sub]

sub — **Is the name of the block data subprogram.**

Figure 6-3. BLOCK DATA Statement

Block data subprograms can contain IMPLICIT, PARAMETER, DIMENSION, type, COMMON, SAVE, EQUIVALENCE, LEVEL, or DATA statements. A block data subprogram ends with an END statement. Data can be entered into more than one common block in a block data program. All variables having storage in the named common must be specified even if they are not all initially defined. A sample block data subprogram with two named common blocks is shown in figure 6-4.

```
BLOCK DATA ANAME
COMMON /CAT/ X,Y,Z /DEF/ R,S,T
COMPLEX X,Y
DATA X,Y /2*(1.0,2.7)/, R/7.6543/
END
```

Figure 6-4. Example of BLOCK DATA

In the example, not all entities in the common blocks are initially defined. The variable Z in block CAT, and the variables S and T in block DEF are not initially defined.

SUBROUTINE SUBPROGRAM

A subroutine subprogram is executed when a CALL statement naming the subroutine is encountered in a program unit. A subroutine must not directly or indirectly call itself. The subroutine communicates with the calling program unit through a list of arguments passed with the CALL statement or through common blocks.

The SUBROUTINE statement must appear as the first statement of the subroutine subprogram and contains the symbolic name that is the main entry point of the subprogram. The subprogram name is not used to return results to the calling program. The name must not be the same as any other program unit or entry name. The name also cannot be the same as any name in the subroutine. The SUBROUTINE statement is shown in figure 6-5.

SUBROUTINE sub[([d[,d]. . .])]

sub — **Is the name of the subroutine subprogram. If there are no dummy arguments, either sub or sub() can be used.**

d — **Is a dummy argument that can be a variable name, array name, dummy procedure name, or *.**

Figure 6-5. Subroutine Statement

Subroutines can contain any statements except a PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. Subroutines begin with a SUBROUTINE statement and end with an END statement. If control flows into the END statement, then a RETURN is implied. Control is returned to the calling program unit when a RETURN or END statement is encountered.

An example of a subroutine call is shown in figure 6-6.

Subroutine ERROR1 is called and executed if A-B is less than zero. Control returns to statement 20. The example illustrates that arguments need not be used.

In a subroutine subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a subroutine call. The SUBROUTINE statement can also have dummy arguments for statement labels; these arguments are represented by asterisks.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in subroutine subprograms. More details can be found later in this section under Referencing a Procedure.

FUNCTION SUBPROGRAM

Function subprograms can be external functions, intrinsic functions, or statement functions. Both external and intrinsic functions are specified externally from the program unit that referenced them; statement functions are contained within the referencing program unit.

External Functions

A function subprogram performs a set of calculations when the name appears in an expression in the referencing program unit. A function must not directly or indirectly reference itself. The function subprogram communicates with the referencing program unit through a value associated with the function symbolic name, through a list of arguments, or through common blocks.

The function statement must appear as the first statement of the function subprogram. The FUNCTION statement contains the symbolic name that is used as the main entry point of the subprogram. A function can have more than one entry point. The FUNCTION statement is shown in figure 6-7.

[typ] FUNCTION fun([d[,d]. . .])

typ Is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BOOLEAN, or CHARACTER*len. The len value specifies the length of the result of the character function.

fun Is the name of the function subprogram; fun is an external function name.

d Is a dummy argument that can be a variable name, array name, or dummy procedure name.

Figure 6-7. FUNCTION Statement

The symbolic name of a function subprogram, or an associated entry name of the same type, is a variable name in the function. The symbolic name specified in a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type statement. If the type of a function is specified in a FUNCTION statement, then the function name cannot appear in a type statement. In an executable statement, the symbolic name can appear only as a variable. During execution, this variable becomes defined and can be referenced or redefined. The value of the function is the value of this variable when control returns to the referencing program unit.

The type of the function name must be the same in the referencing program unit and the referenced function subprogram. When type is omitted, the type of the function is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when the function name is not explicitly typed. The name cannot have its type explicitly specified more than once.

If the name of a function subprogram is of type character, then each entry name must be type character and vice versa. The length of the function symbolic name and any entry names in the function must be specified with the same length. For example, if the function name has a length of (*), all entry names must have a length of (*).

```
      PROGRAM MAIN(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      INTEGER A,B
      READ(5,100) A,B
100   FORMAT(2I2)
      IF (A-B) 10,20,20
10    CALL ERROR1
20    RESULT = (A*100) + 375.2
      STOP
      END
C
      SUBROUTINE ERROR1
      WRITE(6,1)
1     FORMAT(5X, 'NUMBER IS OUT OF RANGE')
      RETURN
      END
```

Figure 6-6. Subroutine Call Example

The symbolic name of a function subprogram must not be the same as any other name, except a common block name. The name can be the same as a name in the function subprogram, if used as a variable name.

Function subprograms can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, or another FUNCTION statement. They begin with a FUNCTION statement and end with an END statement. Control is returned to the referencing program unit when a RETURN or END is encountered; a RETURN statement of the form RETURN e (described under Referencing a Procedure) in a function subprogram is not allowed.

Although alternate returns are prohibited for function subprograms, multiple entries are allowed, as described later in this section. An example is shown in figure 6-8.

```
      PROGRAM MAIN
      INTEGER Z
      Z = JOR(5,3)
      ZZ = JAM(5,3)
      PRINT *, Z,ZZ
      STOP
      END
C
      FUNCTION JOR (X,Y)
      INTEGER X,Y
      JOR = X-Y
      RETURN
      ENTRY JAM(X,Y)
      JAM=X+Y
      RETURN
      END
```

Figure 6-8. Function Reference

Function subprogram JOR is executed when the name appears in the calling program unit. The alternate entry point is entry JAM in function JOR.

In a function subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a function reference.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in function subprograms, as described later in this section under Referencing a Procedure.

Intrinsic Functions

Intrinsic functions are supplied by the FORTRAN library. The rules for using intrinsic functions are the same as for user-written function subprograms. An IMPLICIT statement does not change the type of an intrinsic function. Section 7 discusses intrinsic functions in detail, including generic and specific names, function definitions, type of arguments, and type of results.

Statement Functions

A statement function is a user-defined, single-statement computation that applies only to the program unit containing the definition. A statement function is a nonexecutable statement. A statement function must appear after the specification statements and before the first executable statement in the program unit. A statement function must not directly or indirectly reference itself.

A statement function is specified by a single statement and is similar to an arithmetic, logical, Boolean, or character assignment statement. The syntax for the statement function is shown in figure 6-9.

```
fun([d[,d]. . .]) = expr

    fun     Is the function name.

    d       Is a statement function dummy argument.

    expr    Is an expression in which each primary is an
            expression expr enclosed in parentheses, or is
            one of the following:

                Constant
                Symbolic constant
                Variable reference
                Array element reference
                Intrinsic function reference
                Reference to a statement function which
                  appears in the same program unit,
                  either before or after this statement
                External function reference
                Dummy procedure reference
                Substring reference
```

Figure 6-9. Statement Function

The symbolic name of the function is a variable and contains the value of the expression after execution. During execution, the actual argument expressions are evaluated, converted if necessary to the types of the corresponding dummy arguments according to the rules for assignment, and passed to the function. Thus, an actual argument cannot be an array name or a function name. In addition, if a character variable or array element is used as an actual argument, a substring reference to the corresponding dummy argument must not be specified in the statement function expression. The expression of the function is evaluated, and the resulting value is converted as necessary to the data type of the function. An example of a program that uses statement functions is shown in figure 6-10.

The symbolic name of a statement function is local and must not be the same as any other local name in the program unit, except a common block name. The name of a statement function cannot be an actual argument and must not appear in an INTRINSIC or EXTERNAL statement. If the statement function is used in a function subprogram, then the statement function can contain a reference to the name of the function subprogram or any of its entry names as a variable, but not as a function.

Each variable reference in the expression can be either a reference to a variable within the same program unit or to a dummy argument of the statement function. Statement functions can reference dummy variables that appear in a SUBROUTINE, FUNCTION, or ENTRY statement, but that statement must precede the statement function. If a statement function dummy argument is the same as another variable in the program unit and that name is referenced in the expression of the statement function, the reference is to the statement function dummy argument and not to the other variable. The names used for statement function dummy arguments have a scope of the statement function definition.

```
      PROGRAM SFUNC
      INTEGER SN
      DIMENSION AVG(25)
      ADD(A,B,C,D) = A+B+C+D
      AVRG(T1,T2,T3,T4) = ADD(T1,T2,T3,T4)/4
      GRADE(SCORE,HSCORE) = (SCORE/HSCORE) * 100
      SN=1
1     READ(*,100,END=20) S1,S2,S3,S4
100   FORMAT(4F5.1)
      AVG(SN)=AVRG(S1,S2,S3,S4)
      NS=SN
      SN = SN +1
      GO TO 1
20    HIGH = AVG(1)
      DO 30 SN = 1, NS
      IF(AVG(SN) .GT. HIGH) HIGH = AVG(SN)
30    CONTINUE
      DO 40 SN=1, NS
      CRVEDG = GRADE(AVG(SN),HIGH)
      PRINT *, CRVEDG
40    CONTINUE
      STOP
      END
```

Figure 6-10. Examples of Statement Functions

the reference is to the statement function dummy argument and not to the other variable. The names used for statement function dummy arguments have a scope of the statement function definition.

Multiple Entry

Each procedure subprogram has a primary entry point established by the SUBROUTINE or FUNCTION statement that begins the program unit. A subroutine call or function reference usually invokes the procedure at the primary entry point, and the first statement executed is the first executable statement in the program unit. ENTRY statements can be used to define other entry points. A procedure that contains one or more ENTRY statements has multiple entry points. The ENTRY statement is shown in figure 6-11.

ENTRY ep[([d[,d]. . .])]

ep Is an entry point name in a function or subroutine.

d Is a dummy argument that can be one of the following:

- A variable name
- An array name
- A dummy procedure name
- An asterisk, only if in a subroutine subprogram

Figure 6-11. ENTRY Statement

An ENTRY statement can appear anywhere after the SUBROUTINE or FUNCTION statement in the subprogram. An ENTRY statement cannot appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of the DO loop.

When an entry name is used to reference a procedure, execution begins with the first executable statement that follows the referenced entry point. An entry name is available for reference in any program unit, except in the procedure that contains the entry name. The entry name can appear in an EXTERNAL statement and (for a function entry name) in a type statement.

Each reference to a procedure must use an actual argument list that corresponds in number of arguments and type of arguments with the dummy argument list in the corresponding SUBROUTINE, FUNCTION, or ENTRY statement. Type agreement is not required for actual arguments that have no type, such as a dummy subroutine name. The dummy arguments for an entry point can therefore be different from the dummy arguments for the primary entry point or another entry point. No dummy argument can be used in an executable statement of a procedure unless it has already appeared in a FUNCTION, SUBROUTINE, or ENTRY statement.

A procedure with multiple entry points is shown in figure 6-12.

PROCEDURE COMMUNICATION

Communication between the referencing program unit and the referenced procedure can be through common blocks or by passing actual arguments to the procedure. Common blocks cannot be used to pass data to intrinsic functions or statement functions; the method used to pass data to these procedures is through an argument list. Common blocks and argument lists can be used for external, that is, user-written procedures, but passing procedure names to the external procedures can only be through an argument list.

ACTUAL ARGUMENTS

Actual arguments appear in the argument list of the referencing program unit. The referencing program unit passes actual arguments to the referenced procedure. The procedure receives values from the actual arguments and returns values to the referencing program unit. Actual arguments can be constants, symbolic names of constants,

```
      PROGRAM MAIN
      DIMENSION SET1(25)
1     READ 5,N
5     FORMAT(I1)
      IF(N .EQ. 0) GO TO 900
      IF(N .EQ. 1) CALL CLEAR(SET1)
      IF(N .EQ. 2) CALL FILL(SET1)
      DO 99 I=1,25
      PRINT 6, SET1(I)
6     FORMAT (F5.2)
99    CONTINUE
      GO TO 1
900   STOP
      END
C
      SUBROUTINE CLEAR(ARRAY)
      DIMENSION ARRAY(25)
20    DO 10 I= 1,25
      ARRAY(I) = 0.0
10    CONTINUE
      ENTRY FILL(ARRAY)
      READ *, VALUE,IPLACE
      IF(IPLACE .GT. 25) RETURN
      ARRAY(IPLACE) = VALUE
      RETURN
      END
```

Figure 6-12. Examples of ENTRY Statements

variables, array names, array elements, function references, and expressions. An actual argument cannot be the name of a statement function within the referencing program unit.

DUMMY ARGUMENTS

Dummy arguments appear in the argument list of the referenced procedure. Within the referenced procedure, the dummy arguments are associated with the actual arguments passed. Procedures use dummy arguments to indicate the types of actual arguments, the number of arguments, and whether each argument is a variable, array, procedure, or statement label. Dummy arguments for statement functions can only be variables. Since all names are local to the program unit, the same dummy argument name can be used in more than one procedure. A dummy argument appearing in a SUBROUTINE, FUNCTION, or ENTRY statement must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements except as a common block name. Dummy arguments used in array declarations for adjustable dimensions must be type integer. Dummy arguments representing array names must be dimensioned.

ARGUMENT ASSOCIATION

When a procedure is executed, the actual arguments and dummy arguments are matched up and each actual argument replaces each dummy argument. The type of the actual argument and the dummy argument must be the same. The actual arguments must be in the same order and there must be the same number as the dummy arguments in the referenced procedure. The actual arguments that are evaluated before the association of arguments include: expressions, substring expressions, and array subscripts. If the actual argument is a procedure name, the procedure must be available for execution at the time of the reference to the procedure.

A dummy argument is undefined unless it is associated with an actual argument. Argument association can exist at more than one level of procedure reference, and terminates within a program unit at the execution of a RETURN or END statement.

A subprogram reference can cause a dummy argument to be associated with another dummy argument in the referenced procedure. Any dummy arguments that become associated with each other can be referenced but must not be stored into during the execution of the procedure. For example, if a procedure is defined as:

```
SUBROUTINE ALPHA(X,Y)
```

and referenced with:

```
CALL ALPHA(A,A)
```

then the dummy arguments X and Y would each be associated with the actual argument A. X and Y would be associated with each other and therefore must not be stored into.

A subroutine reference can cause a dummy argument to become associated with an entity in a common block. For example, if a procedure contains the statements:

```
SUBROUTINE ALPHA(X)

COMMON Y
```

and the referencing program unit contains:

```
COMMON A

CALL ALPHA(A)
```

then the actual argument A causes the dummy argument X to become associated with Y, which is in blank common. In this case, X and Y cannot be stored into during execution of the subroutine.

Character Length

For type character, both the dummy and actual arguments must be of type character, and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument of type character is greater than the length of the dummy argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are used as the dummy argument.

If a dummy argument is an array name, length applies to the entire array and not to each array element. Length of array elements in the dummy argument can be different from length of array elements in the actual argument. The total length of the actual argument array must be greater than or equal to the total length of the dummy argument array.

When an actual argument is a character substring, the length of the actual argument is the length of the substring. If the actual argument expression involves concatenation, the sum of the lengths of the operands is the length of the actual argument.

Variables

A variable in a dummy argument can be associated with a variable, array element, substring, or expression in the actual argument. A procedure can define or redefine the associated dummy argument if the actual argument is a variable name, array element name, or substring name. The procedure cannot redefine the dummy argument if the actual argument is a constant, a symbolic constant, a function reference, an expression using operators, or an expression enclosed in parentheses.

Arrays

The array declaration in a type, COMMON, or DIMENSION statement provides the information needed for the array during the execution of the program unit. The actual argument array and the dummy argument array can differ in the number of the dimension and size of the array. A dummy argument array can be associated with an actual argument that is an array, array element, or array element substring.

If the actual argument is a noncharacter array name, the size of the actual argument array cannot be less than the size of the dummy argument array. Each actual argument array element is associated with the dummy argument array element that has the corresponding subscript value.

An association exists for array elements in a character array. Note that unless the lengths of the elements in the dummy and actual argument agree, the dummy and actual argument array elements might consist of different characters. For example, if a program unit has the following statements:

```
DIMENSION A(2)
CHARACTER A*2
.
.
.
CALL SUB(A)
```

and the subroutine has the following statements:

```
SUBROUTINE SUB(B)
DIMENSION B(2)
CHARACTER B*1
```

then the first character of A(1) corresponds to B(1) and the second character of A(1) corresponds to B(2).

If the actual argument is a noncharacter array element name, the size of the dummy argument cannot exceed (as+1-av), where as is the size of the actual argument array and av is the subscript value of the array element. For example, if the program unit has the following statements:

```
DIMENSION ARRAY(20)
.
.
.
CALL CHECK(ARRAY(3))
```

then the value of as is 20, and av is 3. The maximum dummy array size is 18 for the subroutine:

```
SUBROUTINE CHECK (DUMMY)
DIMENSION DUMMY(18)
.
.
.
SWAP= DUMMY(2)
```

The actual argument array elements are associated wih dummy argument array elements, starting with the first element passed. In the example, DUMMY(2) is associated with ARRAY(4), and DUMMY(18) is associated with ARRAY(20).

The association for characters is basically the same as for noncharacter array elements. The actual argument for characters can be an array name, array element name, or array element substring name. If the actual argument begins at character storage position acu of an array, then the first character storage position of the dummy argument array becomes associated with character storage position acu of the actual argument array, and so forth to the end of the dummy argument array.

Procedure Arguments

A dummy argument that is a dummy procedure can be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure. If the dummy argument is used as an external function, the actual argument that is passed must be a function or dummy procedure. The type of the dummy argument must agree with the type of result of all specific actual arguments that become associated with the dummy argument. When a dummy argument is used as an external function and is the name of an intrinsic function, the intrinsic function name corresponding to the dummy argument name is not available. If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure, and the dummy argument must not appear in a type statement or be referenced as a function.

Asterisk Arguments

A dummy argument that is an asterisk can only appear in the argument list of a SUBROUTINE or ENTRY statement in a subroutine subprogram. The actual argument is an alternate return specifier in the CALL statement.

Adjustable Dimensions

Adjustable dimensions enable creation of a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine SUM can accept is 10 elements. If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A, N)
DIMENSION A(N)
```

In this case, the value N is passed as an actual argument.

Character strings and arrays can also be adjustable, as in the subroutine:

```
SUBROUTINE MESSAG(X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares X with a length of (*) to accept strings of varying size. Note that the length of the string is not passed explicitly as an actual argument.

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument, but is determined by the number of elements stored into the array. If an array is dimensioned *, the array in the calling program must be large enough to contain all the elements stored into it in the subprogram. For example:

```
      SUBROUTINE CAT (A,M,N,B,C)
      REAL A(M), B(N), C(*)
      DO 10 I=1, M
10    C(I)=A(I)
      DO 20 I=1, N
20    C(I+M)=B(I)
      RETURN
      END
```

Subroutine CAT places the contents of array A followed by the contents of array B into array C. The dimension of C in the calling program must be greater than or equal to M+N.

Use of the asterisk form of the adjustable dimension prevents subscript checking for the array, so the user must be careful not to reference outside the array bounds. Use of this form is preferable to the common practice of declaring arrays to have dimension 1.

USING COMMON BLOCKS

Common blocks can be used to transfer values between a referencing program unit and a subprogram. Common blocks can reduce the number of storage units required for a program by enabling two or more subprograms to share some of the same storage units. The variables and arrays in a common block can be defined and referenced in all subprograms that contain a declaration of that common block. The names of the variables and arrays in the common block can be different for each subprogram. The association is by storage and not by name.

A reference to data in a common block is valid if the data is defined and is the same type as the type of the name used in the main program or subprogram. The exceptions to agreement between the type in common and the type of the reference are:

Either part of a complex entity can be referenced as real.

A Boolean entity can be referenced as integer.

In a subprogram, entities declared in a labeled common block can remain defined or become undefined at execution of an END or RETURN statement. If a labeled common block with the same name has been declared in a program unit that is directly or indirectly referencing the subprogram, the entities remain defined. Entities specified in a SAVE statement remain defined. Entities that are initially defined by DATA statements, and have neither been redefined nor become undefined, remain defined. Execution of a RETURN or END statement does not cause entities in blank common, or entities in any labeled common block that appears in the main program, to become undefined.

An example using common blocks in a subroutine is shown in figure 6-13.

```
      PROGRAM AVRG
      COMMON NUMBR(10), STORE
      REAL NUMBR, STORE
      READ *, NUMBR
      CALL SUM
      STORE = STORE/10
      PRINT *, 'AVERAGE= ', STORE
      END
C
      SUBROUTINE SUM
      COMMON A(10), B
      REAL A,B
      B = 0.
      DO 10 I= 1, 10
      B = B + A(I)
10    CONTINUE
      RETURN
      END
```

Figure 6-13. Using Common

The array NUMBR in program AVRG and the array A in subroutine SUM share the same locations in common. The values read into locations NUMBR(1) through NUMBR(10) are available to subroutine SUM.

REFERENCING A PROCEDURE

The CALL statement is used to reference a subroutine, the function name is used to reference a function, and the statement function name is used to reference a statement function. Multiple entry points can be used, and alternate return can be used for subroutines.

Subroutine Call

A subroutine subprogram is executed when a CALL statement is encountered in a program unit. The syntax for the the CALL statement is shown in figure 6-14.

CALL sub[([a[,a]. . .])]

sub Is the name of subroutine or dummy procedure.

a Is an actual argument that can be one of the following:

- An expression (except a character expression involving concatenation of a dummy argument with length (*))
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name
- An alternate return specifier of the form *s

s Is the statement label of an executable statement that appears in the same program unit as the CALL statement.

Figure 6-14. CALL Statement

The CALL statement can contain actual arguments and statement labels which must correspond in order, number, and type to those in the subroutine definition. An actual argument of type Boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type Boolean.

An actual argument in a subroutine call can be a dummy argument name that appears in the dummy argument list of the subprogram containing the subroutine call. An asterisk dummy argument cannot be used as an actual argument.

Function Reference

A function is executed when the name is referenced in an expression. A function must not directly or indirectly reference itself. The function reference can appear anywhere in an expression where an operand of the same type can be used. The syntax of a function reference is shown in figure 6-15.

fun([a[,a]. . .])

fun Is the name of the function or dummy procedure.

a Is an actual argument that can be one of the following:

- An expression (except a character expression involving concatenation of a dummy argument with length (*))
- An array name
- An intrinsic function name
- An external procedure name
- A dummy procedure name

Figure 6-15. Function Reference

The type of the function result is the type of the function name. The arguments must agree in order, number, and type with the corresponding dummy arguments. An actual argument of type Boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type Boolean.

Intrinsic and external functions can be referenced in any procedure subprogram. Intrinsic functions are predefined and are described in section 7.

Statement Function Reference

A statement function is evaluated when the name is referenced in an expression. The actual arguments are evaluated and converted to the type of the corresponding dummy argument; the resulting values are used in place of the corresponding dummy arguments in evaluation of the statement function expression. The definition of a statement function must not directly or indirectly reference itself. The statement function reference can appear anywhere in an expression where an operand of the same type can be used. The syntax of a statement function reference is shown in figure 6-16.

fun([a[,a]. . .])

fun Is the name of the statement function.

a Is an actual argument that must be an expression (except a character expression involving concatenation of a dummy argument with length (*)).

Figure 6-16. Statement Function Reference

The type of the statement function result is the type of the statement function name. The arguments must agree in order and number with the corresponding dummy arguments.

A statement function can only be referenced in the program unit where the statement function appears.

Return and Multiple Return

Each procedure subprogram ends with an END statement. Execution of the END statement terminates the procedure. The RETURN and END statements are often used together at the end of the procedure. The RETURN statement also terminates execution of the procedure. RETURN statements can be used wherever appropriate to terminate the procedure. A procedure that contains more than one RETURN statement (or a single RETURN statement that is separated from the END statement by other statements) has multiple returns. The RETURN statement is shown in figure 6-17.

RETURN[e]

e Is an arithmetic or Boolean expression. If e is not of type integer, the value INT(e) is used. The optional expression e can only be used in a subroutine subprogram.

Figure 6-17. RETURN Statement

For a simple return, the optional expression e is not used. An example is shown in figure 6-18.

```
      PROGRAM MAIN
      READ *, X,Y,Z
      CALL VAL(X,Y,Z)
      END
C
      SUBROUTINE VAL(A,B,C)
      IF(A .LT. 0.0) GO TO 12
      PRINT *, 'A IS POSITIVE'
      RETURN
12    CONTINUE
      PRINT *, 'A IS NEGATIVE'
      RETURN
      END
```

Figure 6-18. Multiple Return Example

Alternate Return

Execution of a RETURN or END statement returns control to the next executable statement in the referencing procedure. Control can be returned to a different place in the referencing procedure if the RETURN statement in the form RETURN e is used. A procedure that includes any RETURN e statements has alternate returns. Alternate returns can only be used in subroutine subprograms.

An alternate return returns control to a specified point other than the next executable statement following the procedure reference. The specified point is a statement label in the referencing procedure. The statement labels must be included in the actual argument list, each preceded by an asterisk. Control returns to the statement label determined by the integer value of the alternate return expression. If the value of the expression is less than one, or greater than the number of asterisks in the SUBROUTINE statement or ENTRY statement that is the current entry point, control returns to the statement following the CALL statement. For example, if a CALL statement contains five statement labels and if the alternate return expression evaluates to three, control returns to the third statement label specified in the actual argument in the alternate return list.

An example of an alternate return is shown in figure 6-19. RETURN 1 is a return to statement 20 in the calling program; RETURN 2 is a return to statement 30; RETURN 3 is a return to statement 40. The subroutine contains both the normal RETURN statement and alternate RETURN.

```
      PROGRAM MAIN
      READ *, A,B,C
      CALL XCOMP(A,B,C,*20,*30,*40)
20    CONTINUE
      PRINT *, 'RETURNED TO STMT 20'
      GO TO 10
30    CONTINUE
      PRINT *, 'RETURNED TO STMT 30'
      GO TO 10
40    CONTINUE
      PRINT *, 'RETURNED TO STMT 40'
10    END
C
      SUBROUTINE XCOMP(B1,B2,G,*,*,*)
      IF(B1*B2 - 4.159) 11,12,13
11    CONTINUE
      RETURN 2
12    CONTINUE
      RETURN 1
13    CONTINUE
      IF(B1 .GT. 32.) RETURN 3
      RETURN
      END
```

Figure 6-19. Alternate Return Example

FORTRAN 5 provides certain procedures that are of general utility or are difficult to express in FORTRAN. The supplied procedures are referenced in the same way as user-written procedures. The two classes of supplied procedures are intrinsic functions and utility subprograms.

INTRINSIC FUNCTIONS

An intrinsic function is a compiler-defined procedure that returns a single value. Intrinsic functions are referenced in the same way as user-written functions. If a variable, array, or statement function is defined with the same name as an intrinsic function, the name is a local name that no longer refers to the intrinsic function. If a function subprogram is written with the same name as an intrinsic function, use of the name references the intrinsic function, unless the name is declared as the name of an external function with the EXTERNAL statement described in section 2. Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name. If an intrinsic function is typed something other than the default for that function, the compiler does not honor the type statement and generates an error.

Certain intrinsic functions are generic. If a generic name and specific names exist, a generic name can be used in place of a specific name and is more flexible than a specific name. Except for type conversion generic functions, the type of the argument determines the type of the result.

For example, the generic function name LOG computes the natural logarithm of an argument. Its argument can be real, double precision, or complex. The type of the result is the same as the type of the argument.

Specific function names ALOG, DLOG, and CLOG also compute the natural logarithm. The specific function name ALOG computes the log of a real argument and returns a real result. Likewise, the specific name DLOG is for double precision arguments and results, and the specific name CLOG is for complex arguments and results.

Only a specific name can be used as an actual argument when passing the function name to a user-defined procedure or function. The intrinsic functions are listed in table 7-1. For specific names, the types of the arguments and results are shown.

The mathematical intrinsic functions are listed in table 7-2. The domains and ranges of the functions are shown in the table.

ABS

ABS(a) is a generic function that returns an absolute value. The result is integer, real, or double precision, depending on the argument type. For an integer, real, or double precision argument, the result is $|a|$. For a complex argument, the result is the square root of (ar^2+ai^2). The specific names are IABS, ABS, DABS, and CABS.

ACOS

ACOS(a) is a generic function that returns an arccosine. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ACOS and DACOS.

AIMAG

AIMAG(a) returns the imaginary part of a complex argument. The real result is ai, where the complex argument is (ar,ai).

AINT

AINT(a) is a generic function that returns an integer after truncation. The result is real. For a real or double precision argument, the result is 0 if $|a|<1$. If $|a|\geq 1$, the result is the largest integer with the same sign as argument a that does not exceed the magnitude of a. The specific names are AINT and DINT.

ALOG

ALOG(a) is a specific function that returns the natural logarithm of the argument. The argument is real and the result is real. The generic name is LOG.

ALOG10

ALOG10(a) is a specific function that returns the logarithm base 10 of the argument. The argument is real and the result is real. The generic name is LOG10.

AMAX0

AMAX0(a_1,a_2[,an]...) is a specific function that returns the value of the largest argument. The 2 through 500 arguments are integer, and the result is real. The generic name is MAX.

AMAX1

AMAX1(a_1,a_2[,a_n]...) is a specific function that returns the value of the largest argument. The 2 through 500 arguments are real, and the result is real. The generic name is MAX.

TABLE 7-1. INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type conversion	Conversion to integer, int(a)	1	INT	- INT IFIX IDINT -	Integer Real Real Double Complex	Integer Integer Integer Integer Integer
	Conversion to real	1	REAL	FLOAT REAL - - SNGL -	Integer Integer Real Complex Double Complex	Real Real Real Real Real Real
	Conversion to double	1	DBLE	- - - -	Integer Real Double Complex	Double Double Double Double
	Conversion to complex	1 or 2	CMPLX	- - - -	Integer Real Double Complex	Complex Complex Complex Complex
	Character conversion to integer	1	None	ICHAR	Character	Integer
	Integer conversion to character	1	None	CHAR	Integer	Character
	Conversion to Boolean	1	BOOL	-	Any type except logical	Boolean
Truncation	Defined as int(a)	1	AINT	AINT DINT	Real Double	Real Double
Nearest whole number	Defined as int(a + .5) if a is positive or zero; int(a - .5) if a is negative	1	ANINT	ANINT DNINT	Real Double	Real Double
Nearest integer	Defined as int(a + .5) if a is positive or zero; int(a - .5) if a is negative	1	NINT	NINT IDNINT	Real Double	Integer Integer

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute value	Defined as |a|; if a is complex, square root$_2$ of ((real a)2 + (imaginary a)2)	1	ABS	IABS ABS DABS CABS	Integer Real Double Complex	Integer Real Double Real
Remaindering	Defined as a_1-int(a_1/a_2)*a_2	2	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double
Transfer of sign	Defined as $\|a_1\|$ if a_2 is positive or zero; - $\|a_1\|$ if a_2 is negative	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Positive difference	Defined as a_1-a_2 if a_1 is greater than a_2; 0 if a_1 is not greater than a_2	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Double precision product	Defined as a_1*a_2	2	None	DPROD	Real	Double
Choosing largest value	Defined as max (a_1,a_2 [,a_n]...)	2 - 500	MAX	MAXO AMAX1 DMAX1	Integer Real Double	Integer Real Double
			None	AMAXO MAX1	Integer Real	Real Integer
Choosing smallest value	Defined as min (a_1,a_2 [,a_n]...)	2 - 500	MIN	MINO AMIN1 DMIN1	Integer Real Double	Integer Real Double
			None	AMINO MIN1	Integer Real	Real Integer
Length	Length of character string	1	None	LEN	Character	Integer
Index of a substring	Location of substring a_2 in string a_1	2	None	INDEX	Character	Integer

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Imaginary part of complex argument	Imaginary part of (ar,ai) = ai	1	None	AIMAG	Complex	Real
Conjugate of complex argument	Negation of imaginary part (ar,-ai)	1	None	CONJG	Complex	Complex
Square root	Square root of (a)	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Exponential	Defined as e**a	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural logarithm	Defined as $\log_e$ (a)	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common logarithm	Defined as $\log_{10}$ (a)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine	Defined as sin (a), where a is in radians	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
	Defined as sin (a), where a is in degrees	1	None	SIND	Real	Real
Cosine	Defined as cos (a), where a is in radians	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
	Defined as cos (a), where a is in degrees	1	None	COSD	Real	Real
Tangent	Defined as tan (a), where a is in radians	1	TAN	TAN DTAN	Real Double	Real Double
	Defined as tan (a), where a is in degrees	1	None	TAND	Real	Real
Arcsine	Defined as arcsin (a)	1	ASIN	ASIN DASIN	Real Double	Real Double

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Arccosine	Defined as arccos (a)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arctangent	Defined as arctan (a)	1	ATAN	ATAN DATAN	Real Double	Real Double
	Defined as arctan (a_1/a_2)	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic sine	Defined as sinh (a)	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic cosine	Defined as cosh (a)	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic tangent	Defined as tanh (a)	1	TANH	TANH DTANH	Real Double	Real Double
Hyperbolic arctangent	Defined as arctanh (a)	1	None	ATANH	Real	Real
Error function	Defined as erf (a)	1	None	ERF	Real	Real
Complementary error function	Defined as 1-erf (a)	1	None	ERFC	Real	Real
Lexically greater than or equal	True if a_1 follows a_2, or $a_1=a_2$, in ASCII collating sequence	2	None	LGE	Character	Logical
Lexically greater than	True if a_1 follows a_2 in ASCII collating sequence	2	None	LGT	Character	Logical
Lexically less than or equal	True if a_1 precedes a_2, or $a_1=a_2$, in ASCII collating sequence	2	None	LLE	Character	Logical
Lexically less than	True if a_1 precedes a_2 in ASCII collating sequence	2	None	LLT	Character	Logical

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Shift	Boolean result of a_1 shifted a_2 bit positions (see text for description)	2	None	SHIFT	Any type but character for a_1; integer for a_2	Boolean
Mask	Boolean result of left-justified 1 bits	1	None	MASK	Integer	Boolean
Random number generator	Random number in range (0,1)	0	None	RANF	None	Real
Location	Address of variable, array element, substring, subroutine, or external function (see figure 7-1 for format)	1	None	LOCF	Any type	Integer
Second	CPU time in seconds from start of job	0	None	SECOND	None	Real
Boolean product	Boolean result of .AND. operator	2 - 500	None	AND	Any type but character	Boolean
Boolean sum	Boolean result of .OR. operator	2 - 500	None	OR	Any type but character	Boolean
Exclusive OR	Boolean result of .XOR. operator	2 - 500	None	XOR	Any type but character	Boolean
Non-equivalence	Same as exclusive OR	2 - 500	None	NEQV	Any type but character	Boolean
Equivalence	Boolean result of .EQV. operator	2 - 500	None	EQV	Any type but character	Boolean
Complement	Boolean result of .NOT. operator	1	None	COMPL	Any type but character	Boolean

TABLE 7-2. SUMMARY OF MATHEMATICAL INTRINSIC FUNCTIONS

Function	Syntax	Type of Name	Domain	Definition	Range
arccosine (result in radians)	ACOS(y) ACOS(y) DACOS(y)	Generic Real Double	$\|y\| \leq 1$	$\cos^{-1}(y)$	$0 \leq ACOS(y) \leq \pi$
arcsin (result in radians)	ASIN(y) ASIN(y) DASIN(y)	Generic Real Double	$\|y\| \leq 1$	$\sin^{-1}(y)$	$-\pi/2 \leq ASIN(y) \leq \pi/2$
arctangent (result in radians)	ATAN(y) ATAN(y) DATAN(y)	Generic Real Double		$\tan^{-1}(y)$	$-\pi/2 \leq ATAN(y) \leq \pi/2$
arctangent (2 arguments, result in radians)	ATAN2(y,x) ATAN2(y,x) DATAN2(y,x)	Generic Real Double	$x<0, y<0$ $x=0, y<0$ $x>0$ $x=0, y>0$ $x<0, y\geq 0$ $x=0, y=0$	$-\pi+\tan^{-1}(y/x)$ $-\pi/2$ $\tan^{-1}(y/x)$ $\pi/2$ $\pi+\tan^{-1}(y,x)$ error	$-\pi < ATAN2(y,x) < -\pi/2$ $-\pi/2 < ATAN2(y,x) < \pi/2$ $\pi/2 < ATAN2(y,x) \leq \pi$
inverse hyperbolic tangent	ATANH(y) ATANH(y)	Generic Real	$\|y\| < 1$	$\tanh^{-1}(y)$	$-16.98 < ATANH(y) < 16.98$
trigonometric cosine (argument in radians)	COS(x) COS(x) DCOS(x)	Generic Real Double	$\|x\| < 2^{47}$	cos(x)	$-1 \leq COS(x) \leq 1$
	CCOS(x)	Complex	$\|x\| \leq \pi * 2^{46}$ $\|y\| \leq 741.66$	cos(x+iy)	
trigonometric cosine (argument in degrees)	COSD(x) COSD(x)	Generic Real	$\|x\| < 2^{47}$	cos(x)	$-1 \leq COSD(x) \leq 1$
hyperbolic cosine	COSH(x) COSH(x) DCOSH(x)	Generic Real Double	$\|x\| \leq 742.36$	cosh(x)	$1 \leq COSH(x)$
error function	ERF(x) ERF(x)	Generic Real		$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	$-1 \leq ERF(x) \leq 1$
complementary error function	ERFC(x) ERFC(x)	Generic Real	$x < 25.923$	$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	$0 < ERFC(x) \leq 2$
exporential	EXP(x) EXP(x) DEXP(x)	Generic Real Double	$-675.81 \leq x \leq 741.66$	e^x	$0 < EXP(x)$
	CEXP(x)	Complex	$-675.81 \leq x \leq 741.66$ $\|y\| \leq \pi * 2^{46}$	$e^{(x+iy)}$	
natural logarithm	LOG(x) ALOG(x) DLOG(x)	Generic Real Double	$x > 0$	$\log_e(x)$	
	CLOG(x)	Complex	$x^2+y^2 \neq 0$	$\log_e(x+iy)$	$-\pi <$ imaginary part $\leq \pi$

TABLE 7-2. SUMMARY OF MATHEMATICAL INTRINSIC FUNCTIONS (Contd)

Function	Syntax	Type of Name	Domain	Definition	Range
common logarithm (base 10)	LOG10(x) ALOG10(x) DLOG10(x)	Generic Real Double	$x>0$	$\log_{10}(x)$	
trigonometric sine (argument in radians)	SIN(x) SIN(x) DSIN(x)	Generic Real Double	$\|x\|<2^{47}$	sin(x)	$-1\leq SIN(x)\leq 1$
	CSIN(x)	Complex	$\|x\|\leq \pi*2^{46}$ $\|y\|\leq 741.66$	sin(x+iy)	
trigonometric sine (argument in degrees)	SIND(x) SIND(x)	Generic Real	$\|x\|<2^{47}$	sin(x)	$-1\leq SIND(x)\leq 1$
hyperbolic sine	SINH(x) SINH(x) DSINH(x)	Generic Real Double	$\|x\|\leq 742.36$	sinh(x)	
square root	SQRT(x) SQRT(x) DSQRT(x)	Generic Real Double	$x\geq 0$	$x^{0.5}$	$SQRT(x)\geq 0$
	CSQRT(x)	Complex	$x\geq 0, x<0$		value in right half plane
trigonometric tangent (argument in radians)	TAN(x) TAN(x) DTAN(x)	Generic Real Double	$\|x\|\leq \pi*2^{46}$	tan(x)	
trigonometric tangent (argument in degrees)	TAND(x) TAND(x)	Generic Real	$\|x\|<2^{47}$	tan(x)	
hyperbolic tangent	TANH(x) TANH(x)	Generic Real		tanh(x)	$-1\leq TANH(x)\leq 1$

AMIN0

AMIN0(a_1,a_2[,a_n]...) is a specific function that returns the value of the smallest argument. The 2 through 500 arguments are integer, and the result is real. The generic name is MIN.

AMIN1

AMIN1(a_1,a_2[,a_n]...) is a specific function that returns the value of the smallest argument. The 2 through 500 arguments are real, and the result is real. The generic name is MIN.

AMOD

AMOD(a_1,a_2) is a specific function that returns a_1 modulus a_2. The arguments are real and the result is real. If a_2 is zero, results are undefined. The generic name is MOD.

AND

AND(a_1,a_2[,a_n]...) is a specific function that returns a Boolean product. The 2 through 500 arguments can be any type but character, and the result is Boolean.

ANINT

ANINT(a) is a generic function that returns the nearest whole number. The result is real or double precision, depending on the argument type. The specific names are ANINT and DNINT.

ASIN

ASIN(a) is a generic function that returns an arcsine. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ASIN and DASIN.

ATAN

ATAN(a) is a generic function that returns an arctangent. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ATAN and DATAN.

ATANH

ATANH(a) is a specific function that returns a hyperbolic arctangent. The argument and result are real. See table 7-2.

ATAN2

ATAN2(a_1,a_2) is a generic function that returns an arctangent. The result is expressed in radians. The result is real or double precision, depending on the type of the arguments. The arguments must not both be zero. See table 7-2. The specific names are ATAN2 and DATAN2.

BOOL

BOOL(a) is a generic function that performs type conversion and returns a Boolean value. The argument can be integer, real, double precision, complex, character, or Boolean. For an integer, real, or Boolean argument, the result is the bit string constituting the data. For a double precision or complex argument, the result is the bit string after conversion of the argument to real with REAL(a). For a character argument, the result is the value of the Hollerith constant nHf, where n is the length and f is the character value; if n is greater than 10, the rightmost characters are truncated. There are no specific names.

CABS

CABS(a) is a specific function that returns a real result from a complex argument. The generic name is ABS.

CCOS

CCOS(a) is a specific function that returns a complex result from a complex argument. The generic name is COS.

CEXP

CEXP(a) is a specific function that returns a complex result from a complex argument. The generic name is EXP.

CHAR

CHAR(a) returns the character value of an integer argument. The value returned depends on the collating sequence used. If the ASCII collating sequence is used, the argument must be in the range $0 \le a \le 63$; the first character in the collating sequence corresponds to value 0, the second character to value 1, the third to value 2, and so forth. The result is the selection of a single character from the collating sequence. If, in a user-specified collating sequence, more than one character has weight a, the character returned can be any of them.

CLOG

CLOG(a) is a specific function that returns a complex result from a complex argument. The generic name is LOG.

CMPLX

CMPLX(a) or CMPLX(a_1,a_2) is a generic function that performs type conversion and returns a complex value. CMPLX can have one or two arguments. A single argument can be integer, real, double precision, or complex. If two arguments are used, the arguments must be of the same type and must both be integer, real, or double precision. For a single integer, real, or double precision argument, the result is complex, with the argument used as the real part and the imaginary part zero. For a single complex argument, the result is the same as the argument. For two arguments a_1 and a_2, the result is complex, with argument a_1 used as the real part and argument a_2 used as the imaginary part. There are no specific names.

COMPL

COMPL(a) returns a complemented value. The argument can be any type except character, and the result is Boolean. If the argument is not Boolean, the argument is converted with BOOL(a). The result is the result of the logical operator .NOT. on a Boolean value.

CONJG

CONJG(a) returns a conjugate of a complex argument. The result is complex. For a complex argument (ar,ai), the result is (ar,-ai) with the imaginary part negated.

COS

COS(a) is a generic function that returns a cosine. The argument is assumed to be in radians. The result is real,

double precision, or complex, depending on the argument type. See table 7-2. The specific names are COS, CCOS, and DCOS.

COSD

COSD(a) returns a cosine. The argument is assumed to be in degrees. The argument and result are real. See table 7-2.

COSH

COSH(a) is a generic function that returns a hyperbolic cosine. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are COSH and DCOSH.

CSIN

CSIN(a) is a specific function that returns the sine of the argument. The argument and result are complex. The generic name is SIN.

CSQRT

CSQRT(a) is a specific function that returns a complex result from a complex argument. The generic name is SQRT.

DABS

DABS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ABS.

DACOS

DACOS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ACOS.

DASIN

DASIN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ASIN.

DATAN

DATAN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ATAN.

DATAN2

DATAN2(a_1,a_2) is a specific function that returns a double precision result from a double precision argument. The generic name is ATAN2.

DBLE

DBLE(a) is a generic function that performs type conversion and returns a double precision result. The argument can be integer, real, double precision, or complex. For an integer or real argument, the result has as much precision as the double precision field can contain. For a double precision argument, the result is the argument. For a complex argument, the real part is used, and the result has as much precision as the double precision field can contain. There are no specific names.

DCOS

DCOS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is COS.

DCOSH

DCOSH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is COSH.

DDIM

DDIM(a_1,a_2) is a specific function that returns a double precision result from double precision arguments. It returns the value of a_1-a_2; if $a_1 < a_2$, it returns zero. The generic name is DIM.

DEXP

DEXP(a) is a specific function that returns a double precision result from a double precision argument. The generic name is EXP.

DIM

DIM(a_1,a_2) is a generic function that returns a positive difference. The result is integer, real, or double precision, depending on the argument type. Both arguments must be the same type. The result is a_1-a_2 if $a_1 > a_2$, and the result is 0 if $a_1 \leq a_2$. The specific names are DIM, IDIM, DDIM.

DINT

DINT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is AINT.

DLOG

DLOG(a) is a specific function that returns a double precision result from a double precision argument. The generic name is LOG.

DLOG10

DLOG10(a) is a specific function that returns a double precision result from a double precision argument. The generic name is LOG10.

DMAX1

DMAX1(a_1,a_2[,a_n]...) is a specific function that returns a double precision result from 2 through 500 double precision arguments. The generic name is MAX.

DMIN1

DMIN1(a_1,a_2[,a_n]...) is a specific function that returns a double precision result from 2 through 500 double precision arguments. The generic name is MIN.

DMOD

DMOD(a_1,a_2) is a specific function that returns a double precision result from two double precision arguments. If a_2 is zero, results are undefined. The generic name is MOD.

DNINT

DNINT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ANINT.

DPROD

DPROD(a_1,a_2) returns a double precision product. The arguments are real, and the result is double precision. The result is a_1*a_2.

DSIGN

DSIGN(a_1,a_2) is a specific function that returns a double precision result from two double precision arguments. The generic name is SIGN.

DSIN

DSIN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SIN.

DSINH

DSINH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SINH.

DSQRT

DSQRT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SQRT.

DTAN

DTAN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is TAN.

DTANH

DTANH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is TANH.

EQV

EQV(a_1,a_2[,a_n]...) returns an equivalence result. The 2 through 500 arguments can be any type except character, and the result is Boolean. The result is the same as for the Boolean .EQV. operator.

ERF

ERF(a) returns an error function result. The argument and result are real. See table 7-2.

ERFC

ERFC(a) returns a complementary error function result. The argument and result are real. The result is 1-erf(a). See table 7-2.

EXP

EXP(a) is a generic function that returns an exponential. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are EXP, DEXP, and CEXP.

FLOAT

FLOAT(a) is a specific function that returns a real result from an integer argument. The generic name is REAL.

IABS

IABS(a) is a specific function that returns an integer result from an integer argument. The generic name is ABS.

ICHAR

ICHAR(a) returns an integer value from a character argument. The value returned depends on the collating weight of the character in the collating sequence used. For the ASCII collating sequence, the first character in the collating sequence is at position 0, the second character at position 1, the third at position 2, and so forth. For a user-specified collating sequence, two or more characters can have the same value. The argument is a character value with a length of one character, and the value returned is the integer position of that character in the collating sequence.

IDIM

IDIM(a_1,a_2) is a specific function that returns an integer result from integer arguments. It returns the value of a_1-a_2; if a_1< a_2, it returns zero. The generic name is DIM.

IDINT

IDINT(a) is a specific function that returns an integer result from a double precision argument. The generic name is INT.

IDNINT

IDNINT(a) is a specific function that returns an integer result from a double precision argument. The generic name is NINT.

IFIX

IFIX(a) is a specific function that returns an integer result from a real argument. The generic name is INT.

INDEX

INDEX(a_1,a_2) returns the location of a substring within a string. Both arguments must be character string arguments. If string a_2 occurs as a substring within string a_1, the result is an integer indicating the starting position of the substring a_2 within a_1. If a_2 does not occur as a substring within a_1, the result is 0. If a_2 occurs as a substring more than once within a_1, only the starting position of the first occurrence is returned.

INT

INT(a) is a generic function that performs type conversion to integer. The result is integer, and the argument can be integer, real, double precision, or complex. For an integer argument, the result is the argument. For a real or double precision argument where $|a|<1$, the result is 0. Where $|a| \geq 1$, the result is the largest integer with the same sign as argument a that does not exceed the magnitude of a. For a complex argument, the real part is used, and the result is the same as for a real argument. The specific names are INT, IFIX and IDINT.

ISIGN

ISIGN(a_1,a_2) is a specific function that returns an integer result from two integer arguments. The generic name is SIGN.

LEN

LEN(a) returns the length of a character string. The argument is a character string, and the result is an integer indicating the length of the string.

LGE

LGE(a_1,a_2) returns a result indicating lexically greater than or equal to. The arguments are character strings. The result is true only if a_1 follows a_2 or a_1 is equal to a_2 in the ASCII collating sequence (shown in appendix A).

LGT

LGT(a_1,a_2) returns a result indicating lexically greater than. The arguments are character strings. The result is true only if a_1 follows a_2 in the ASCII collating sequence (shown in appendix A).

LLE

LLE(a_1,a_2) returns a result indicating lexically less than or equal to. The arguments are character strings. The result is true only if a_1 precedes a_2 or a_1 is equal to a_2 in the ASCII collating sequence (shown in appendix A).

LLT

LLT(a_1,a_2) returns a result indicating lexically less than. The arguments are character strings. The result is true only if a_1 precedes a_2 in the ASCII collating sequence (shown in appendix A).

LOCF

LOCF(a) returns a location, that is, an address. The argument can be a variable, array element, substring, subroutine, or external function. For a noncharacter argument, the result is integer. For a character argument, LOCF returns the argument address, length, and beginning character position. A flag indicating extended memory or central memory residence is always returned. The format of the result for character arguments is shown in figure 7-1.

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.

LOG

LOG(a) is a generic function that returns a natural logarithm. The result is real, double precison, or complex, depending on the argument type. See table 7-2. For a complex argument (ar,ai), the range of the imaginary part of the result is $-\pi < ai \leq \pi$. The imaginary part of the result is only zero when ar>0 and ai=0. The specific names are ALOG, DLOG, and CLOG.

LOG10

LOG10(a) is a generic function that returns a common logarithm. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ALOG10 and DLOG10.

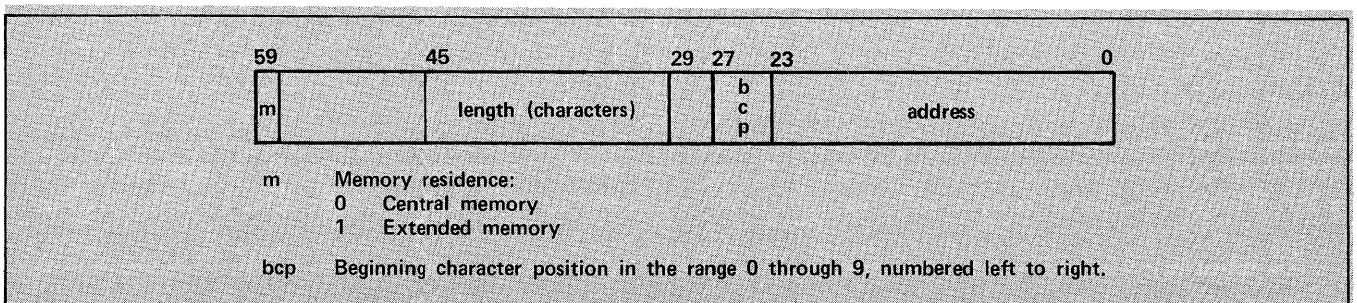


Figure 7-1. LOCF Result for Character Argument

MASK

MASK(a) returns a Boolean result. The argument is integer or Boolean in the range 0 through 60. The result is a word of left-justified one bits followed by (60-a) zero bits (a<0→a=0).

MAX

MAX(a_1,a_2[,a_n]...) is a generic function that returns the largest value. The result is integer, real, or double precision, depending on the type of the 2 through 500 arguments. The specific names are MAX0, AMAX1, and DMAX1.

MAX0

MAX0(a_1,a_2[,a_n]...) is a specific function that returns as an integer result the largest value from 2 through 500 integer arguments. The generic name is MAX.

MAX1

MAX1(a_1,a_2[,a_n]...) is a specific function that returns as an integer result the largest value from the 2 through 500 real arguments.

MIN

MIN(a_1,a_2[,a_n]...) is a generic function that returns the smallest value from the 2 through 500 arguments. The result is integer, real, or double precision, depending on the type of arguments. The specific names are MIN0, AMIN1, and DMIN1.

MIN0

MIN0(a_1,a_2[,a_n]...) is a specific function that returns as an integer result the smallest value from the 2 through 500 integer arguments. The generic name is MIN.

MIN1

MIN1(a_1,a_2[,a_n]...) is a specific function that returns as an integer result the smallest value from the 2 through 500 real arguments.

MOD

MOD(a_1,a_2) is a generic function that returns the remainder of a_1 divided by a_2. The result is integer, real, or double precision, depending on the argument type. The result is a_1-(int(a_1/a_2)*a_2). If a is zero, results are undefined. The specific names are MOD, AMOD, and DMOD.

NEQV

NEQV(a_1,a_2[,a_n],...) returns a nonequivalence result. The result is Boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the Boolean exclusive or (.XOR.) operator.

NINT

NINT(a) is a generic function that returns the nearest integer. The result is integer, and the argument can be real or double precision. For a real or double precision argument where a is zero or positive, the result is (int(a+.5)). For an argument where a is negative, the result is (int(a-.5)). The specific names are NINT and IDNINT.

OR

OR(a_1,a_2[,a_n]...) returns a Boolean sum. The result is Boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the Boolean .OR. operator.

RANF

RANF returns a random number. Since there is no argument, RANF is referenced as RANF(). The result is real and is in the range 0<result<1.

REAL

REAL(a) is a generic function that performs type conversion and returns a real result. The argument can be integer, real, double precision, or complex. For a complex argument (ar,ai), the result is real(ar). The specific names are REAL, FLOAT, and SNGL.

SECOND

SECOND returns the CPU time in seconds since the beginning of the job. Since there is no argument, SECOND is referenced as SECOND(). The result is real.

SHIFT

SHIFT(a_1,a_2) returns a shifted result. The argument a_1 is any type but character, and argument a_2 is integer or Boolean. The Boolean result is a_1 shifted a_2 bit positions. The shift is left circular if a_2 is positive; right with sign extension and end off if a_2 is negative. a_2 is in the range -60 and +60.

SIGN

SIGN(a_1,a_2) is a generic function that returns a value after a transfer of sign. The result is integer, real, or double precision, depending on the argument type. The result is $|a_1|$ if a_2 is zero or positive. The result is $-|a_1|$ if a_2 is negative. The specific names are SIGN, ISIGN, and DSIGN.

SIN

SIN(a) is a generic function that returns a sine. The argument is assumed to be in radians. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are SIN, DSIN, and CSIN.

SIND

SIND(a) returns a sine. The argument is assumed to be in degrees. The argument and result are real. See table 7-2.

SINH

SINH(a) is a generic function that returns a hyperbolic sine. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are SINH and DSINH.

SNGL

SNGL(a) is a specific function that returns a real result from a double precision argument. The generic name is REAL.

SQRT

SQRT(a) is a generic function that returns a principal square root. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are SQRT, DSQRT, and CSQRT.

TAN

TAN(a) is a generic function that returns a tangent. The argument is assumed to be in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are TAN and DTAN.

TAND

TAND(a) returns a tangent. The argument is assumed to be in degrees. The argument and result are real. See table 7-2.

TANH

TANH(a) is a generic function that returns a hyperbolic tangent. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are TANH and DTANH.

XOR

XOR(a_1,a_2[,a_n]...) returns an exclusive OR value. The 2 through 500 arguments can be any type but character, and the result is Boolean. The result is the same as for the Boolean exclusive or (.XOR.) operator.

MISCELLANEOUS UTILITY SUBPROGRAMS

The utility subprograms described below are supplied by the system and are always called by name. A user-supplied subprogram with the same name as a library subprogram overrides the library subprogram. Other utility routines are described later in this section.

GETPARM

The GETPARM call shown in figure 7-2, is used to access user parameters that have been declared on the execution control statement (described in section 11).

Each call to GETPARM returns the next parameter from the control statement.

```
CALL GETPARM (c1,c2,i)

    c1    Character variable substring or array
          element to receive the parameter name.

    c2    Character variable substring or array
          element to receive the parameter value.

    i     Integer return code. Possible values are:

            -1   No user parameters remain on
                 execution control statement (c1,
                 c2 undefined).

             0   Normal return (values stored in
                 c1 and c2).

             1   Parameter name only specified on
                 execution control statement (c2
                 contains blanks).
```

Figure 7-2. GETPARM Call

RANSET

The RANSET call shown in figure 7-3 initializes the seed of RANF. Bit 0 will be set to 1 (forced odd), and bits 59 through 48 will be set to 1717_8.

```
CALL RANSET(n)

    n     Is a 1-word bit pattern used
          to generate the seed for RANF.
```

Figure 7-3. RANSET Call

RANGET

The RANGET call shown in figure 7-4 obtains the current seed of RANF between 0 and 1. The value returned to n is not necessarily normalized. The value returned can be passed to RANSET at a later time to regenerate the same sequence of random numbers.

```
CALL RANGET(n)

    n     Is the symbolic name to receive the
          random number seed.
```

Figure 7-4. RANGET Call

OPERATING SYSTEM INTERFACE ROUTINES

Operating system interface routines include a variety of subprograms. Each is described separately.

NOTE

Refer to appendix G for recommendations on the use of these routines.

DATE

The DATE function shown in figure 7-5 returns the current date as the value of the function in the form Δmm/dd/yy (NOS/BE and SCOPE 2) or Δyy/mm/dd (NOS), where mm is the number of the month, dd is the day within the month, and yy is the year. The format might be different at a particular installation. The value returned is type character with a length of 10. DATE must be declared type character*10 in the calling program.

```
DATE()
```

Figure 7-5. DATE Function

JDATE

The JDATE function shown in figure 7-6 returns the current date as the value of the function in the form yyddd, where yy is the year and ddd is the number of the day within the year. The value returned is type character with a length of 5. JDATE must be declared type character*5 in the calling program. (JDATE is not available on SCOPE 2.)

```
JDATE()
```

Figure 7-6. JDATE Function

TIME or CLOCK

The TIME function shown in figure 7-7 or CLOCK function shown in figure 7-8 returns the current reading of the system clock as the value of the function in the form Δhh.mm.ss., where hh is hours from 0 to 23, mm is minutes, and ss is seconds. The value returned is type character with a length of 10; the first character in the value returned is system dependent and is not always a blank. TIME and CLOCK must be declared type character*10 in the calling program.

```
TIME()
```

Figure 7-7. TIME Function

```
CLOCK()
```

Figure 7-8. CLOCK Function

DISPLA

The DISPLA call shown in figure 7-9 places a name and a value in the dayfile. The character constant h cannot be more than 50 characters; k is a real or integer variable or expression and is displayed as an integer or real value. Characters with display codes greater than 57_8 are listed in the dayfile, but are replaced by blanks when displayed at the operator's console. If the first character is $, the message will flash at the console (on NOS/BE).

```
CALL DISPLA(h,k)
    h    Is a character expression to be displayed.
    k    Is a real or integer variable or expression
         whose value is to be displayed.
```

Figure 7-9. DISPLA Call

REMARK

The REMARK call shown in figure 7-10 places a message in the dayfile. The maximum message length is 80 characters displayed 40 characters per line under NOS/BE, 90 characters displayed on one line under SCOPE 2, and one line of 30 characters under NOS. A message exceeding the maximum length is truncated. Characters with display codes greater than 57_8 are listed in the dayfile, but are replaced by blanks when displayed at the operator's console. If the first character is $, the message will flash at the console (on NOS/BE).

```
CALL REMARK(h)
    h    Is a character expression.
```

Figure 7-10. REMARK Call

SSWTCH

The SSWTCH call shown in figure 7-11 tests sense switches. If sense switch i is on, j is set to 1; if sense switch i is off, j is set to 2. The value i is 1 to 6. If i is out of range, an informative diagnostic is printed, and j is set to 2. The sense switches are set or reset by the computer operator or by the control statements SWITCH (NOS, NOS/BE, and SCOPE 2), ONSW (NOS only), and OFFSW (NOS only).

```
CALL SSWTCH(i,j)
    i    Is a sense switch number.
    j    Is an integer return variable.
```

Figure 7-11. SSWTCH Call

EXIT

The EXIT call shown in figure 7-12 terminates program execution and returns control to the operating system.

NOTE

Use of the STOP statement is preferable to CALL EXIT.

```
CALL EXIT
```

Figure 7-12. EXIT Call

CHEKPTX

A checkpoint dump of the files specified is taken. If n is zero, all files are checkpointed. If n is nonzero, the files specified by filelist are checkpointed. Figure 7-13 shows the format for CHEKPTX and figure 7-14 illustrates CHEKPTX used with an array containing three file names. The first element of the array declares how many files are to be checkpointed. The next three elements contain file names and how they are to be processed. For additional information about CHEKPTX, see the appropriate operating system reference manual.

RECOVR

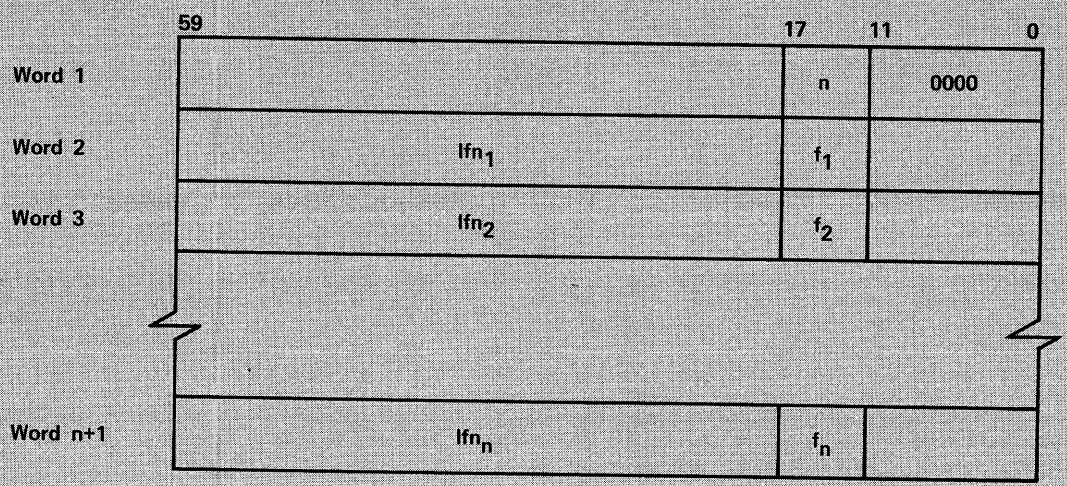
The RECOVR subroutine allows a user program to gain control at the time that normal or abnormal job termination procedures would otherwise occur. RECOVR is not available on SCOPE 2. Figure 7-15 shows the format for RECOVR. Initialization of RECOVR at the beginning of a program establishes the conditions under which control is to be regained and specifies the address of user recovery code. If the stated condition occurs during program execution, control returns to the user code. If necessary, the system increases the CP time limit, input/output time limit, or mass storage limit to provide an installation-defined minimum of time and mass storage for RECOVR processing. No limit is increased more than once in a job. RECOVR can be called more than once during program initialization to reference different user recovery subroutines. These calls to RECOVR can use different combinations of conditions for the same or different user recovery subroutines.

```
      DIMENSION IFILES(4)
      IFILES(1) = O"30000"
      IFILES(2) = L"TAPE1" .OR. O"10000"
      IFILES(3) = L"TAPE2" .OR. O"30000"
      IFILES(4) = L"TAPE3"
      .
      .
      .
      CALL CHEKPTX(IFILES, 3)
```

Figure 7-14. CHEKPTX Example

CALL CHEKPTX(filelist,n)

filelist Is an array in the following format:



n Is the number of files in following list, to a maximum of 42.

lfn$_i$ Is the name (in left-justified display code) of user storage files to be processed.

f$_n$ Is a number indicating specific manner in which lfn is to be processed (does not apply to SCOPE 2).

- 0 Mass storage file is copied from beginning-of-information to its position at checkpoint time, and only that portion will be available at restart. The file is positioned at the latter point.
- 1 Mass storage file is copied from its position at check time to end-of-information, and only that portion will be available at restart. The file is positioned at the former point.
- 2 Mass storage file is copied from beginning-of-information to end-of-information; the entire file will be available at restart time. The file is positioned at the point at which the checkpoint was taken.
- 3 The last operation on the file determines how the mass storage file is copied.

Figure 7-13. CHEKPTX Call

CALL RECOVR(name,flags,checksum)

name	Is the name of subroutine to be executed if flagged conditions occur (must be specified in an EXTERNAL statement).
flags	Is an octal value for conditions under which recovery code is to be executed, as outlined below. Conditions can be combined as desired, with octal values up to 177 allowed.
	001 Arithmetic mode error.
	002 PP call or auto-recall error.
	004 Time or storage limit exceeded.
	010 Operator drop, kill, or rerun.
	020 System abort.
	040 CP abort.
	100 Normal termination.
checksum	Is the last word address of recovery code to be checksummed; 0 if no checksum is desired.

Figure 7-15. RECOVR Call

No more than five routines can be specified by RECOVR in one program. If an error occurs and more than one routine has been established for that error, the routines are called successively, with the routine most recently specified called first. The second specification of a subroutine overrides its previous parameters. This override can be used to remove a subroutine from the RECOVR list by passing a mask of zero.

A checksum of the user recovery code can be requested during initialization. If flagged conditions subsequently occur, RECOVR again checksums the code before returning control to it. This gives some assurance of user code integrity before it is executed. If the checksum parameter is zero, no checksum is done.

If one of the user's selected error conditions occurs, RECOVR gains control, performs internal tasks, and then transfers control to the user's recovery subroutines. The following three arguments are passed to the user's recovery subroutine:

1. A 17-word integer array. The first 16 words are an image of the exchange package; the 17th word is the contents of RA+1.

2. A flag that, upon return, determines the type of program termination. If the user's recovery subroutine sets the flag to nonzero, the job terminates normally, as if no errors had occurred. If the flag remains zero, the job continues as if RECOVR had not been called, that is, the original system error code is restored and processed.

3. An array, starting at RA+1, that allows a subroutine to access all of the user's field length.

If the recovery subroutine was called because of normal termination, the subroutine, before returning, should flush the buffers of all output files. Buffers can be flushed by an ENDFILE, REWIND, or CLOSE statement.

In an overlay-structured program, calls to RECOVR as well as the user recovery subprograms should be in the (0,0) overlay.

For additional information about RECOVR, see the appropriate operating system reference manual.

INPUT/OUTPUT STATUS CHECKING

Status checking for input/output statements such as READ and WRITE should be done with the optional specifiers (section 5), but can also be done with the functions UNIT, EOF, and IOCHEC. UNIT and EOF return an end-of-file status for any of the following conditions:

End-of-section (for file INPUT only)

End-of-partition

Nondeleted W format flag record

Embedded tape mark

Terminating double tape mark

Terminating end-of-file label

Embedded zero length level 17 block

The functions UNIT and IOCHEC return a parity error indication for every record within or spanning a block containing a parity error; such an indication, however, does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the CYBER Record Manager input/output routines.

On SCOPE 2 only, parity status can be checked on write operations that access mass storage files when the write check option has been specified on the REQUEST statement for the file. (See the SCOPE 2 reference manual.) Write parity errors for other types of devices (such as staged/on-line tape) are detected by the operating system, and a message to this effect is written in the dayfile.

UNIT

The UNIT function shown in figure 7-16 is used to check the status of a BUFFER IN or BUFFER OUT operation for an end-of-file or parity error condition on logical unit u. When UNIT is referenced, the user program does not regain control until input/output operations on the unit are complete. The function returns the following values:

-1.	Unit ready, no end-of-file or parity error encountered on the previous operation.
+0.	Unit ready, end-of-file encountered on the previous operation.
+1.	Unit ready, parity error encountered on the previous operation.

UNIT(u)	
u	Is the unit specifier.

Figure 7-16. UNIT Function

Example:

```
IF (UNIT(5)) 12,14,16
```

Control transfers to the statement labeled 12, 14, or 16 if the value returned was -1., 0., or +1., respectively.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. UNIT should only be called for a file processed by BUFFER statements.

The UNIT function is of type real.

EOF

The EOF function shown in figure 7-17 is used to test for an end-of-file condition on unit u following a formatted, list directed, NAMELIST, or unformatted sequential read. Zero is returned if no end-of-file is encountered, or a nonzero value if end-of-file is encountered. If an end-of-file is encountered, EOF clears the indicator before returning control. For example:

```
IF (EOF(5) .NE. 0) GO TO 20
```

transfers control to statement 20 if an end-of-file is encountered on unit 5.

EOF(u)	
u	Is the unit specifier.

Figure 7-17. EOF Function

If the IOSTAT= or END= specifier is not used in the READ statement that reads the end of file, the program will be terminated before EOF can be used to check for end of file.

The EOF function returns a zero value following read or write operations on random access files (files accessed by READMS/WRITMS), and also following write operations on all types of files, regardless of whether an end-of-file condition has been detected; therefore, the EOF function should not be used in those circumstances.

The EOF function does not replace the END= or the IOSTAT= parameters in the READ statement.

The EOF function is of type real.

IOCHEC

The IOCHEC function shown in figure 7-18 tests for a parity error on unit u following a formatted, list directed, NAMELIST, or unformatted read. The value zero is returned if no error has been detected. If a parity error occurs, IOCHEC clears the parity indicator before returning. Parity errors are handled in this way regardless of the type of the external device.

IOCHEC(u)	
u	Is the unit specifier.

Figure 7-18. IOCHEC Function

Example:

```
J=IOCHEC(6)
IF(J .NE. 0) GO TO 25
```

Zero value is returned to J if no parity error occurs and nonzero if an error does occur; control transfers to the statement labeled 25 if an error occurs.

OTHER INPUT/OUTPUT SUBPROGRAMS

Other input/output subprograms are also supplied. Each is described separately.

LENGTH

The LENGTH function or LENGTHX subroutine shown in figure 7-19 returns information regarding the previous BUFFER IN or READMS call of the file designated by u. The argument nw of the value of LENGTHX is set to the number of words read. The argument ubc is set to the number of unused bits in the last word of the transfer. The arguments nw, ubc, and value returned are type integer.

LENGTH(u) CALL LENGTHX(u,nw,ubc)	
u	Is the unit specifier.
nw	Is the argument set to the number of words read.
ubc	Is the argument set to the number of unused bits in the last word of the transfer.

Figure 7-19. LENGTH Subprogram

After an unformatted BUFFER IN on a 9-track S or L tape, the unused bit count parameter of LENGTHX is rounded down so as to indicate a whole number of 6-bit characters. For example, a BUFFER IN of a 23-character record returns a length of four words with an unused bit count of 54, even though the actual unused bit count is 56.

If an odd number of words is written to a 9-track S or L tape by an unformatted BUFFER OUT, the record on the tape contains four additional zero bits at the right so as to be a whole number of 8-bit characters. If such a record is subsequently read by BUFFER IN, the length indication in LENGTH or LENGTHX is one word greater than the number of words originally written.

For a file accessed by buffer statements, LENGTH or LENGTHX should be called only after a call to UNIT ensures that input/output activity is complete; otherwise, file integrity might be endangered.

LABEL

The LABEL call shown in figure 7-20 passes label information to the operating system. (LABEL is recognized, but ignored on SCOPE 2.) The value labinfo is the name of a 4-word array containing label information in the format given for words 9 through 12 of the file environment table (FET) in the operating system reference manual. Before the CALL LABEL statement can be used, the control statement that requests the tape for the job must specify that the tape has labels.

CALL LABEL(u,labinfo)

u	**Is the unit specifier.**
labinfo	**Specifies the array for label information.**

Figure 7-20. LABEL Call

On input, the specified file's label is compared with the indicated information in labinfo (unless it was so checked when an earlier LABEL control statement was executed). If any of the relevant fields were filled with binary zeros by CALL LABEL, these fields are set to the values contained in the label read. If there is a mismatch between the label read and any field not zero-filled, a request is sent to the operator for a GO or DROP response.

On output, the appropriate information from labinfo is written as a label at the beginning of the specified file. If any of the relevant fields contain only binary zeros, the corresponding label field will be set to an appropriate default value.

CALL LABEL should not be used with files accessed with CYBER Record Manager interface routines.

MOVLEV

The MOVLEV call shown in figure 7-21 transfers n consecutive words of data between a and b. MOVLEV can be used to transfer blocks of data between ECS or LCM and central memory. The arguments a and b are variables or array elements; n is an integer value. The argument a is the starting address of the data to be moved, and b is the starting address of the receiving location.

CALL MOVLEV(a,b,n)

a	**Specifies a variable that is the starting address of the data to be moved.**
b	**Specifies a variable that is the starting address of the receiving location.**
n	**Specifies the number of words to be moved.**

Figure 7-21. MOVLEV Call

No conversion is done by MOVLEV. If data from a real variable is moved to a type integer receiving field, the data remains real.

Example:

```
CALL MOVLEV (A, I, 1000)
```

After the move, I does not contain the integer equivalent of A.

Example:

```
DOUBLE PRECISION D1(500), D2(500)
CALL MOVLEV (D1, D2, 1000)
```

Since D1 is defined as double precision, n should be set to 1000 to move the entire D1 array.

NOTE

MOVLEV of character data is not allowed.

MOVLCH

The MOVLCH call shown in figure 7-21.1 transfers n consecutive characters between a and b. MOVLCH can be used to transfer characters between ECS or LCM and central memory. MOVLCH can also transfer characters between two blocks of storage resident in the same level of memory, LCM to LCM or ECS to ECS. Both arguments a and b must be of type character; a diagnostic is issued if one, or both, is of any other type.

Example:

```
CHARACTER * 123,CH1(10),CH2(5)
CALL MOVLCH (CH1(8),CH2(3),369)
```

CALL MOVLCH (a, b, n)

a	**Specifies a variable, array element, or substring that represents the starting location of the character string to be moved.**
b	**Specifies a variable, array element, or substring that represents the starting location of the receiving area.**
c	**Specifies the number of characters to move.**

Figure 7-21.1. MOVLCH Call

The last three elements of character array CH1 are to be moved to the last three elements of character array CH2. Each element is 123 characters long; therefore, the total number of characters to move is 3 * 123 = 369.

CONNEC

The subroutine CONNEC connects files to the terminal. Figure 7-22 shows the format for CONNEC.

If a program to be run interactively calls for input/output operations through the user's remote terminal, all files to be accessed through the terminal must be formally associated with the terminal at the time of execution.

CALL CONNEC(u,cs)

u Is the unit designator.

cs Is an optional character set designator (applicable to NOS/BE only); cs is an integer value 0, 1, or 2.

0 Display code (default)
1 ASCII-95
2 ASCII-256

Figure 7-22. CONNEC Call

In particular, the file INPUT must be connected to the terminal if data is to be entered there and a numbered logical unit is not designated in the READ statement. The file OUTPUT must be connected to the terminal if execution diagnostics are to be displayed or printed at the terminal, or if data is to be displayed or printed there and a numbered unit is not designated in the WRITE statement. These files are automatically connected to the terminal when the program is executed under NOS/BE, using the RUN command of the EDITOR utility of INTERCOM, or under NOS.

Under all operating systems, the user can connect any file from within the program by using the CALL CONNEC statement. Under INTERCOM, any file can be connected to the terminal by the CONNECT command. More information about INTERCOM is in the INTERCOM reference manual and the INTERCOM Interactive Guide for Users of FORTRAN. Under HELLO7, for SCOPE 2, any file can be connected by providing a FILE control statement specifying CNF = YES. More information about NOS is in the NOS Time-Sharing User's reference manual and the Interactive Facility reference manual.

Under any system, if a file specified in a CONNEC exists as a local file but is not connected at the time of the call, the file's buffer is flushed before the file is connected to the terminal.

CONNEC Under NOS/BE

Under NOS/BE, if cs is not specified, it is set to 0. If display code is selected, input/output operations must be formatted, list directed, NAMELIST, or buffered. If either of the ASCII codes is selected, input/output operations must be either formatted or buffered. When a CONNEC specifies a file already connected with the character set specified, the call is ignored. If the file specified is already connected with a character set other than that specified, cs is reset accordingly.

Data input or output through a terminal under INTERCOM is represented ordinarily in a CDC or ASCII 64-character set, depending on installation option. For these sets, ten characters in 6-bit display code are stored in each central memory word. As described above, a terminal user can specify from within a program that data represented in an ASCII 95-character set (providing the capability for recognizing lowercase letters) or an ASCII 256-character set (providing the capability for recognizing lowercase letters, control codes, and parity) be input or output through the terminal. For the ASCII 95-character and 256-character sets, characters are stored in five 12-bit bytes in each central memory word. Characters in the ASCII 95-character set are represented in 7-bit ASCII code right-justified in each byte with binary zero fill. Characters in the ASCII 256-character set are represented in 8-bit ASCII code right-justified in each byte with binary zero fill. See appendix A.

When data represented in either ASCII character set code is transferred with a formatted input/output statement, the maximum record length should be specified in the PROGRAM statement as twice the number of characters to be transferred (section 6). Allowance should also be made in input/output operations for the fact that internal characters require twice as much space as external characters.

CONNEC Under NOS

Under NOS, if CONNEC specifies an existing local file, the buffers for the file are flushed (if it is an output file) and the file is returned. A subsequent DISCON for the file causes the connected file to be returned, but the preexisting file is not reassociated with the file name.

For a program run under NOS, any file can be connected to the terminal by the ASSIGN command. In addition, the user can connect any file from within the program by using CALL CONNEC.

Data input or output through a terminal under NOS is represented ordinarily in a standard 64-character set. However, the user can elect to have data represented in an ASCII 128-character set (which provides the capability for recognizing control codes and lowercase, as well as uppercase, letters) by entering the ASCII command. Characters contained in the standard set are stored internally in 6-bit display code, whether or not the ASCII command has been entered. The additional characters which complete the ASCII 128-character set are stored internally in 12-bit display code if the ASCII command has been entered; otherwise, they are mapped into the standard 64-character set and stored internally in 6-bit display code. See appendix A.

DISCON

The DISCON call shown in figure 7-23 disconnects a file from within a program. This request is ignored if the specified file is not connected. After execution of this statement under NOS/BE, the specified file remains local to the terminal. In addition, if the file existed prior to connection, the file name is reassociated with the information contained on the device where the file resided prior to connection. Data written to a connected file is not contained in the file after it is disconnected. Under NOS, a CALL DISCON causes the connected file to be returned; the disconnected file name is not reassociated with the preexisting information.

CALL DISCON(u)

u Is the unit designator.

Figure 7-23. DISCON Call

MASS STORAGE INPUT/OUTPUT

Mass storage input/output (MSIO) subroutines allow the user to create, access, and modify files on a random basis without regard for their physical positioning. Each record in the file can be read or written at random without logically affecting the remaining file contents. The length and content of each record are determined by the user. A random file can reside on any mass storage device. CYBER Record Manager word addressable file organization is used to implement MSIO files. The CYBER Record Manager reference manual contains details of word-addressable implementation.

A file processed by mass storage subroutines should not be processed by any other form of input/output.

RANDOM FILE ACCESS

A randomly accessible file capability is provided by the mass storage input/output subroutines. Random files offer the same advantages as direct access files (described in section 5). In a random file, as in a direct access file, any record can be read, written, or rewritten directly, because the file resides on a random access mass storage device that can be positioned to any portion of a file.

NOTE

Direct access files should be used where possible because they are ANSI standard. However, applications requiring variable length randomly accessible records must use the random file subroutines since the standard direct access file capability only allows fixed length records.

To permit random accessing, each record in a random file is uniquely and permanently identified by a record key. A key is an 18- or 60-bit quantity, selected by the user and included as a parameter on the call to read or write a record. When a record is first written, the key in the call becomes the permanent identifier for that record. The record can be retrieved later by a read call that includes the same key, and it can be updated by a write call with the same key.

When a random file is in active use, the record key information is kept in an array in the user's field length. The user is responsible for allocating the array space by a DIMENSION, type, or similar array declaration statement, but must not attempt to manipulate the array contents. The array becomes the directory or index to the file contents. In addition to the key data, it contains the word address and length of each record in the file. The index is the logical link that enables the mass storage subroutines to associate a user call key with the hardware address of the required record.

The index is maintained automatically by the mass storage subroutines. The user must not alter the contents of the array containing the index in any manner: to do so might result in destruction of the file contents. (In the case of a subindex, the user must clear the array before using it as a subindex, and read the subindex into the array if an existing file is being reopened and manipulated. However, individual index entries should not be altered.)

When a permanent file that was created by mass storage input/output routines is to be modified it must be attached with modify and extend permissions (append permission under NOS). Under NOS/BE and SCOPE 2, the EXTEND control statement should be used after the file is modified. Failure to extend the file can render it unusable.

In response to a call to open the file, the mass storage subroutine automatically clears the assigned index array. The index array should be noncharacter to insure that it begins on a word boundary. If an existing file is being reopened, the mass storage subroutines locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to the mass storage device. When the file is reopened, by the same job or another job, the index is again read into the index array space provided, so that file manipulation can continue.

Object time input/output subroutines control the transfer of records between central memory and mass storage.

NOTE

The ARG=FIXED parameter cannot be specfied on the FTN5 control statement if any default parameters are used in the following routines.

OPENMS

OPENMS opens the mass storage file and informs the system that it is a random (word addressable) file. Figure 7-24 shows the format for OPENMS.

The array (ix) specified in the call is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

Example:

```
DIMENSION I(11)
CALL OPENMS(5,I,11,0)
```

These statements prepare for random input/output on the file TAPE5 using an 11-word (10 entry) master index of the number type. If the file already exists, the master index is read into memory starting at address I.

CALL OPENMS(u,ix,lngth,t)

u	Is the unit specifier.
ix	Is the name of the array containing the master index.
lngth	Is the length of the master index; for a number index lngth $\geqslant$ (number of entries in master index)+1; for a name index lngth $\geqslant$2 *(number of entries in master index)+1.
t	Is the type of index; can have integer value 0 or 1: t = 0 File has a number master index. t = 1 File has a name master index.

Figure 7-24. OPENMS Call

WRITMS

WRITMS transmits data from central memory to the file. Figure 7-25 shows the format for WRITMS.

CALL WRITMS(u,fwa,n,k,r,s)

u	Is the unit specifier.
fwa	Is the name of an array in central memory or LCM (address of first word).
n	Is the number of 60-bit words to be transferred.
k	Is the record key; for number index, $1 \leqslant k \leqslant$ lgnth-1; for name index, k = any 60-bit quantity except 0.
r	Specifies rewrite; can have integer value -1, 0, 1: r = -1 Rewrite in place if new record length does not exceed old record length; otherwise write at end-of-data. r = 0 No rewrite; write at end-of-data (default value). r = 1 Rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length.
s	Specifies subindex flag; can have value 0 or 1: s = 0 Do not write subindex marker flag in index control word (default value). s = 1 Write subindex marker flag in index control word for this record.

Figure 7-25. WRITMS Call

The end-of-data (for r=-1 and r=0) is defined to be immediately after the end of the data record which is closest to end-of-information. The first record written at end-of-data overwrites the old index.

CYBER Record Manager operates more efficiently if n is always a multiple of 64. The r parameter can be omitted if the s parameter is also omitted. The s parameter marks a subindex record which may aid user utilities to distinguish subindex records from data records.

Example:

```
CALL WRITMS (3,DATA,25,6,1)
```

This statement unconditionally rewrites in place of file TAPE3, starting at the address of the array named DATA, a 25-word record with an index number key of 6. The default value is taken for the s parameter.

READMS

READMS transmits data from the file to central memory. Figure 7-26 shows the format for READMS. CYBER Record Manager operates more efficiently if n is always a multiple of 64.

CALL READMS(u,fwa,n,k)

u	**Is the unit specifier.**
fwa	**Is the name of an array in central memory or LCM (address of first word).**
n	**Is the number of 60-bit words to be transferred. If n is less than the record length, n words are transferred without diagnostic.**
k	**Is the record key; for a number index, $1 \leq k \leq$ lngth-1; for name index, k = any 60-bit quantity except ± 0.**

Figure 7-26. READMS Call

Example:

```
CALL READMS (3,MORDAT,25,2)
```

This statement reads the first 25 words of record 2 from unit 3 (TAPE3) into central memory starting at the address of the array MORDAT.

CLOSMS

CLOSMS writes the master index from central memory to the file and closes the file. Figure 7-27 shows the syntax for CLOSMS. CLOSMS is provided to close a file so that it can be returned to the operating system before the end of a run, to preserve a file created by an experimental job that might subsequently abort, or to perform other special functions.

CALL CLOSMS(u)

u	**Is the unit specifier.**

Figure 7-27. CLOSMS Call

Since new data records can overwrite the old index, a file which has had new data records added is invalid unless the file is closed. (Under NOS/BE and SCOPE 2, permanent files must also be extended.) Jobs which might abort before closing the files should use RECOVR to recover and terminate normally (that is, STOP) to cause the files to be closed.

When using mass storage input/output subroutines in overlays or segments, care should be taken to close a file before program termination. If this is not possible, the mass storage input/output routines must reside in the (0,0) overlay or root segment. This can be done by including a call to an MSIO routine in the (0,0) overlay or root segment (the call need not be executed), or by using the LIBLOAD control statement.

Example:

```
CALL CLOSMS (2)
```

This statement closes the file TAPE2.

STINDX

STINDX selects a different array to be used as the current index to the file. Figure 7-28 shows the format for STINDX. The call permits a file to be manipulated with more than one index. For example, when the user wishes to use a subindex instead of the master index, STINDX is called to select the subindex as the current index. The STINDX call does not cause the subindex to be read or written; that task must be carried out by explicit READMS or WRITMS calls. It merely updates the internal description of the current index to the file.

CALL STINDX(u,ix,lngth,t)

u	**Is the unit specifier.**
ix	**Is the name of the array in central memory containing the subindex (first word address).**
lngth	**Is the length of subindex; for a number index lngth $\geq$ (number of entries in subindex)+1; for a name index lngth $\geq$2*(number of entries in subindex)+1.**
t	**Is the type of subindex; can have integer value 0 or 1; if omitted, t is the same as the current index:** **t = 0 File has a number subindex.** **t = 1 File has a name subindex.**

Figure 7-28. STINDX Call

Example:

```
DIMENSION SUBIX (10)
CALL STINDX (3,SUBIX,10,0)
```

These statements select a new index, SUBIX, for file TAPE3 with an index length of 10 (up to nine entries). The records referenced via this subindex use number keys.

Example 2:

```
DIMENSION MASTER (5)
CALL STINDX (2,MASTER,5)
```

These statements select a new index, MASTER, from file TAPE2 with an index length of 5 and index type unchanged from the last index used.

Index Key Types

There are two types of index key, name and number. A name key can be any 60-bit quantity except +0 or -0. A number key must be a simple positive integer, greater than 0 and less than or equal to the length of the index in words, minus 1 word. The user selects the type of key by the t parameter of the OPENMS call. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. If the user should inadvertently attempt to reopen an existing file with an incorrect index type parameter, the job will be aborted. (This does not apply to subindexes chosen by STINDX calls; proper index type specification is the sole responsibility of the user.) In addition, key types cannot be mixed within a file. Violation of this restriction might result in destruction of a file.

The choice between name and number keys is left entirely to the user. The nature of the application may clearly dictate one type or the other. However, where possible, the number key type is preferable. Job execution will be faster and less central memory space will be required. Faster execution occurs because it is not necessary to search the index for a matching key entry (as is necessary when a name key is used). Space is saved due to the smaller index array length requirement.

Master Index

The master index type for a given file is selected by the t parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created; attempts to do so by reopening the file with the opposite type index are treated as fatal errors.

Subindex

The subindex type can be specified independently for each subindex. A different subindex name/number type can be specified by including the t parameter in the STINDX call. If t is omitted, the index type remains the same as the current index. Intervening calls which omit the t parameter do not change the most recent explicit type specification. The type remains in effect until changed by another STINDX call.

STINDX cannot change the type of an index which already exists on a file. The user must ensure that the t parameter in a call to an existing index agrees with the type of the index in the file. Correct subindex type specification is the responsibility of the user; no error message is issued.

Multilevel File Indexing

When a file is opened by an OPENMS call, the mass storage routines clear the array specified as the index area, and if the call is to an existing file, locates the file index and reads it into the array. This creates the initial or master index.

The user can create additional indexes (subindexes) by allocating additional index array areas, preparing the area for use as described below, and calling the STINDX subroutine to indicate to the mass storage routine the location, length and type of the subindex array. This process can be chained as many times as required, limited only by the amount of central memory space available. (Each active subindex requires an index array area.) The mass storage routine uses the subindex just as it uses the master index; no distinction is made.

A separate array space must be declared for each subindex that will be in active use. Inactive subindexes can, of course, be stored in the random file as additional data records.

The subindex is read from and written to the file by the standard READMS and WRITMS calls, since it is indistinguishable from any other data record. Although the master index array area is cleared by OPENMS when the file is opened, STINDX does not clear the subindex array area. The user must clear the subindex array to zeros. If an existing file is being manipulated and the subindex already exists on the file, the user must read the subindex from the file into the subindex array by a call to READMS before STINDX is called. STINDX then informs the mass storage routine to use this subindex as the current index. The first WRITMS to an existing file using a subindex must be preceded by a call to STINDX to inform the mass storage routine where to place the index control word entry before the write takes place.

If the user wishes to retain the subindex, it must be written to the file after the current index designation has been changed back to the master index, or to a higher level subindex by a call to STINDX.

The following examples illustrate the use of index key type. In figure 7-29, program MS1 creates a random file with a number index. The program MS2 adds two new records to the file created by MS1.

Figure 7-30 shows how program MS3 creates a random file with a name index. The key names are RECORD1 through RECORD4.

Finally, in figure 7-31 program MS4 creates a subindexed file with a number index. The program uses four subindexes with nine records within each subindex, for a total of 36 records.

DEBUGGING ROUTINES

A number of miscellaneous routines for debugging are provided. The user should refer to section 10 for the description of additional debugging capabilities.

```
      PROGRAM MS1(TAPE3)
C
C   CREATE RANDOM FILE WITH NUMBER INDEX
      DIMENSION INDEX(11), DATA(25)
      CALL OPENMS(3, INDEX, 11, 0)
      DO 50 NRKEY = 1,10
      .
      .    (Generate record in array named DATA.)
      .
      CALL WRITMS(3, DATA, 25, NRKEY)
   50 CONTINUE
      END

      PROGRAM MS2(TAPE3)
C
C   MODIFY RANDOM FILE CREATED BY PROGRAM MS1
C   NOTE LARGER INDEX BUFFER TO ACCOMMODATE TWO NEW RECORDS.
      DIMENSION INDEX(13), DATA(25), MORDAT(40)
      CALL OPENMS(3, INDEX, 13, 0)
C   READ 8TH RECORD FROM FILE TAPE3
      CALL READMS(3, DATA, 25, 8)
      .
      .    (Modify array named DATA.)
      .
C   WRITE MODIFIED ARRAY AS RECORD 8 AT END-OF-INFORMATION
C   IN THE FILE
      CALL WRITMS(3, DATA, 25, 8)
C   READ 6TH RECORD
      CALL READMS(3, DATA, 25, 6)
      .
      .    (Modify array)
      .
C   REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6
      CALL WRITMS(3, DATA, 25, 6, 1)
C   READ 2ND RECORD INTO LONGER ARRAY AREA
      CALL READMS(3, MORDAT, 25, 2)
      .
      .    (Add 15 new words to array named MORDAT.)
      .
C   IN-PLACE REWRITE OF RECORD 2.  IT WILL DEFAULT TO A NORMAL
C   WRITE AT END-OF-INFORMATION SINCE THE NEW RECORD IS LONGER
C   THAN THE OLD ONE, AND FILE SPACE IS THEREFORE UNAVAILABLE.
      CALL WRITMS(3, MORDAT, 40, 2, -1)
C   READ THE 4TH AND 5TH RECORDS
      CALL READMS(3, DATA, 25, 4)
      CALL READMS(3, MORDAT, 25, 5)
      .
      .    (Modify the arrays named DATA and MORDAT.)
      .
C   WRITE THE ARRAYS TO THE FILE AS TWO NEW RECORDS
      CALL WRITMS(3, DATA, 25, 11)
      CALL WRITMS(3, MORDAT, 25, 12)
      END
```

Figure 7-29. Random File With Number Index

```
      PROGRAM MS3(TAPE7)
C
C   CREATE A RANDOM FILE WITH NAME INDEX
      DIMENSION INDEX(9), ARRAY(15, 4)
      CHARACTER REC1, REC2
      DATA REC1, REC2/'RECORD1','RECORD2'/
      CALL OPENMS(7, INDEX, 9, 1)
      .
      .   (Generate data in array area.)
      .
C   WRITE FOUR RECORDS TO THE FILE.  NOTE THAT
C   KEY NAMES ARE RECORD1, RECORD2, RECORD3, AND RECORD4
      CALL WRITMS(7, ARRAY(1, 1), 15, REC1)
      CALL WRITMS(7, ARRAY(1, 2), 15, REC2)
      CALL WRITMS(7, ARRAY(1, 3), 15, 'RECORD3')
      CALL WRITMS(7, ARRAY(1, 4), 15, 'RECORD4')
C   CLOSE THE FILE
      CALL CLOSMS(7)
      END
```

Figure 7-30. Random File With Name Index

```
      PROGRAM MS4(TAPE2)
C   GENERATE SUBINDEXED FILE WITH NUMBER INDEX.  FOUR
C   SUBINDEXES WILL BE USED, WITH NINE DATA RECORDS IN
C   EACH SUBINDEX, FOR A TOTAL OF 36 RECORDS
      DIMENSION MASTER(5), SUBIX(10), RECORD(50)
      CALL OPENMS(2, MASTER, 5, 0)
      DO 10 MAJOR = 1,4
C   CLEAR THE SUBINDEX AREA
        DO 20 I = 1, 10
        SUBIX(I) = 0
20      CONTINUE
C   CHANGE THE INDEX IN CURRENT USE TO SUBIX
      CALL STINDX(2, SUBIX, 10)
C   GENERATE AND WRITE NINE RECORDS
        DO 30 MINOR = 1, 9
        .
        .
        .
C   WRITE A RECORD
          CALL WRITMS(2, RECORD, 50, MINOR)
30      CONTINUE
C   CHANGE BACK TO THE MASTER INDEX
      CALL STINDX(2, MASTER, 5)
C   WRITE THE SUBINDEX TO THE FILE
      CALL WRITMS(2, SUBIX, 10, MAJOR, 0, 1)
10    CONTINUE
C   READ THE 5TH RECORD INDEXED UNDER THE 2ND SUBINDEX
      CALL READMS(2, SUBIX, 10, 2)
      CALL STINDX(2, SUBIX, 10)
      CALL READMS(2, RECORD, 50, 5)
      .
      .   (Manipulate the selected record as desired.)
      .
      END
```

Figure 7-31. Subindexed File With Number Index

DUMP and PDUMP

The DUMP call shown in figure 7-32 and the PDUMP call shown in figure 7-33 dump central memory on the OUTPUT file in the indicated format, except when the ARG=FIXED parameter is specified.

```
CALL DUMP(a,b,f[,a,b,f] . . .)

    a     Specifies the beginning of storage to be
          dumped.

    b     Specifies the end of storage to be
          dumped.

    f     Is a format indicator that can be:

          0   Produce octal dump.
          1   Produce real dump.
          2   Produce integer dump.
          3   Same as 0.
```

Figure 7-32. DUMP Call

```
CALL PDUMP(a,b,f[,a,b,f] . . .)

    a     Specifies the beginning of storage to be
          dumped.

    b     Specifies the end of storage to be
          dumped.

    f     Is a format indicator that can be:

          0   Produce octal dump.
          1   Produce real dump.
          2   Produce integer dump.
          3   Same as 0.
```

Figure 7-33. PDUMP Call

PDUMP returns control to the calling program; DUMP terminates program execution. The maximum number of arguments for the triplet a, b, and f is 20.

For f values 0 through 3, a and b are the first and last words dumped. If 4 is added to any f value, the values of a and b are used as the addresses of the first and last words dumped within the job's field length. The LOCF function can be used to get addresses for the a and b parameters.

STRACE

The STRACE call is shown in figure 7-34. STRACE provides traceback information from the subroutine calling STRACE back to the main program. Traceback information is written to the file OUTPUT.

```
CALL STRACE
```

Figure 7-34. STRACE Call

LEGVAR

The LEGVAR function is shown in figure 7-35. LEGVAR checks the value of variable a and returns the result -1 if the variable is indefinite, +1 if out of range, and 0 otherwise. Variable a is type real; the result is type integer.

```
LEGVAR(a)

    a     Is a real variable.
```

Figure 7-35. LEGVAR Function

SYSTEM

The subroutine SYSTEM enables the user to issue an execution-time error message. Figure 7-36 shows the format for SYSTEM.

```
CALL SYSTEM(ernum,mesg)

    ernum   Is the error number. A decimal
            integer value from 0 through 9999.
            Error numbers used by the compiler
            retain the severity associated with
            them. Error numbers 51 (nonfatal)
            and 52 (fatal) are reserved for the
            user. If an error number greater
            than the highest number defined in
            appendix B is specified, 52 is
            substituted.

    mesg    Is the error message. Entered as
            character constant with the first
            character used as a carriage con-
            trol character and not printed.
```

Figure 7-36. SYSTEM Call

If error number zero is entered, the message is ignored, the output buffers are flushed, and control is returned to the calling program. Each line is printed unless the line limit of the OUTPUT file is exceeded, in which case the job is terminated.

Example:

```
CALL SYSTEM (3,'CHECK DATA')
```

SYSTEMC

SYSTEMC enables the user to alter the contents of the error table, which contains specifications that regulate error processing. The error table is ignored for erroneous data input from a connected (terminal) file. Figure 7-37 shows the format for SYSTEMC.

In the error table, the first error corresponds to error number 1, the second to error number 2, and so forth. Each entry has the format shown in figure 7-38.

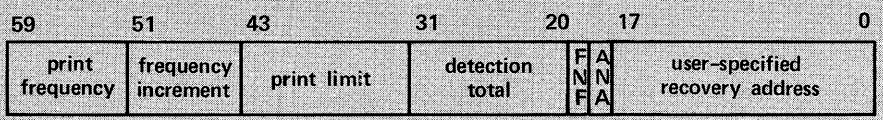
CALL SYSTEMC(errnum,speclist)

errnum — Is the error number for which nonstandard recovery is to be implemented.

speclist — Is an integer array containing error processing specification in consecutive locations:

word 1	F/NF (1 = fatal, 0 = nonfatal).
word 2	Print frequency.
word 3	Frequency increment.
word 4	Print limit.
word 5	User-specified error recovery routine address.
word 6	Maximum traceback limit applicable to al errors; this limit is 20 unless changed by a call to SYSTEMC.

Figure 7-37. SYSTEMC Call



print frequency — By default, print frequency value is 0. If the value is changed to n by a call to SYSTEMC, diagnostic and traceback information is listed every nth time until the print limit is reached.

frequency increment — By default, frequency increment value is 1. This specification can be changed by a call to SYSTEMC if the call specifies print frequency as 0. When frequency increment is 0, diagnostic and traceback information is not listed; when it is 1, such information is listed until the print limit is reached; when the frequency increment is n > 1, such information is listed only the first n times unless the print limit is reached first.

print limit — By default, print limit value is 10. It can be changed by a call to SYSTEMC.

detection total — Detection total is a running count of the number of times an error occurs. The final value is reported in the error summary issued at end-of-job if SYSTEMC is called during execution.

F/NF — This bit specifies the severity of the error: 1 indicates a fatal error; 0, nonfatal. The severities of system defined errors are given in appendix B. All errors defined by the user with these numbers in a call to SYSTEM retain the specified severity. The severity of any error can be changed by a call to SYSTEMC, however.

A/NA — The A/NA bit is ignored unless a nonstandard recovery address is specified; it can be set only during assembly of SYSTEMC. When this bit is set, the address in an auxiliary table is passed in the third word of the secondary argument list to the recovery routine. Each word in the auxiliary table must have the error number in its upper 10 bits, so that the address of the first error number match is passed to the recovery routine. An entry in the auxiliary table for an error is not limited to any specific number of words.

user-specified recovery address — This address is specified in a call to SYSTEMC.

Figure 7-38. Error Table Entry

In an overlay program, if SYSTEMC is not called in the (0,0) overlay, the routine might not be available to higher level overlays. When SYSTEMC is called from an overlay or segment, it must reside in the (0,0) overlay or the root segment.

A negative value for any word in the speclist indicates that the current value of that specification is not to be changed. A user-specified error recovery routine activated by a call to SYSTEMC can be canceled by a subsequent call with word 5 of the speclist set to zero.

If SYSTEMC has been called, an error summary is issued at job termination indicating the number of times each error occurred since the first call to SYSTEMC. Figure 7-39 shows a standard error recovery in a math library routine and how to suppress error message 115.

For an error detected by a routine in the math library, a user-supplied error recovery routine should be a function subprogram of the same type as the function detecting the error. For any other error, a user-supplied error recovery should be a subroutine subprogram.

When an error previously referenced by a SYSTEMC call is detected, the following sequence of operations is initiated:

1. Diagnostic and traceback information is printed in accordance with the specification in the pertinent error table entry. The traceback information is terminated for any of the following conditions:

   Calling routine is a program.

   Maximum traceback limit is reached.

   No traceback information is supplied.

2. If the SYSTEMC call references a user-specified error recovery routine address, SYSTEMC, FORSYS=, and the routine detecting the error are delinked from the calling chain, and the user-supplied error recovery routine is entered.

3. If the error is nonfatal, control returns to the routine that called the routine detecting the error. An error summary is printed at job termination.

4. If the error is fatal, all output buffers are flushed, an error summary is printed, and the job is terminated.

If a nonstandard recovery address is specified in the SYSTEMC call, the following information is available to the user recovery routine:

Register	Contents
A1	Address of argument list passed to routine detecting the errors detected by a math library routine. Address of the FIT for error 103. Undefined for all other errors.
X1	Address of the first argument in the list for errors detected by a math library routine. Undefined for all other errors.
A0	Address of argument list of routine that called the routine detecting the error.
B1	Address of a secondary argument list containing, in successive words: Error number associated with this error. Address of message associated with this error. Address within auxiliary table if A/NA bit set; otherwise 0. In upper 30 bits, instruction consisting of RJ to SYSERR.j; in lower 30 bits, address of traceback information for routine detecting the error. Information in the secondary argument list is not available to user-supplied error recovery routines coded in FORTRAN.

```
      PROGRAM EXPECT(OUTPUT)
      DIMENSION IRAY(6)
      DATA IRAY/6* -0/
C     SET PRINT LIMIT TO ZERO
      IRAY(4) = 0
      X = EXP(800.0)
      X = EXP(-800.0)
C     CALL SYSTEMC TO INHIBIT PRINTING OF ERROR 115
C     AND START ERROR SUMMARY ACCUMULATION
      CALL SYSTEMC(115, IRAY)
      PRINT *
      PRINT *, '*****SYSTEMC IS CALLED TO SUPPRESS '//
     +         'PRINTING OF ERROR 115'
      X = EXP(800.0)
      X = EXP(-800.0)
      PRINT *
      PRINT *, '*****ERROR 115 DETECTED BUT NOT PRINTED'
      END
```

Figure 7-39. Suppressing an Error Message

A2	Address of error table entry for this error.
X2	Contents of error table entry for this error.

LIMERR and NUMERR

The LIMERR call shown in figure 7-40 and NUMERR function shown in figure 7-41 enable the user to input data without the risk of termination when improper data is encountered. When LIMERR is used, the program does not terminate when data errors are encountered until the number of errors occurring after the call exceeds the value of lim. The NUMERR function returns the number of errors since the last LIMERR call. The result is integer.

```
CALL LIMERR(lim)

lim    Is the limit for the number of errors.
```

Figure 7-40. LIMERR Call

```
NUMERR( )
```

Figure 7-41. NUMERR Function

LIMERR can be used to inhibit job termination when data is being input with a formatted, NAMELIST, or list directed read or with DECODE statements. It operates only when data is encountered that would ordinarily cause job termination under error number 78 ("ILLEGAL DATA IN FIELD") or error number 79 ("DATA OVERFLOW"). LIMERR has no effect on the processing of errors in data input from a connected (terminal) file.

LIMERR initializes an error count and specifies a maximum limit (lim) on the number of data errors allowed before termination. LIMERR continues in effect for all subsequent READ statements until the limit is reached. LIMERR can be reactivated with another call, which will reinitialize the error count location and reset the limit. A LIMERR call with lim specified as zero nullifies a previous call; improper data will then result in job termination as usual.

When improper data is encountered in a formatted or NAMELIST read (or in a DECODE statement) with LIMERR in effect, the bad data field is bypassed, and processing continues at the next field. When improper data is encountered in a list directed read, control moves to the statement immediately following the READ statement.

NUMERR returns the number of errors since the last LIMERR call. The previous error count is lost when LIMERR is called, and the count is reinitialized to zero.

Figure 7-42 illustrates the use of LIMERR and NUMERR to suppress normal fatal termination when large sets of data are being processed.

When LIMERR is called, a limit of 200 errors is established. The number of errors is reset to zero. After ARAY is read, NUMERR() is checked. If errors occur, the following statements are not processed and a branch is made to statement 500. Had LIMERR not been called, fatal errors would have terminated the program before the branch to statement 500. At statement 500, LIMERR once more initializes the error count, and execution continues.

```
      CALL LIMERR(200)
      READ(1, 125) (ARAY(I), I = 1, 1500)
125   FORMAT (3F10.5, E10.1)
      IF (NUMERR( ) .GT. 0) GO TO 500
      .
      .
      .
500   CONTINUE
      CALL LIMERR(200)
      READ(1, 125) (BRAY(I), I = 1, 1500)
      IF (NUMERR( ) .GT. 0) GO TO 600
      .
      .
      .
600   CONTINUE
      CALL LIMERR(100)
      READ(1, 230) (LRAY(I), I = 1, 500)
      WRITE(2, *) NUMERR( )
      READ(4, 127) (MRAY(I), I = 1, 500)
      WRITE(2, *) NUMERR( )
      .
      .
      .
```

Figure 7-42. Suppressing Fatal Termination

COLLATING SEQUENCE CONTROL

Character relational expressions are evaluated according to a collating sequence, determined by a collation weight table. A weight table is a 1-dimensional integer array of size 63 or 64 with a lower bound of zero. Each element of the weight table has a value between zero and S-1 inclusive. The value of element i of the weight table is the collating weight for the character with the character code of i. The character codes and graphic representations for all characters supported by the processor are given in appendix A. If c(i) and c(j) are characters and i and j are their respective collating weights, then c(i) .op. c(j) has the value .TRUE. if and only if i.op.j has the value .TRUE., where .op. is any of the relational operators.

The value of a weight table element does not have to be unique within the table; that is, several characters can have the same collating weight.

Collation can be directed by the fixed collation weight table or by the user-specified collation weight table. The fixed table is predefined as the display code weight table and cannot be modified; the user-specified weight table is predefined as the ASCII6 weight table and can be accessed and modified by the program. The CS parameter on the FTN5 control statement and the C$ COLLATE compiler directive determine whether the fixed or user-specified table controls current collation. The predefined weight tables appear in appendix A.

The ASCII collating sequence used by the intrinsic functions LGE, LGT, LLE, and LLT is independent of both the fixed and user-specified weight tables and is therefore unaffected by either the compiler-call statement option or the collation control directive. The intrinsic function INDEX does not use either collation weight table.

A program can access or modify the user-specified collation weight table by using the procedures COLSEQ, CSOWN or WTSET.

COLSEQ

A processor-defined user-specified weight table is selected by a reference to utility subroutine COLSEQ, shown in figure 7-43.

CALL COLSEQ(a)

a Is a character expression whose value when any trailing blanks are removed is one of the values ASCII6, COBOL6, DISPLAY, or STANDARD.

Figure 7-43. COLSEQ Call

The collating sequences which can be selected are:

ASCII6
COBOL6
DISPLAY
STANDARD

If STANDARD is specified, COBOL6 values are used when the operating system is using the CDC graphic set; and ASCII6 values are used when the operating system is using the CDC ASCII subset.

Example:

To select a collating sequence identical to COBOL6, with the exception that characters $ and . sort equally ('$' .EQ. '.'), the following can be used.

```
CALL COLSEQ ('COBOL6')
CALL WTSET ('$',ICHAR('.'))
```

The user-specified weight table is initialized to the COBOL standard collating sequence, and the entry for the character code $ (53_8) is reset to 12_{10} (the value of the weight table indexed by 57_8 ('.')). Refer to appendix A for the collation weight tables.

WTSET

A program can modify the user-specified weight table by a reference to utility subroutine WTSET, shown in figure 7-44.

CALL WTSET(ind,wt)

ind Is a character expression of length 1 or an integer expression with a value between zero and S-1 inclusive.

wt Is an integer expression with a value between zero and S-1 inclusive.

Figure 7-44. WTSET CALL

If ind is a character expression with value c, the element of the weight table indexed by the character code of c is replaced with wt; if ind is an integer expression, the weight table element indexed by ind is replaced with wt.

CSOWN

A program can specify a partial collating sequence by a reference to utility subroutine CSOWN, shown in figure 7-45.

CALL CSOWN(str)

str Is a character expression of length 1 through S inclusive, with a value of the form:

$c(1)c(2)c(3) \ldots c(n)$, $1 \leq n \leq S$

No c(i) can equal c(j) unless i equals j.

Figure 7-45. CSOWN Call

CSOWN explicitly defines the weight table elements for the n characters and then sets all other elements to zero. For i from 1 to n, the element p(i) of the user-specified weight table is set to i-1, where p(k) is the character code of c(k).

STATIC CAPSULE LOADING ROUTINES

The STATIC option can be used for programs that are not compatible with Common Memory Manager (CMM). For example, in programs which require overindexing of blank common, CMM cannot be used. STATIC is also used for time-critical real-time applications where the cost of using Fast Dynamic Loader (FDL) cannot be afforded.

To use the STATIC option, the user must:

1. Compile the main program with the STATIC option specified on the FTN5 control statement (subroutines can also be compiled by using this option, but it is not essential).
2. The PROGRAM statement must specify all files to be used.
3. The program must include calls to all STLxxx routines appropriate to the types of I/O in the program.

The initial load of the program requires the specification of which CYBER Record Manager (CRM) capsules are to be included. This specification is accomplished either:

Internally through a reference to one of the STLxxx subroutines supplied by the FORTRAN5 library

or

Externally through the use of the FILE statement USE parameter in conjunction with a LDSET (STAT=logical file name) directive.

The STLxxx subroutines supply only those capsules required for the default file attributes of a particular input/output operation. For a description of the external FILE/LDSET technique, refer to the CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual.

During program execution, each STLxxx subroutine reference immediately returns control upon invocation. References to STLxxx subroutines are made solely for the load-time effect of the LDSET directives contained within them. The execution of these subroutines is not required.

Necessary FORTRAN error processing capsules are also loaded by any reference to a STLxxx subroutine, which in turn force loads the STLERR subroutine. In an overlay or segmented environment, all STLxxx calls should be made within the (0,0) overlay or root segment.

Table 7-3 gives a description of the STLxxx subroutine names, the I/O operations requiring them, and the supported file organization (FO), block type (BT), and record type (RT) applicable to each routine.

The use of the terms capsule and CYBER Record Manager does not apply to SCOPE 2. However, calls to the STLxxx subroutines are required to inhibit Common Memory Manager operation on SCOPE 2. Note that all STLxxx subroutine names are described to provide complete documentation, but only STLERR need be called.

NOTE

Please refer to appendix G for recommendations on the use of the STATIC option.

TABLE 7-3. STATIC CAPSULE LOADING ROUTINES

Subroutine Name	Required by	Supports		
		FO	RT	BT
CALL STLBAK	Sequential BACKSPACE	SQ	I C C	W Z S
CALL STLENF	Sequential ENDFILE	SQ	I C C	W Z S
CALL STLERR	All static jobs to omit CMM	Force load FORTRAN error recovery capsule		
CALL STLIBI (Input Binary)	Sequential Unformatted READ	SQ	I	W
CALL STLIBU (Input Buffered)	Sequential BUFFER IN	SQ	C	S
CALL STLICO (Input Coded)	Sequential Formatted List Directed and Namelist Input	SQ	C	Z
CALL STLIDB (Input Direct Access Binary)	Direct Access Unformatted Input	WA	C	U
CALL STLIDC (Input Direct Access Coded)	Direct Access Formatted Input	WA	C	U
CALL STLINQ	Explicit INQUIRE statements	Force load INQCAP from FTN5LIB		
CALL STLOBI (Output Binary)	Sequential Unformatted Output	SQ	I	W
CALL STLOBU (Output Buffered)	Sequential BUFFER OUT	SQ	C	S
CALL STLOCO (Output Coded)	Sequential Formatted, List Directed and Namelist output	SQ	C	Z
CALL STLODB (Output Direct Access Binary)	Direct Access Unformatted Output	WA	C	U
CALL STLODC (Output Direct Access Coded)	Direct Access Formatted Output	WA	C	U
CALL STLOPE	Explicit OPEN Statements	Force load OPECAP from FTN5LIB		
CALL STLREW	Sequential REWIND	SQ	I C C	W Z S
CALL STLRMS (Random Access Mass Storage)	Random Access I/O Routines (OPENMS, etc)	WA	C	U

This section describes the FORTRAN interfaces for CYBER Record Manager, Common Memory Manager, Sort/Merge, and COMPASS Assembly Language, as well as the methods used by FORTRAN for subprogram linkage.

FORTRAN-CYBER RECORD MANAGER INTERFACE

NOTE

Refer to appendix G for recommendations on the use of this feature.

The CYBER Record Manager interface subroutines correspond closely to the CYBER Record Manager COMPASS macros. The names are different in some cases, and the parameters are not necessarily specified in the same order, but the processing performed by each subroutine is for the most part the same as the corresponding COMPASS macro.

Only a summary of the format, parameters, and purpose of each subroutine is given here. The differences in usage of these routines among the file organizations are not discussed. In order to use these routines, it is necessary to refer to the CYBER Record Manager AAM and BAM reference manuals.

The user can either allocate buffers within a program block or allow CYBER Record Manager to allocate them dynamically when the file is opened.

To allocate a buffer within the program block, an array must be dimensioned and the length and position of the array specified by the BFS and FWB fields of the file information table. If either of these fields is zero when the file is opened, CYBER Record Manager allocates a buffer in central memory following the executable code and blank common (if declared). In an overlay program, dynamically allocated buffers are assigned to memory beyond the last word address of the longest overlay chain.

These routines are available under NOS/BE and NOS only. If any of these routines are called, the ARG=FIXED control statement option cannot be specified.

PARAMETERS

The first parameter in the call to every subroutine is the name of the array containing the file information table being processed. This array should be dimensioned 35 words long; 20 words for the file information table itself and 15 for the file environment table. Any other parameters can be omitted; default values are supplied by CYBER Record Manager. With the exception of FILExx, parameters are identified strictly by position; thus, parameters can be omitted only from the right.

When a program is compiled with OPT=2 or OPT=3, wsa must be specified on all calls to GET, GETP, and GETN. Also, ka must be specified on calls to GETN and PUT for indexed sequential, direct access, and actual key files. If wsa and ka are provided as character type data, they must be word aligned. Word alignment can only be assured by placing 10-character strings of the character data in a common block. (A common block begins on a word boundary.)

Most of the parameters establish values for file information table fields. CYBER Record Manager always uses the most recent value established for a field; if a parameter is omitted, the previous contents of the field are used instead.

If the same subroutine is called twice in the same program unit with a different number of parameters, an informative diagnostic is issued by the compiler.

Values for the parameters can be:

Array or variable names, identifying areas used for communication between the user program and CYBER Record Manager

Subprogram names for user owncode exits (must be specified in an EXTERNAL statement)

Integer values

L format Hollerith constants, used to express symbolic options and to identify file information table fields

SUBROUTINES

Following are brief descriptions of the FORTRAN-CYBER Record Manager subroutine calls. The actual formats of the calls, along with descriptions of the parameters, are shown in figure 8-1. The precise meaning of any parameter depends on the file organization of the file being processed, as well as the subroutine being called. Not all parameters are applicable to all file organizations.

CLOSEM

CLOSEM closes the file after all processing has been completed. Only STOREF and IFETCH can follow execution of CLOSEM.

DLTE

DLTE deletes a record from an indexed sequential, direct access, or actual key file. The key of the record to be deleted is in the location specified by ka.

ENDFILE

ENDFILE writes an end-of-partition.

FILExx

FILExx inserts the specified values into the FIT fields. All parameters, with the exception of fit, are paired. The first parameter in each pair is the name of a file information table field, in L format. The second parameter of each pair is the value to be set in that field. FILExx must be called before the file is opened.

```
CALL CLOSEM(fit,cf,type)
CALL DLTE(fit,ka,kp,pos,ex)
CALL FILExx(fit,keyword,value[,keyword,value] . . .)
    xx is one of the following:
    SQ    sequential file.
    IS    indexed sequential file.
    DA    direct access file.
    AK    actual key file.
    WA    word addressable file.
CALL ENDFILE(fit)
CALL FITDMP(fit,id)
CALL FLUSH1(fit)
CALL FLUSHM(afit)
CALL GET(fit,wsa,{ka|wa},kp,mkl,rl,{ex|dx})
CALL GETN(fit,wsa,ka,ex)
CALL GETNR(fit,wsa,ka,ex)
CALL GETP(fit,wsa,ptl,skip,dx)
IFETCH(fit,field)
CALL IFETCH(fit,field,value)
CALL OPENM(fit,pd,of)
CALL PUT(fit,wsa,rl,{ka|wa},kp,pos,ex)
CALL PUTP(fit,wsa,ptl,rl,ex)
CALL REPLC(fit,wsa,rl,ka,kp,pos,ex)
CALL REWND(fit)
CALL SEEKF(fit,ka,kp,mkl,ex)
CALL SKIP(fit,count)
CALL STARTM(fit,ka,kp,mkl,ex)
CALL STOREF(fit,keyword,value)
CALL WEOR(fit,lev)
CALL WTMK(fit)
```

Values for parameters can be:

Array or variable names, identifying areas used for communication between the user program and CYBER Record Manager

Subprogram names for user owncode exits (must be specified in an EXTERNAL statement)

Integer values

L format Hollerith constants, used to express symbolic options and to identify file information table fields

The following mnemonics are used in the subroutine formats. The precise meaning of any parameter depends on the file organization of the file being processed, as well as the subroutine being called. Not all parameters are applicable to all file organizations.

afit Name of an array that contains a list of addresses of FITs terminated by a word of zeros.

fit Name of a file information table. Linked to the actual file by means of the LFN field.

wsa Working storage area. A variable, array, or array element name indicating the starting location from which data is to be read or into which data is to be written. If character type data is used, wsa must be word aligned.

pd Processing direction established when file is opened:

'INPUT'	Read only.
'OUTPUT'	Write only.
'I-O'	Read and write.
'NEW'	File creation. (Indexed sequential, direct access, actual key only.)

of File positioning at open time:

'R'	Rewind.
'N'	No file positioning.
'E'	Extend; file is positioned immediately before end-of-information.

cf File positioning after close:

'R'	Rewind.
'N'	No positioning.
'U'	Unload.
'RET'	Return.
'DIS'	Disconnect (terminal files only).
'DET'	No positioning; release buffer space and remove from active file list.

Figure 8-1. FORTRAN-CYBER Record Manager Interface Calls (Sheet 1 of 2)

type Type of close (not a file information field):

'FILE'	File close
'VOLUME'	Volume close

ka Location of key for access to record in a direct access, indexed sequential, or actual key file. For GETN, key is returned to this location. If character type data is used, ka must be word aligned.

wa Location of word address for read or write of record in a word addressable file.

kp Character position (0 through 9) within word designated by ka in which key begins (direct access, indexed sequential only).

mkl Major key length (indexed sequential only).

rl Record length in characters for record to be read or written.

ex Name of user owncode error exit subroutine.

dx Name of user owncode data exit subroutine.

pos For duplicate key processing (applies only to Initial Indexed Sequential Files):

'P'	Write record preceding current record.
'N'	Write record as next record.
'C'	Delete or replace current record.
0	Delete or replace first record in duplicate key chain.

count Number of records to be skipped; positive count indicates forward skip, negative count indicates backward skip, zero count should not be used.

ptl Number of characters to be used for a partial read or write.

skip Positioning before execution of GETP:

0	Continue reading at current position.
'SKIP'	Skip to beginning of next record before reading.

lev Level number for end-of-section; 0 through 17.

id FIT identifier.

Figure 8-1. FORTRAN-CYBER Record Manager Interface Calls (Sheet 2 of 2)

FITDMP

FITDMP dumps the contents of the file information table to the error file ZZZZZEG. The CRMEP control statement (described in the CYBER Record Manager AAM reference manual) can then be used to print the contents of ZZZZZEG.

FLUSHM

FLUSHM performs all the file close operations (such as buffer flushing), but the file remains open. The process is repeated for each file in the list of FITs in the parameter array.

FLUSH1

FLUSH1 Performs all of the file close operations (such as buffer flushing), but the file remains open.

GET

GET reads a record and returns it to the working storage area (wsa). The last parameter specifies dx for sequential files, ex for all other files.

GETN

GETN reads the next record in sequential order from an indexed sequential, direct access, or actual key file. The key of the record is placed in ka after the read.

GETNR

GETNR transfers the next record in sequential order to the working storage area, unless an input/output operation is required, in which case control returns to the user before the input is complete. The user must continue to call GETNR until the transfer is complete (FP field of the FIT is set to 0).

GETP

GETP reads a partial record. The number of characters to be read is indicated by ptl.

IFETCH

IFETCH is an integer function that returns the current value of a single file information table field. A one-bit field is returned in the sign bit; if the bit is 1, the value of the function is negative; if the bit is 0, the value of the function is positive.

IFETCH can also be called as a subroutine; then, the value is returned in the integer variable specified as the third parameter.

OPENM

OPENM opens a file and prepares it for further processing. Only FILExx, STOREF, and IFETCH can precede execution of OPENM.

PUT

PUT writes a record to the file from the working storage area (wsa).

PUTP

PUTP writes a partial record. The number of characters to be written by PUTP is indicated by ptl; the total number of characters to be written is given by rl (required only for record types U, W, and R).

REPLC

REPLC replaces a record on a sequential, indexed sequential, direct access, or actual key file. The key of the record to be replaced is in the location specified by ka; the new record is in the working storage area indicated by wsa. For sequential files, the last record read is replaced by a record of exactly the same size.

REWND

REWND positions a tape file to the beginning of the current volume. It positions a mass storage file to the beginning-of-information.

SEEKF

SEEKF initiates block transfer to the file buffer. The program can continue processing while the transfer occurs. This overlapping of control memory processing and input/output activity can shorten program execution time.

SKIP

SKIP repositions an indexed sequential or actual key file in a forward or backward direction a specified number of records. If count contains a negative value, the direction is backward; if count contains a positive value, the direction is forward. SKIP does not return a record to the working storage area.

STARTM

STARTM positions an indexed sequential or alternate key index file to a record that meets a specific condition; the record is not transferred to the working storage area. The file is positioned according to the key relation field in the file information table and the current value at the key address location.

STOREF

STOREF specifies a value for a single file information table field. It can be called before or after the file is opened. The keyword is the name of a file information table field, in L format, and value is the value to be placed in that field.

WEOR

WEOR terminates a section or partition, or S type record.

WTMK

WTMK writes a tape mark (equivalent to end-of-partition).

ERROR CHECKING

CYBER Record Manager interface routines perform limited error checking to determine whether the call can be interpreted, but actual parameter values are not checked.

The following fatal error conditions are detected at execution time, and a message appears in the dayfile:

FIT ADDRESS NOT SPECIFIED

Array name was not specified.

FORMAT ERROR

Parameters were not paired (FILExx), or required parameters were not specified (STOREF, IFETCH, or SKIP).

UNDEFINED SYMBOL

A file information table field mnemonic or symbolic option was specified incorrectly; for example, an incorrect spelling, or the OF parameter in OPENM was not specified as R, N, or E.

MULTIPLE INDEX PROCESSING

FORTRAN provides the capability of multiple indexing for IS, DA, and AK files via CYBER Record Manager.

Each multiple-index file has an associated alternate key index file. An alternate key index is a cross-reference table of alternate values and IS, DA, or AK primary key values. The key-field position identifies each table, which consists of all the different alternate key values that occur in the records of the file. Associated with each alternate key value is a list of primary keys, each of which identifies a record containing the alternate key value.

To utilize this capability, the index file is specified in the XN field of the file information table. To open the index file, the subroutine call shown in figure 8-2 is used.

CALL RMOPNX(fit,pd,of)

fit Name of array containing the file information table.

pd Processing direction established when file is opened:

'INPUT'	Read only.
'OUTPUT'	Write only.
'I-O'	Read and write.
'NEW'	File creation (indexed sequential, direct access, actual key only).

of File positioning at open time:

'R'	Rewind.
'N'	Extend; file is positioned.
'E'	Extend; file is positioned immediately before end-of-information.

Figure 8-2. RMOPNX Call

The parameters are the same as those of CALL OPENM. The file can be opened by a CALL OPENM instead of CALL RMOPNX if XN was specified on a FILE control statement rather than by a CALL FILExx.

COMMON MEMORY MANAGER INTERFACE

Common Memory Manager (CMM) is used for the management of field length, except when using the static loading options. CMM ensures that the field length is increased or decreased properly to accommodate assigned blocks. CMM blocks contain random information and are not initialized to any known value.

The FORTRAN 5 user can interface to CMM to assign blocks of memory for arrays. This assignment is completely dynamic, and the blocks should be returned to the system when finished.

The CMM reference manual should be read for a detailed description of CMM usage. the following descriptions are for simple CMM usage:

- CMMALF is called to allocate a fixed position block. The array to be assigned is defined in the FORTRAN program as an array of length 1. The proper offset to the base address of the array is calculated by using the LOCF function, adding one to this base address, and subtracting this value from the first word address of the block returned by CMM. This calculated address, plus any subscript of the array desired, is used to reference array elements. For example, the following statements assign a block and set the fifth element to 1:

```
      PROGRAM CMM1
      DIMENSION CMMAR(1)
      ILEN=10
      CALL CMMALF(ILEN,0,0,IFWA)
      IOFF=IFWA-LOCF(CMMAR(1))+1
      CMMAR(IOFF +5)=1.0
      .
      .
      .
      CALL CMMFRF(IFWA)
      .
      .
      .
```

The calling sequence for CMMALF is:

```
CALL CMMALF(IBLKSZ,ISZCDE,IGRPID,IBLFWA)
```

IBLKSZ Number of words required for the block

ISZCDE Size code

- 0 Fixed size block (should be used in most cases).
- 1 Block can grow at last word address.
- 2 Block can shrink at last word address.
- 4 Block can shrink at first word address.
- 5 Block can grow at last word address and shrink at first word address.
- 6 Block can shrink at first and last word address.
- 7 Block can shrink at first and last word addresses and grow at last word address.

IGRPID Group identifier

- 0 Item does not belong to a group (normal usage).
- 1 The block is assigned to this group. The group number is determined by calling CMMAGR. (See the Common Memory Manager reference manual.) The group number may be any value greater than 0.

The value returned from a call to CMMALF is:

IBLFWA First word address of block allocated by CMM

- CMMFRF is called to free the fixed-position block when it is no longer needed. When the block is freed, the contents of the block are no longer accessible.

  The calling sequence for CMMFRF is:

  CALL CMMFRF(IBLFWA)

  IBLFWA First word address of block (must have been returned by CMMALF)

Other routines are available to accomplish other tasks, such as determining maximum field length and other statistics, assigning blocks to groups, and releasing groups of blocks (see the Common Memory Manager reference manual). All CMM interface routines for NOS and NOS/BE are on the library SYMLIB. Therefore, the statement LDSET,LIB=SYMLIB must be included in the loader directives for a run using the CMM interface routines or the user may add a CALL SYMLIB statement in the main program to select library SYMLIB. SCOPE 2 users must specify SYMIO in the LDSET statement instead of SYMLIB.

The subroutine shown in figure 8-3 should be called to describe a key field when creating a new IS, DA, or AK file. It must be called once for each key field in the record.

CALL RMKDEF(fit,kw,kp,kl,ki,kt,ks,kg,kc)

fit	Name of an array containing the file information table.
kw	Word of record in which key starts (0 = first word).
kp	Starting character position of key (0 through 9).
kl	Key length in characters (1 through 255).
ki	Summary index; reserved (0).
kt	Key type: 0 = symbolic, 1 = signed integer, 2 = unsigned.
ks	Substructure for each primary key list in the index: I = index-sequential; F = FIFO; U (default) = unique; specified as L format Hollerith constant.
kg	Size of repeating group in which key resides (default = 0).
kc	Occurrences of group (default = 0).

Figure 8-3. RMKDEF Call

To position a multiple-index file, the subroutine call shown in figure 8-4 is used.

If the RKW and RKP parameters are set to indicate the primary key, STARTM positions the data file, and subsequent calls to GETN retrieve records in sequential order. If RKW and RKP indicate an alternate key, STARTM positions the index file, and subsequent calls to GETN retrieve records in their order on the index file.

FORTRAN-SORT/MERGE INTERFACE

NOTE

Refer to appendix G for recommendations on the use of this feature.

CALL STARTM(fit,ka,kp,mkl,ex)

fit	Name of array containing file information table.
ka	Location of key for access to record in a direct access, indexed sequential, or actual key file. If character type data is used, ka must be word aligned.
kp	Character position (0 through 9) within word designated by ka in which key begins (direct access, indexed sequential only).
mkl	Major key length (indexed sequential only).
ex	Name of user owncode error exit subroutine.

Figure 8-4. STARTM Call

FORTRAN provides the capability for processing data records under the Sort/Merge system from within a FORTRAN program. The FORTRAN user of this feature should be familiar with the autonomous functioning of the Sort/Merge system as described in the Sort/Merge reference manual.

Sort/Merge uses the unused part of the field length as a scratch area; if this is not adequate, additional field length is obtained from the system. The ARG=FIXED control statement option is not permitted for programs using Sort/Merge.

FORTRAN interfaces with Sort/Merge through the subroutines described in this subsection. Sort/Merge subroutines cannot be used with programs in static mode under NOS and NOS/BE or with programs in dynamic mode under SCOPE 2. The series of calls to Sort/Merge subroutines must begin with a call to SMSORT, SMSORTB, SMSORTP, or SMMERGE. If a file is processed by CYBER Record Manager subroutines, OPENM should be called before any of these routines.

In an overlay structured program using blank common, the Sort/Merge interface routines must not be called from the (0,0) overlay.

The following paragraphs describe the FORTRAN-Sort/Merge subroutines.

SMSORT

SMSORT (figure 8-5) calls for a sort on rotating mass storage.

CALL SMSORT(mrl,ba)

mrl	Maximum length in characters of records to be sorted.
ba	Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. For SCOPE 2, ba is the size of LCM buffer area.

Figure 8-5. SMSORT Call

SMSORTB

SMSORTB (figure 8-6) calls for a balanced tape sort. SMTAPE must also be called.

CALL SMSORTB(mrl,ba)

mrl	Maximum length in characters of records to be sorted.
ba	Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. SMSORTB is not supported on SCOPE 2.

Figure 8-6. SMSORTB Call

SMSORTP

SMSORTP (figure 8-7) calls for a polyphase tape sort.

CALL SMSORTP(mrl,ba)

mrl	Maximum length in characters of records to be sorted.
ba	Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. SMSORTP is not supported on SCOPE 2.

Figure 8-7. SMSORTP Call

SMMERGE

SMMERGE (figure 8-8) calls for merge-only processing.

CALL SMMERGE(mrl,ba)

mrl	Maximum length in characters of records to be merged.
ba	Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. For SCOPE 2, ba is the size of LCM buffer area.

Figure 8-8. SMMERGE Call

SMFILE

SMFILE (figure 8-9) identifies the file to be sorted or merged. SMFILE must be called for each file to be sorted or merged, and once for the file to receive the output (unless SMOWN is called). Files should be properly positioned before they are sorted or merged.

SMKEY

SMKEY (figure 8-10) describes the sort key to be used. One SMKEY call is required for each key. The first call indicates the major key; subsequent calls indicate additional or minor keys in the order encountered.

CALL SMFILE(dis,i/o,lfn,action)

dis — Character expression indicating file disposition:

'SORT'	File to be sorted.
'MERGE'	File to be merged.
'OUTPUT'	File to receive output.

i/o — Expression indicating mode of file input/output:

'FORMATTED' or 'CODED'	File accessed with formatted input/output.
'BINARY'	File accessed with unformatted input/output.
0†	File accessed with interfacing CYBER Record Manager subroutines (see text).

lfn — Integer or Boolean file name indicator:

u	Logical unit number, 0 through 999.
L"filename"	File name left-justified with zero fill.
fit†	When i/o is specified as 0, an array containing the file information table.

action — Character expression indicating file disposition following sort or merge:

'REWIND'
'UNLOAD'
'NONE' (default)

†Does not apply to SCOPE 2.

Figure 8-9. SMFILE Call

CALL SMKEY(charpos,bitpos,nchar,nbits,code,colseq,order)

charpos — Integer specifying position of first character of sort key, considering the first characters as position number 1.

bitpos — Integer specifying position of first bit of sort key in character (or 6-bit byte) specified by charpos, considering the first bit as position number 1.

nchar — Integer specifying number of characters or complete 6-bit byte in sort key.

nbits — Integer specifying number of bits in sort key in excess of those indicated by nchar.

code — Coding identifier; a character expression having one of the following values:

'DISPLAY'	Internal display code.
'FLOAT'	Floating-point data.
'INTEGER'	Signed integer data.
'LOGICAL'	Unsigned integer data (default).

The following identifiers must be specified in pairs separated by a comma, as indicated. Each pair is positionally interchangeable.

'SIGN','LEADING'	Numeric data in display code; sign present as an overpunch at beginning of field.
'SIGN','TRAILING'	Numeric data in display code; sign present as an overpunch at end of field.
'SEPARATE','LEADING'	Numeric data in display code; sign is a separate character at beginning of field.
'SEPARATE','TRAILING'	Numeric data in display code; sign is a separate character at end of field.

colseq — Character expression specifying collating sequence (applicable only if code is specified as DISPLAY);

'ASCII6'	6-bit ASCII collating sequence (default for installations using ASCII character set).
'COBOL6'	6-bit COBOL collating sequence (default for installations using CDC character set).
'DISPLAY'	Internal display collating sequence.
'INTBCD'	Internal BCD collating sequence.
seqname	Name of a collating sequence specified in a call to SMSEQ (see text).

order — Character expression specifying order of sort processing;

'A'	Ascending (default).
'D'	Descending.

Figure 8-10. SMKEY Call

SMSEQ

SMSEQ (figure 8-11) specifies a user's collating sequence, or redefines the default to be a user collating sequence or a standard collating sequence other than the system default.

CALL SMSEQ(seqname,seqspec)

seqname — Name of user-supplied collating sequence.

seqspec — Name of integer array, terminated with a negative number, containing entire sequence of characters in order of collation.

Figure 8-11. SMSEQ Call

The characters in seqspec can be specified as their octal equivalents in the form O"ij" or as Hollerith constants in the form R"x". Characters to collate equal are specified in a call to SMEQU. Unspecified characters collate high (following the last character specified in seqspec) and equal.

SMEQU

SMEQU (figure 8-12) specifies that two or more characters in the collating sequence are equal for comparison purposes.

CALL SMEQU(colseq,equspec)

colseq — Collating sequence determined by a previous call to SMKEY (and perhaps SMSEQ).

equspec — Name of an integer array, terminated with a negative number, containing characters to collate equal to the last character, which must be included in colseq.

Figure 8-12. SMEQU Call

SMOPT

SMOPT (figure 8-13) specifies special record handling options. If SMOPT is called on SCOPE 2, it must be done immediately after the call to SMSORT or SMMERGE.

SMTAPE

SMTAPE (figure 8-14) specifies tape files to be used in balanced or polyphase tape merge. SMTAPE is not available on SCOPE 2. The file names in taplist must not be declared in the PROGRAM statement. A balanced merge requires a minimum of four tapes; a polyphase merge requires a minimum of three tapes.

SMOWN

SMOWN (figure 8-15) specifies owncode exits to be used during Sort/Merge processing.

Each subname specified in a call to SMOWN must appear in an EXTERNAL statement in the calling program. For each subname specified, the user must supply a subroutine which exits through a call to system subroutine SMRTN, in accordance with the owncode exit number and return address as shown in table 8-1.

No parameters are needed on SUBROUTINE subname for exit number 1 if there are no input files.

SMEND

SMEND (figure 8-16) initiates execution of the sort or merge. SMEND is required as the last in a series of Sort/Merge interface subroutines.

SMABT

SMABT (figure 8-17) terminates a sequence of Sort/Merge interface calls without initiating execution of Sort/Merge. The state of the interface is the same as if no calls had been made.

CALL SMTAPE(taplist)

taplist List of logical file names, each in the form L"filename", to be used in balanced or polyphase tape merge.

Figure 8-14. SMTAPE Call

CALL SMOWN(exitnum,subname[,exitnum,subname]. . .)

exitnum Number of the owncode exit.

subname Name of the user-supplied owncode exit subroutine.

Figure 8-15. SMOWN Call

CALL SMEND

Figure 8-16. SMEND Call

CALL SMABT

Figure 8-17. SMABT Call

CALL SMOPT(opt[,opt]. . .)

opt Nonordered options separated by commas:

'VERIFY'	Check output for correct sequencing (important for insertions during output and merge input).
'RETAIN'	Retain records with identical sort keys in order of appearance on input file.
'VOLDUMP'†	Checkpoint dump at end-of-volume.
'DUMP'†	Checkpoint dump after 50 000 records.
'DUMP',n†	Checkpoint dump after n (decimal) records.
'NODUMP'†	No checkpoint dumps.
'NODAY'†	Suppress dayfile messages.
'ORDER',mo†	Merge order = mo (default: mo = 5).
'COMPARE'	The key comparison sorting technique is to be used.
'EXTRACT'	The key extraction sorting technique is to be used.

'COMPARE' and 'EXTRACT' are mutually exclusive. If both are omitted, Sort/Merge decides which to use. 'COMPARE' usually decreases elapsed time while increasing central processor time, whereas 'EXTRACT' usually decreases central processor time while increasing elapsed time.

†Does not apply to SCOPE 2.

Figure 8-13. SMOPT Call

TABLE 8-1. OWNCODE EXIT NUMBERS

exitnum	entry	exit
1 or 3	SUBROUTINE subname (a,rl)	CALL SMRTN (retaddr), for retaddr = 1 or 3 CALL SMRTN (retaddr,b,rl), for retaddr = 0 or 2
2 or 4	SUBROUTINE subname	CALL SMRTN (retaddr), for retaddr = 0 CALL SMRTN (retaddr,b,rl), for retaddr = 1
5	SUBROUTINE subname (a_1,rl_1,a_2,rl_2)	CALL SMRTN (b_1,rl_1,b_2,rl_2), for retaddr = 0 CALL SMRTN (b_1,rl_1) for retaddr = 1

retaddr Return address:

0 Normal return address
1 Normal return address + 1
2 Normal return address + 2
3 Normal return address + 3

a Integer array of length (rl +9)/10 in which Sort/Merge stores a record when subname is called. b should not be the same as a. Storing into
b a causes indeterminate results.

rl Record length in characters.

INTERMIXED COMPASS SUBPROGRAMS

NOTE

Because of anticipated changes to this product, use of this feature is not recommended. For guidelines, see appendix G.

Subprograms in COMPASS assembly language can be intermixed with FORTRAN coded subprograms in the source deck. Intermixed COMPASS subprograms must begin with a source line containing the word IDENT in columns 11 through 15, with columns 2 through 10 blank, and column 16 blank, as illustrated in figure 8-18.

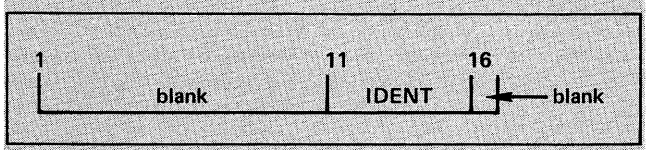


Figure 8-18. IDENT Statement

The subprogram ends with any legal COMPASS END line. A COMPASS subprogram cannot interrupt a FORTRAN program unit. An IDENT line must immediately follow a previous END line or be the first line in the record. A COMPASS subprogram can also be the first or last program unit in a source deck.

Both subroutines and functions written in COMPASS can be called from a FORTRAN source program. For either, register A0 is the only register that must be restored to its initial condition before the subprogram returns control to the calling routine.

If the COMPASS subprogram changes the value of A0, it must restore the initial contents of A0 upon returning control to the calling subprogram. When the COMPASS subprogram is entered by a noncharacter function reference, the subprogram must return the function result in X6 or X6 and X7 with the less significant or imaginary part of the double precision or complex result appearing in X7.

When a FORTRAN-generated subprogram is called, the calling routine must not depend on values being preserved in any registers other than A0.

See the COMPASS reference manual for more details on systems texts.

An example of a simple COMPASS function and the calling FORTRAN main program is illustrated in figure 8-19. The parity function, PF, returns an integer value; therefore, it must be declared type integer in the calling program. The argument to PF can be either real, integer, or Boolean.

The title and comments are unnecessary; they are included to encourage good programming practice. The following is a recommended convention:

```
PF  EQ  *+1S17  ENTRY/EXIT
```

This statement causes a jump to 400000_8 plus the location of the entry point of the routine if the function is not entered with a return jump. This results in a mode error that can quickly be identified. Since A0 is not used in this subprogram, it need not be restored.

SUBPROGRAM LINKAGE

Two methods of passing arguments to subprograms are used by the FORTRAN compiler. These methods are pass by reference and pass by value. Their use depends on the type of subprogram being called.

Sample Program:

```
      PROGRAM NPSAMP(OUTPUT)
      INTEGER PF, PVAL(24)
      DO 1 I = 1,24
1     PVAL(I)= PF(I)
      PRINT 2, (I,I=1,24), PVAL
2     FORMAT ('0 INTEGERS AND THEIR PARITY BELOW', /(24I3))
      STOP
      END
          IDENT  PF
          ENTRY  PF
PF        TITLE PF - COMPUTE PARITY OF WORD.
          COMMENT    COMPUTE PARITY OF WORD.
PF        SPACE  4,11
***       PF -   COMPUTE PARITY OF WORD.
*
*         FORTRAN SOURCE CALL --
*
*                PARITY = PF (ARG)
*
*         RESULT = 1  IFF ARG HAS ODD NUMBER OF BITS SET.
*                = 0  OTHERWISE.
*
**        ENTRY  (X1) = ADDRESS OF ARGUMENT.
*         EXIT   (X6) = RESULT.
PF        EQ     *+1S17
          SA2    X1
          CX3    X2
          MX0    -1
          BX6    -X0*X3
          EQ     PF            EXIT..
          END
```

Output:

```
 INTEGERS AND THEIR PARITY BELOW
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 1  1  0  1  0  0  1  1  0  0  1  0  1  1  0  1  0  0  1  0  1  1  0  0
```

Figure 8-19. Intermixed COMPASS Code

PASS BY REFERENCE SEQUENCE

The FORTRAN compiler uses the pass by reference sequence for the following references:

References to user-defined functions.

References to intrinsic functions when DB = TB is specified on the FTN5 control statement.

References to any subroutine.

The pass by reference sequence generated is as follows:

```
    SA1   list-address
+   RJ    subprogram-name
-   VFD   12/lnr
-   VFD   18/TRACE.
```

List-address is the address of the argument list; this instruction is not present if the call has no arguments and the ARG=FIXED option is specified on the FTN5 control statement.

Subprogram-name is the name of the subprogram being called. If an intrinsic function is being called, then the name is suffixed with an equal sign.

lnr is the source line number of the statement containing the calls.

TRACE. is the address of the traceback word in the calling routine.

Arguments in the call must correspond with the argument usage in the called routine, and they must reside in the same memory level. An actual argument of level 1 or 2, however, can correspond to a dummy argument of level 0.

The argument list consists of consecutive words having the format given in table 8-2. If ARG=FIXED is not selected, a zero word terminates the list. Statement label actual arguments appear at the end of the list, preceded by a generated-label (fall through address) parameter, and have a type of label (6 in bits 48 through 50 of argument list word) if ARG=COMMON is specified. The called routine accesses the addresses by offsetting the address of the argument list, which is contained in register A0.

TABLE 8-2. ARGUMENT LIST FORMAT

Bit Position	Description
59	Memory where argument resides: 0 Central memory or SCM. 1 Extended memory (ECS or LCM).
58-51	Zero.
50-48	Data type (present only when ARG=COMMON specified): 0 Real, Boolean, Intrinsic 1 Integer. 2 Double precision. 3 Complex. 4 Logical. 5 Character. 6 Label.
47-30	Argument length, in characters. Zero if argument is noncharacter.
29-28	Zero.
27-24	Beginning character position of argument. Value is 0 through 9: 0 means the leftmost character position of a word, 9 means the rightmost character position. 0 if argument is noncharacter.
23-0	First word address of argument.

PASS BY VALUE SEQUENCE

For increased efficiency, the compiler generates a more compact code sequence known as pass by value. The pass by value sequence is used for reference to mathematical intrinsic functions when DB=TB is not specified on the FTN5 control statement.

The pass by value code sequence consists of instructions to load the first argument into registers X1 and X2, and the second argument into registers X3 and X4 (the second register of each pair is used only if the argument is type double precision or complex) followed by a return jump instruction. The external linkage name for external functions called by value consists of the function name suffixed with a period.

FUNCTION RESULT

In both calling sequences, the result value of a noncharacter function is returned in register X6 (X6 and X7 for type double precision or complex). For functions whose result is type character, however, the compiler inserts an extra word at the beginning of the argument list to transmit the result. This word has the format shown in table 8-2.

ENTRY POINT

For subprograms written in FORTRAN, the entry point of the subprogram (for reference by an RJ instruction) is preceded by two words. The first is a trace word for the subprogram; it contains the subprogram name in left-justified display code (blank-filled) in the upper 42 bits and the subprogram entry address in the lower 18 bits. The second word is used to save the contents of A0 upon entry to the subprogram. The subprogram restores A0 upon exit. The code generated is as follows:

```
Trace word:     VFD     42/name, 18/entry address

A0 word:        DATA    0

Entry point:    DATA    0
```

RESTRICTIONS ON USING INTRINSIC FUNCTION NAMES

Functions written in FORTRAN that have names identical to the mathematical intrinsic function names described in section 7, such as AMAX1 or SQRT, must be declared EXTERNAL in the calling program unit. This declaration causes the compiler to reference the user-defined function, using a pass by reference sequence.

Functions written in COMPASS that have intrinsic function names can use either a pass by reference or a pass by value sequence. An EXTERNAL declaration and a pass by reference sequence must be used in the following cases:

If the function is to be passed as an argument to a subprogram.

If the function has the same name as an intrinsic function that generates in-line code (If the function is not declared EXTERNAL, the compiler generates in-line code).

If a COMPASS routine has the same name as an intrinsic function and pass by value is to be used, a period must be appended to the function name (for example, SIN.).

Figure 8-20 illustrates a sample program containing a call to the intrinsic function SQRT, a call to an external function ZEUS, and a reference to an intrinsic function, AMAX1, that generates in-line code. The object code generated by these calls is shown in figure 8-21. The code generated for the external function ZEUS is illustrated in figure 8-22.

```
      PROGRAM SUBLNK
      X=SQRT(7.0)
      Y=ZEUS(X,1.0)
      END
      FUNCTION ZEUS(ARG1,ARG2)
      ZEUS=AMAX1(ARG1,ARG2,0.)
      RETURN
      END
```

Figure 8-20. Program SUBLNK and Function ZEUS

```
        BLOCK          ADDRESS     LENGTH
        START.              0B         6B
        CODE.               6B        10B
        LITERL.            16B         2B
        FORMAT.            20B         0B
        TEMPS.             20B         0B
        APLST.             20B         4B
        IOAPL.             24B         0B
        NAMLST.            24B         0B
        VARS.              24B         4B
        SUB.               30B         0B
        SUBO.              30B         0B
        BUFER.             30B         0B

                                                          IDENT   SUBLNK
   0B   23250214161355000005 +                  SUBLNK    TRACE.
   2B                                           FILVEC.   BSS     0
   2B   00000000000001000000                              ADDR    0,1
   3B   00000000000000011610                              FVEC    11610B
                                                          USE     LITERL.
  16B   17227000000000000000                    CON.      CON        17227000000000000000B } constant table
  17B   17204000000000000000                              CON        17204000000000000000B }
  17B                                                     USE     FORMAT.
                                                          USE     APLST.          }
  20B                                           AP.1      BSS     0               }
  20B   00000000000000000026                              APL     0003,X+0        }
  21B   00000000000000000017                              APL     0003,CON.+1     } parameter list
  22B   00000000000000000000                              0                       }
  23B                                           AP.2      BSS     0               }
  23B   00000000000000000000                              APL     0000,TRACE.+0   }
                                                          USE     IOAPL.
                                                          USE     NAMLST.
                                                          USE     CODE.
                                                *                    LINE 2 <---- source line number
   6B   6102000002                                        SB0     B2+0+2
                  5110000016+                             SA1     CON. <---- get actual parameter into X1
   7B   0100000000                                        RJ      =XSQRT.
  10B   5160000026           +                            SA6     X
                                                *                    LINE 3
  11B   6102000003                                        SB0     B2+0+3
                  5110000020+                             SA1     AP.1 <---- get address of parameter list into A1
  12B   01000000000003000000                              RJT     =XZEUS,3
  13B   5160000027           +                            SA6     Y
                                                *                    LINE 4
  14B   6102000004                                        SB0     B2+0+4
                  5110000023+                             SA1     AP.2
  15B   01000000000004000000                              RJT     =XEND5.,4
  16B                                                     BSS     0
  16B                                                     USE     START.
   4B   6102777747                                        SB0     B2+0-30B
                  6102000000+                             SB0     B2+TRACE.
   5B                                           SUBLNK    BSS     0
   5B   5110000002           +                            SA1     FILVEC.
   5B             0100000000                              RJ      =XQ5RPV.
   6B                                                     BSS     0
   6B                                                     USE     TEMPS.
  20B                                           ST.       BSS     0
  20B                                           CT.       BSS     0
  20B                                           IT.       BSS     0
  20B                                           OT.       BSS     0
  20B                                           VD.       BSS     0
  20B                                           LC.       BSS     0
                                                          USE     BUFER.
  30B                                           LENP.     EQUN
                                                          END     SUBLNK
```

Figure 8-21. Object Listing for Program SUBLNK

```
      BLOCK          ADDRESS      LENGTH
      START.              0B         10B
      CODE.              10B         10B
      LITERL.            20B          0B
      FORMAT.            20B          0B
      TEMPS.             20B          0B
      APLST.             20B          0B
      IOAPL.             20B          0B
      NAMLST.            20B          0B
      VARS.              20B          3B        ——name of program unit and entry point address
      SUB.               23B          0B
      SUBO.              23B          0B
      BUFER.             23B          0B

                                                    IDENT   ZEUS
 0B   32052523555555000006 +          ZEUS          TRACE.
 2B                                   SAVEA1.       BSS     0
 2B   00000000000000000000                          0
 3B                                                 USE     LITERL.
                                                    USE     FORMAT.
                                                    USE     APLST.
                                                    USE     IOAPL.
                                                    USE     NAMLST.
                                                    USE     CODE.
                                      *                             LINE 2
10B   6102000002                                    SB0     B2+0+2
                13111                               BX1     X1-X1
11B   5020000001                                    SA2     A0+0+1
                53220                               SA2     X2
                      31021                         FX0     X2-X1
12B   21073                                         AX0     73B
           13121                                    BX1     X2-X1
                11310                               BX3     X1*X0
                      13032                         BX0     X3-X2
13B   54300                                         SA3     A0
           53330                                    SA3     X3
                31430                               FX4     X3-X0
                      21473                         AX4     73B
14B   13530                                         BX5     X3-X0
           11054                                    BX0     X5*X4
                13703                               BX7     X0-X3
15B   5170000020           +                        SA7     VALUR.
                                      *                             LINE 3
16B   6102000003                                    SB0     B2+0+3
16B             0400000004+                         EQ      EXIT.
                                      *                             LINE 4
17B   6102000004                                    SB0     B2+0+4
20B                                                 USE     TEMPS.
20B                                   SUBOI.        BSS     0
20B                                                 BSS     0
20B                                                 USE     START.
 3B   6102777754                                    SB0     B2+0-23B
                6102000000+                         SB0     B2+TRACE.
 4B                                   EXIT.         BSS     0
 4B   5120000001           +                        SA2     TEMPA0.  }
                53020                               SA0     X2       } restores A0 on exit
 5B   5140000020           +                        SA4     VALUE.   }
                10644                               BX6     X4     } entry point
 6B                                   ZEUS          BSS     0      }
 6B   0400000006           +                        EQ      ZEUS
 7B   74600                                         SX6     A0       }
           54010                                    SA0     A1       } saves A0 and sets A0 to the new A1
                5160000001+                         SA6     TEMPA0.  }
10B                                                 BSS     0
10B                                                 USE     TEMPS.
20B                                   ST.           BSS     0
20B                                   CT.           BSS     0
20B                                   IT.           BSS     0
20B                                   OT.           BSS     0
20B                                   VD.           BSS     0
20B                                   LC.           BSS     0
                                                    USE     BUFER.
23B                                   LENP.         EQUN
                                                    END
```

Figure 8-22. Object Listing for Function ZEUS

NOTE

Refer to appendix G for recommendations on the use of this feature.

To reduce the amount of storage required, and to make efficient use of the available field length, the programmer can divide a program into overlays. Other methods of dividing large programs include segmentation, capsules and overlay capsules. These are described in the CYBER Loader reference manual for NOS and NOS/BE, and the SCOPE 2 Loader reference manual for SCOPE 2.

OVERLAYS

Each overlay is an executable program, and the overlays are a collection of programs combined into an overlay structure. Before program execution, the object modules of an overlay program are linked by the loader and placed on a storage device (mass storage or tape) in their absolute form. Overlays are loaded at execution time without relocation; no linking is required because the linking has already been done. (For more information, see the appropriate loader reference manual.) As a result, the size of the resident loader for overlays can be substantially reduced. Overlays can be used when the organization of the program in memory can be defined prior to execution.

When each overlay is generated, the loader includes library and user subprograms and links them together. The generated overlay is in fixed format, with internal references fixed in their relationship to one another. The generated overlay has a fixed origin address within the field length and is not relocatable. At execution time, the loader simply reads the required overlay from the overlay file and loads the overlay at the preestablished origin within the field length for the user program.

MAIN, PRIMARY, AND SECONDARY OVERLAYS

Overlays are loaded into memory at three levels. The general positioning of main, primary, and secondary level overlays is shown in figure 9-1.

Overlays are identified by a pair of integers, in the following way:

(0,0)	Main overlay
(n,0)	Primary overlay
(n,k)	Secondary overlay

The values n and k are positive octal integers in the range 1 through 77_8. For any given program execution, all overlay identifiers must be unique. For example, (1,0),

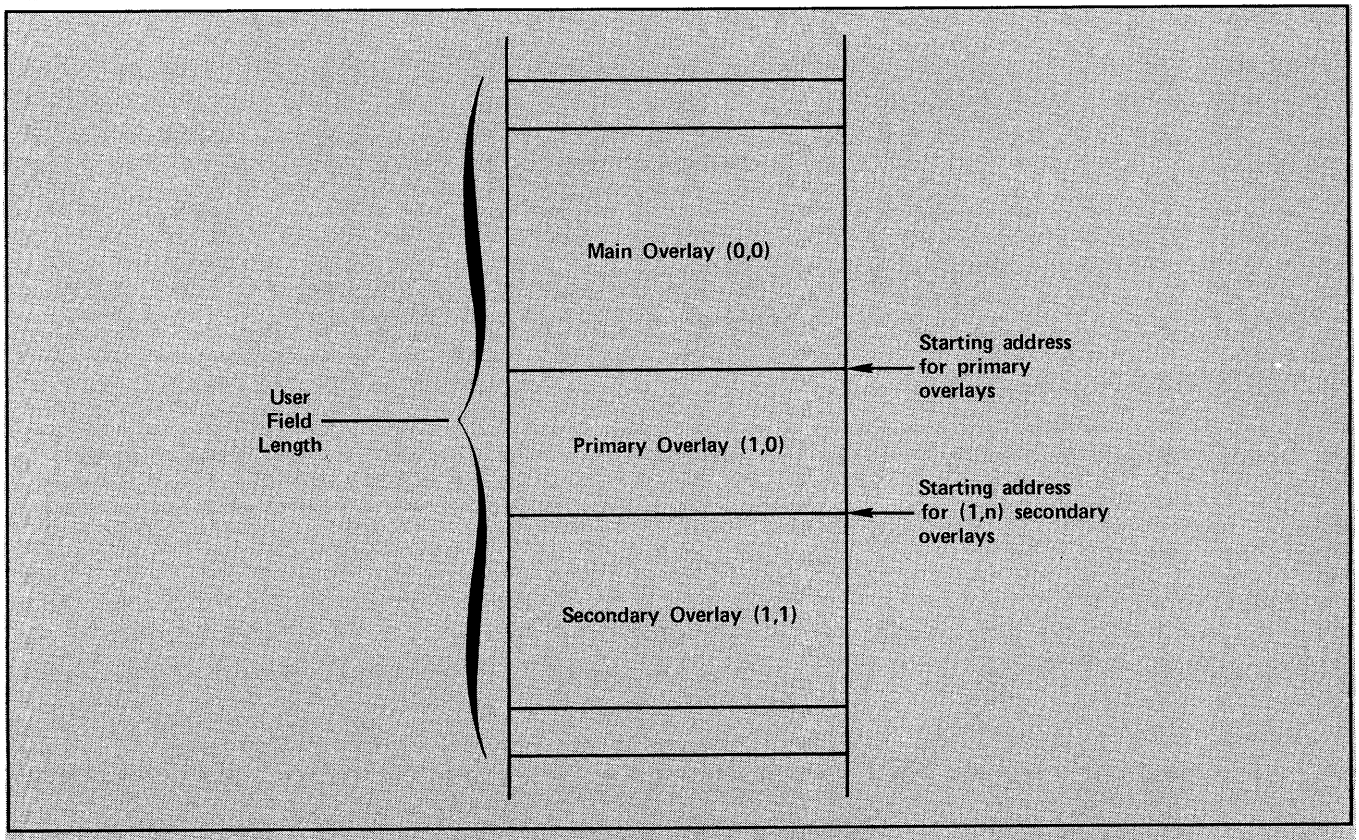


Figure 9-1. Overlay Positioning

(2,0), (3,0), and (4,0) would be primary overlays; and (3,1), (3,2), (3,5), and (3,7) would be secondary overlays associated with primary overlay (3,0). Secondary overlays are identified by the primary overlay number and a nonzero secondary number. Overlay numbers of the form (0,n) with n greater than zero are not valid.

The main or zero overlay is loaded first and remains in memory at all times. A primary overlay can be loaded immediately following the zero overlay, and a secondary overlay can be loaded immediately following the primary overlay. Overlays can be replaced by other overlays. For example, if a different secondary overlay is required, the loader simply moves it from the overlay file into memory, placing it at the same starting address as the previously loaded secondary overlay.

When a primary overlay is loaded, the previously loaded primary overlay and any associated secondary overlays are destroyed. For this reason, no primary overlay can load other primary overlays. In the same way, loading a secondary overlay destroys a previously loaded secondary overlay.

A secondary overlay can be called into memory only by its primary overlay. For example, overlay (1,0) can call overlay (1,2), but overlay (2,0) cannot call overlay (1,2). Execution is faster if the more commonly used subprograms are placed in the main overlay, which remains in memory at all times. The less commonly used subprograms can be placed in primary or secondary overlays that are called into memory as required.

An overlay can consist of one or more FORTRAN or COMPASS program units. Each overlay must contain one FORTRAN main program, but the FORTRAN main program need not be the first program unit in the 60481300 B overlay. When the overlay is called, the program name in the PROGRAM statement becomes the primary entry point for the overlay.

OVERLAY COMMUNICATION

Data is passed between overlays through blank common or labeled common. An element of a blank or labeled common block in the main overlay (0,0) can be referenced by any higher (primary or secondary) level overlay. Any blank or labeled common block declared in a primary overlay can be referenced by the primary overlay and the associated secondary overlays, but not by the main overlay.

Blank common is located at the end, that is, the highest address of the first overlay in which blank common is declared. An example is shown in figure 9-2.

In the example, blank common is declared in the main (0,0) overlay. Blank common is located at the end of the (0,0) overlay and is accessible to all other overlays. If blank common is declared in the (1,0) overlay, blank common is at the end of the (1,0) overlay and is accessible only to the associated (1,k) secondary overlays.

Labeled common blocks are generated in the overlay in which they are first encountered. Data in a labeled common block can only be preset in the overlay in which the labeled common block is generated.

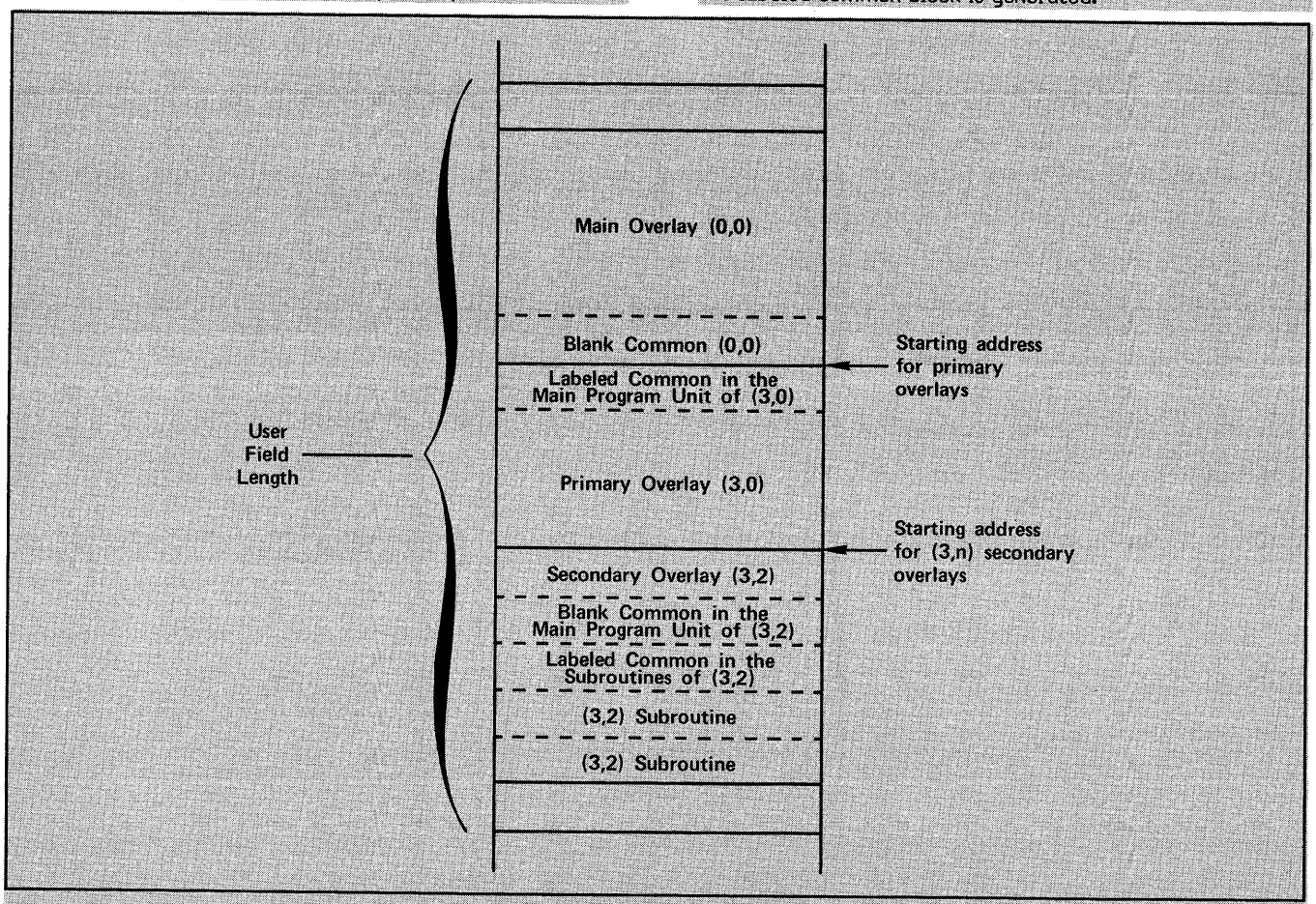


Figure 9-2. Overlay Positioning Showing Common

In the example, the labeled common declared in the main program unit of primary overlay (3,0) is at the bottom of the primary overlay. For the secondary overlay (3,2), the blank common in the main program unit is at the end of the main program part of the overlay. The labeled common declared for the two subroutines precedes the subroutines.

CREATING OVERLAYS

An overlay is identified by an OVERLAY statement that precedes the PROGRAM statement in each overlay. An overlay consists of all program units between the OVERLAY statement and the next OVERLAY statement, or the end of the source input.

The OVERLAY statement is shown in figure 9-3.

The OVERLAY statement cannot be continued. The required order of the OVERLAY statements, and the requirements for fname, i, j, orig, and OV=m are described in greater detail in the appropriate loader reference manual.

Loading overlays from a file requires an end-around search of the file for the specified overlay, which can be time-consuming in large files. If speed is essential, the overlays should be called in the same order in which they were generated, or fast overlay loading should be used. (See the appropriate loader reference manual.) The group of relocatable decks to be processed by the loader to create an overlay-structured program must be presented to the loader in a specific order. The main overlay must be loaded first. Any primary group is followed by its associated secondary group, then any other primary group is followed by its associated secondary group, and so forth.

The first OVERLAY statement must have a file name and i,j must be 0,0 for the main overlay. Subsequent statements can omit file name, indicating that the overlays are to be written on the same file. The second OVERLAY statement must be of a primary overlay such as (3,0). If the orig parameter is omitted, the overlay is loaded in the normal way directly after the main overlay. The orig parameter cannot be included on the main OVERLAY statement, but is used on primary and secondary overlay statements to change the size of blank common at overlay generation time.

CALLING OVERLAYS

Only the main overlay (0,0) is loaded when the control statement that calls the main overlay is encountered. Primary and secondary overlays are called with the OVERLAY call shown in figure 9-4.

If the recall parameter is specified, the overlay is not reloaded if it is already in memory. If the overlay is already in memory and the recall parameter is not used, the overlay is actually reloaded, thereby changing the values of variables in the overlay.

When a RETURN or END statement is encountered in the main program of the main overlay, execution of the program terminates and control returns to the operating system. When either of the statements is encountered in the main program of a primary or secondary overlay, control returns to the next executable statement after the CALL OVERLAY statement that invoked the current overlay.

OVERLAY ([fname,] i,j[,orig] [,OV=m])

fname	Is the name of the file on which the generated overlay is to be written.	
i,j	Are the overlay level numbers in octal (0 through 77). The numbers specified are not checked or converted by FORTRAN.	
orig	Is an optional parameter specifying the origin of the overlay; not allowed for (0,0) overlay. The loader accepts any of the following forms:	
	Cnnnnnn	The overlay is loaded nnnnnn words from the start of blank common; nnnnnn must be an octal number (0 through 777777).
	O=nnnnnn	The overlay is loaded at the address specified; nnnnnn must be an octal number $\geq 110_8$.
	O=ept	The overlay is loaded at the address of the entry point specified, which must have been declared in a lower level overlay.
	O=ept±nnnnnn	The overlay is loaded at the address of the entry point specified, but the address is biased by the amount of the offset nnnnnn.
	OV=m	Is an optional parameter specifying a decimal number for the total number of higher level overlays in the overlay structure. The OV parameter is valid only for the (0,0)† overlay and causes the overlay generator and loader to use Fast Overlay Loading† (FOL). See the CYBER Loader reference manual.

†Not valid on SCOPE 2.

Figure 9-3. OVERLAY Statement

An example is shown in figure 9-5.

Execution of RETURN in the (1,1) overlay returns control to the statement in the (1,0) overlay following the (1,1) call. Execution of RETURN in the 1,0 overlay returns control to the statement in the main overlay following the (1,0) call.

CALL OVERLAY (fname,i,j[,recall[,k]])

fname — Is either the file name of the file containing the (i,j) overlay to be executed (if k is absent or zero), or the overlay name (if k is nonzero). The name can be either:

A Boolean expression with the value nHf, where f is the name of the file.

An arithmetic expression with the value nHf, after conversion BOOL(fname).

A character expression containing the name of the file, after trailing blanks are removed.

i,j — Are the overlay level numbers and can be integer or Boolean expressions. The values are converted as necessary to INT(i) and INT(j).

recall — Is the recall parameter and can be:

A Boolean expression with the value 6HRECALL.

An arithmetic expression with the value 6HRECALL, after the conversion BOOL(recall).

A character expression with the value 'RECALL', after trailing blanks are removed.

k — Is an indicator affecting the interpretation of fname. If k is nonzero, fname is an overlay in a library. See the appropriate loader reference manual.

Figure 9-4. OVERLAY Call

OVCAPS

The use of overlay capsules is another method for reducing the amount of storage required for running large programs. Overlay capsules, commonly called OVCAPs, provide a similar capability to overlays. The difference is that OVCAPs are not specified by levels and there exists no limitation on the number of OVCAPs that can be present in memory at any one time. OVCAPs are not supported on SCOPE 2.

An overlay has a fixed first word address (FWA), an OVCAP has no such limitation. An OVCAP is loaded into memory within a block that is made available by Common Memory Manager (CMM). This allows user code to co-exist, in the CMM-managed portion of memory, with other capsules that can belong to the user or to system routines. By making use of the user-callable CMM routines, the user can manage memory very efficiently by utilizing OVCAPs instead of overlays.

Data communication between OVCAPs is carried out similiar to data communication between overlays. An OVCAP, as defined by the loader, is a logical extension of an (0,0) overlay. OVCAPs require a (0,0) overlay to be present; communication is via common blocks specified in the main overlay and in the capsule. Any common blocks that exist in an OVCAP, but not in the (0,0) overlay, are processed as local to the OVCAP.

```
      OVERLAY(XFILE,0,0,OV=2)
      PROGRAM ONE
      .
      .
      .
      CALL OVERLAY(5HXFILE,1,0,0)
      .
      .
      .
      STOP
      END
      OVERLAY(XFILE,1,0)
      PROGRAM ONE ZERO
      CALL OVERLAY(5HXFILE,1,1,0)
      .
      .
      .
      RETURN
      END
      OVERLAY(XFILE,1,1)
      PROGRAM ONE ONE
      .
      .
      .
      RETURN
      END
```

Figure 9-5. Sample Overlay Structure

An OVCAP cannot contain references to data resident in ECS or LCM. If a user places an intermixed COMPASS routine in an OVCAP, no non-standard relocation is permitted (standard relocation is the relocation of 18 bit fields whose rightmost bit is bit 0,15 or bit 30 of a central memory word).

CREATING OVCAPS

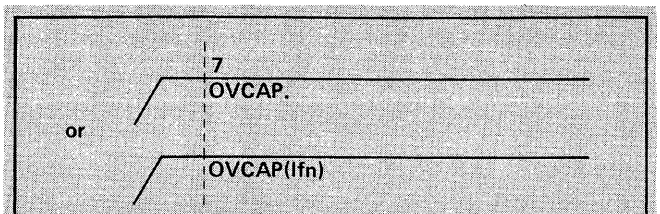
An OVCAP consists of one or more program units. The first program unit specifies the OVCAP name. Unlike overlays, the first program unit must be a subroutine and not a main program; no main programs are permitted in OVCAPs.

An OVCAP is identified by an OVCAP directive placed before the first subroutine of each OVCAP. The OVCAP consists of all program units between the OVCAP statement and the next OVCAP statement, or at the end of source input. The name of the OVCAP is the name of the first subroutine in that OVCAP, which must immediately follow the OVCAP statement. The first subroutine must be a subroutine subprogram without alternate entry points. The format of the OVCAP directive is shown in figure 9-6.

OVCAP directives are placed in the source input stream in the same manner as overlay directives. The only exception is that at least one overlay directive, and all overlay directives for the current overlay structure, must precede OVCAP directives. The first directive of the source input stream must be a (0,0) overlay directive whether or not any other overlays are to be created.

An OVCAP generation load sequence must be terminated by a NOGO directive, or a fatal loader error is generated.

OVCAPs cannot exist in a STATIC environment. All OVCAPs must be generated in the same load sequence.



lfn Is the logical file name of the file on which the OVCAP binary is to be written. If lfn is coded on a NOGO control statement, the file named on the NOGO statement overrides the file named on the OVCAP directive. If no file name is specified on either a NOGO statement or the OVCAP directive, the output is written to file ABS; in which case, the overlay lfn must be ABS also. When a file name is taken from an OVCAP directive, the file name remains in effect until a new file name is specified on a subsequent directive.

Figure 9-6. Format of an OVCAP Directive

LOADING AND UNLOADING OVCAPS

The FTN5 compiler supports three types of call for manipulation of OVCAPs:

CALL LOVCAP (name)	Load OVCAP name.
CALL XOVCAP (name)	Execute OVCAP name.
CALL UOVCAP (name)	Unload OVCAP name.

- (name) is a variable or constant of type character.
- CALL LOVCAP (name) causes the specified OVCAP to be loaded into memory. Execution is not initiated. OVCAPs are loaded from the same place as the main overlay; this can be a file or a library (library loading of OVCAPs is not supported on NOS).
- CALL XOVCAP (name, user parameter list) initiates execution of the OVCAP specified. If the OVCAP has not been loaded prior to this call, it is first loaded and then executed. The user parameters are passed to the subroutine entered in the OVCAP.
- CALL UOVCAP (name) unloads the specified OVCAP.

Any number of OVCAPs can be in memory at any time. Any OVCAP can load and execute any other OVCAP. An OVCAP must exit through the entry point of the first subroutine to return to the statement after the statement that initiated execution of the OVCAP. Any OVCAP can unload any other OVCAP. Be sure that no OVCAP unloads an OVCAP that was called in the chain prior to the execution of the present OVCAP. Undefined results occur if this is attempted.

CID cannot be used to debug OVCAPs. If OVCAPs are present during a PMD run, they are ignored.

If execution of an OVCAP is initiated by an overlay other than the (0,0) overlay, that overlay must not be replaced by an overlay of the same level until the called OVCAP has returned to the statement that initiated its execution.

A batch job set up to compile and execute a user job containing OVCAPs is shown in figure 9-7.

```
JOB.
USER(TOM,DICK)
CHARGE(1234,56789)
FTN5.
LOAD(LGO)
NOGO(CAPS)
CAPS.
7/8/9(multi-punch)
      OVERLAY (FRUIT,0,0)
      PROGRAM APPLE
      .
      .
      .
      CALL XOVCAP (PEACH,A,3)
      .
      .
      .
      CALL UOVCAP (PEACH)
      CALL UOVCAP (PEAR)
      CALL UOVCAP (BERRY)
      .
      .
      .
      END
      OVCAP
      SUBROUTINE PEACH (X,I)
      .
      .
      .
      CALL LOVCAP (PEAR)
      CALL LOVCAP (BERRY)
      .
      .
      .
      CALL XOVCAP (PEAR)
      END
      OVCAP
      SUBROUTINE BERRY
      .
      .
      .
      END
      OVCAP
      SUBROUTINE PEAR
      .
      .
      .
      CALL XOVCAP (BERRY)
      .
      .
      .
      END
6/7/8/9 (multi-punch)
```

Figure 9-7. Batch Job Set Up for OVCAPS

Two debugging aids are available to help the user find execution-time errors in a FORTRAN program. These are the CYBER Interactive Debug facility (CID) and the Post Mortem Dump facility. CID is intended primarily for interactive use, whereas the Post Mortem Dump facility is intended mainly for use with batch jobs. The Post Mortem Dump facility is available under all operating systems. CID is available under NOS and NOS/BE, but is not available under SCOPE 2. Sample programs illustrating the use of CYBER Interactive Debug and Post Mortem Dump are included in section 12.

Only a brief summary of CID is presented in this section. Refer to the CYBER Interactive Debug reference manual for more detailed information.

CYBER INTERACTIVE DEBUG

CYBER Interactive Debug is a supervisory program that allows the user to monitor and control the execution of a FORTRAN program from a terminal. CID provides the capability to:

- Suspend program execution at specified locations called breakpoints.
- Suspend program execution on the occurrence of specified events such as the loading of an overlay. These events are called traps.
- Display or alter the contents of program variables while execution is suspended.
- Resume program execution at the point of suspension or at any other specified location.

No special statements are required in the source program to use CID. However, a special compilation is required if the symbolic address capabilities and special FORTRAN commands are to be used.

PROGRAM COMPILATION

To use all of the capabilities of CID the source program must be compiled in debug mode. This can be done either by specifying the DEBUG control statement prior to compilation or by specifying the DB=ID parameter on the FTN5 control statement. Note that a program that has not been compiled in debug mode can still be executed under CID control; however, program addresses cannot be referenced symbolically and the special FORTRAN commands cannot be used.

CID requires OPT=0 compilation.

DEBUG Control Statement

The DEBUG control statement activates debug mode. The format of this statement is:

```
DEBUG.
```

or

```
DEBUG(ON)
```

Debug mode can be subsequently turned off by the statement:

```
DEBUG(OFF)
```

When a source program is compiled in debug mode, the compiler produces a symbol table and a line number table along with the binary object code. CID accesses these tables during program execution to allow the user to reference program addresses symbolically.

To execute under control of CID, debug mode must be turned on prior to the program load. The CID module is loaded along with the compiled code and becomes part of the user's field length.

A program that has been compiled with DEBUG(ON) can be executed in a normal manner with DEBUG(OFF).

DB Parameter

To compile a program in debug mode, the DB=ID parameter can be specified on the FTN5 control statement (section 11). After DB compilation, the DEBUG(ON) control statement must be specified to execute the program under CID control. The DB=0 parameter overrides a previous DEBUG(ON) control statement for the duration of compilation. To use CID without the Post Mortem Dump facility, the DB parameter must be specified in the form DB=0/ID. An example of an FTN5 statement to compile a program in debug mode is:

```
FTN5,I=PROGA,DB=ID.
```

The source file PROGA is compiled, and written to file LGO along with debug tables.

INITIATING A DEBUG SESSION

To execute a program under CID control, the DEBUG(ON) control statement must be specified prior to initiating program execution. The debug session is then initiated by entering the name of the binary object file. Normally, this initiates execution of the user program. In debug mode, however, control transfers, not to the user program, but to an entry point in CID. CID responds with the message:

```
CYBER INTERACTIVE DEBUG
?
```

The ? is a prompt signifying that CID is waiting for user input. At this point the user can enter CID commands. Typically, the user takes this opportunity to set traps and breakpoints that will cause execution to be suspended, and then enters the command to begin execution of the program.

SOME CID COMMANDS

The following paragraphs introduce some CID commands that can be used to conduct a simple debug session.

GO Command

The GO command initiates or resumes execution of the user program. The format of this command is:

```
GO
```

Program execution begins at the location where it was suspended.

SET,BREAKPOINT Command

A breakpoint is a location within a program where execution is to be suspended. The format of the command to establish a breakpoint is:

```
SET,BREAKPOINT,L.n
```

where n is the FORTRAN source line number where the breakpoint is to be established. When CID encounters a breakpoint during program execution, control immediately transfers to CID which issues the message and prompt:

```
*B #n at L.n
?
```

At this point, the user can enter any CID command. In a typical session, commands are entered to display or change the current values of program variables.

SET,TRAP Command

A trap is a CID device that detects the occurrence of a specified condition during program execution, suspends execution at that point, and transfers control to CID. The format of the command to establish a trap is:

```
SET,TRAP,type,location
```

where type is the trap type and location is the trap location. CID provides several trap types. One of the most useful to the FORTRAN programmer is the LINE trap. This trap gives control to CID immediately prior to the execution of each FORTRAN line, allowing the user to step through the execution of a program one line at a time. An example of a command to set a LINE trap is:

```
SET,TRAP,LINE,P.PROGA
```

This command sets a trap to give control to CID immediately before execution of each line in program unit PROGA.

PRINT Command

The PRINT command prints the values of specified variables. This command is identical in form and function to the FORTRAN list directed PRINT statement. The format of the PRINT command is:

```
PRINT*,list
```

where list is a list of variable names separated by commas. The values of the variables are formatted according to the type declared in the source program. An example of the PRINT command is:

```
PRINT*, A,B,C(5)
```

This command prints the values of A, B, and the fifth element of array C.

The PRINT command can be issued any time in a debug session after CID has issued a ? prompt.

Assignment Command

The assignment command is used to assign new values to program variables. This command is identical in form and function to the FORTRAN assignment statement. The format of the assignment command is:

```
variable=expression
```

where expression is any valid FORTRAN expression not involving exponentiation or function references. The expression is evaluated and the value replaces the current value of the variable. Some examples of assignment commands are as follows:

```
A=1.0

DX=DY*DZ+1.0

F=(A+T)/(B+T)
```

QUIT Command

The QUIT command terminates a debug session. The format of this command is:

```
QUIT
```

Note that after a session is terminated by the QUIT command, debug mode remains on until DEBUG(OFF) is entered.

OTHER CID FEATURES

Other CID features include:

- A trap to suspend program execution when data is stored into a specified variable
- A trap to suspend program execution when an overlay is loaded
- The capability of defining sequences of CID commands to be executed automatically on the occurence of a trap or breakpoint
- The capability of saving trap and breakpoint definitions on a separate file
- The capability of interrupting a debug session

Refer to the CYBER Interactive Debug reference manual for descriptions of these and other CID features.

POST MORTEM DUMP

Post Mortem Dump is designed to analyze the cause of execution time errors in FORTRAN programs. Post Mortem Dump provides interpreted output in a form which is more easily understood than the octal dump normally output following a fatal error. Post Mortem Dump prints a readable summary of the error condition and the state of the program at the time of failure in terms of the names used in the original program. Thus, the names and values of the variables in the routine in which the error was detected are printed; this is repeated tracing back through the calling sequence of routines until the main program is reached.

Use of Post Mortem Dump (PMD) does not affect the use of CYBER Interactive Debug.

PMD does not require modification of the source program or the control statements, and PMD does not significantly affect the compiler-generated code. Thus, the user is free to use such compiler options as DB (debug) and OPT (optimization).

The compiler presets each address of memory to a negative indefinite value if PMD is selected; these values override any value specified elsewhere by the user. Specification of PMD does not significantly increase the execution time for jobs that terminate normally, nor does it significantly increase the user's run-time field length.

To use PMD, the DB=PMD parameter must be specified on the FTN5 control statement. PMD will then be activated by a fatal execution error or by one of the user-callable subroutines PMDLOAD or PMDSTOP. Information provided by the dump includes the following (where applicable):

A summary of all nonfatal loader errors

A list of all COMMON block length clashes

The nature of the error that activated PMD

The array-dumping parameters selected and the field length required to load and run the user program

The activity of each file used by the user program at the time of the error

The overlays in memory at the time of the error

The location of the error in terms of statement labels and line numbers, if possible

An annotated register dump for some system errors; an attempt is made to associate each address register with a variable or array referenced within the routine in which the error occurred

An alphabetic list of all variables and their values, accessible from the current routines

A printout of arrays according to specified parameters

A message-tracing call beginning at the previous routine and ending when the main program is reached

A completion message upon reaching the main program

Variables are printed in an alphabetically sorted list. The column labeled RELOCATION is left blank for local variables. It contains the block name for COMMON variables and reads F.P. nn for formal parameters, where nn indicates the parameter number as assigned by the compiler.

In addition to being printed as numbers, INTEGER variables are interpreted as masks or characters in H, L, or R FORMAT. In character representation, binary zeros are converted to blanks within a word, but a word with binary zeros at each end has the first binary zero printed as a colon.

The column headed COMMENTS flags undefined local variables as UNDEF, which indicates a potential source of error.

Variables passed as parameters to the previous routine in the traceback tree are labeled PARAM nn in the COMMENTS column. The COMMENTS column contains F.P. nn ,where the same variable occurs more than once in an argument string; nn points to the last occurrence. Constants passed to the previous routine are also printed at the end of the list and given the symbolic name CONSTANT. Untraceable functions and subroutines passed as arguments are printed.

Full checking is carried out on subroutine or function arguments and a warning message is issued if:

A routine is called with the wrong number of arguments.

A type conflict exists between actual and formal arguments.

The argument was a constant and the called routine either treated it as an array or corrupted it.

A conflict in the use of EXTERNAL arguments is detected; note that the results given for EXTERNAL arguments can be imprecise because several utilities can reside within the same routine and PMD cannot differentiate between them. For example, both SIN and COS reside within the routine SINCOS=.

A warning message is also issued if a real variable contains an unnormalized value, for example, an integer.

Variables that have never had a value assigned to them are printed as having the value NOT INITIALIZED. Variables that have not had storage assigned to them by the compiler because of optimization are printed as having the value NOT ASSIGNED STORAGE.

For batch jobs, the dump is written to the file OUTPUT. For jobs executed from an interactive terminal, the disposition of the dump is determined by options specified on the execution control statement (typically LGO) as follows:

LGO,*OP=option[option][option].

where option is one of the following:

T A condensed form of the dump is displayed at the terminal. This option is valid for interactive jobs only.

F A full dump is written on the file PMDUMP when the job is executed with the file OUTPUT connected. This option is valid for interactive jobs only.

A The variables in all active routines are included in the dump. An active routine is a routine that has been executed but is not necessarily in the traceback chain. This option is valid for batch, as well as interactive, jobs.

PMD can be used with overlay programs. In this case, only variables defined in the overlay currently in memory are dumped. The overlay numbers of the current overlay appear in the PMD output.

PMD suppresses any user-specified load map directive or MAP(ON) control statement. For example, the following statements do not produce a load map if DB=PMD was specified:

```
LDSET(MAP=SBEX)
LOAD(LGO)
EXECUTE.
```

However, the loader always writes a block-type map to file ZZZZZMP if DB=PMD was specified. The contents of this file can be printed by copying it to file OUTPUT. If nonfatal loader errors occur, a summary of the errors is included in the PMD output.

When DB=PMD is specified on the FTN5 control statement, the compiler generates a loader request to preset all memory to a special value for initialization testing. This preset is similar to that produced by the following load sequence:

```
LDSET(PRESETA=6000000000433400000)
LOAD(LGO)
EXECUTE.
```

Any user LDSET(PRESET=) loader specification is overridden.

Post Mortem Dump output produced by a program compiled under a given optimization level can differ from that produced by the same program compiled under a different optimization level. This occurs because different optimization levels generate different sequences of object code. Thus, at the actual time of an abort, the machine instruction being executed for a specified optimization level might be different from the instruction being executed for a different optimization level.

Variable values printed by the Post Mortem Dump might differ for successive executions of the same program on certain computer systems. This can occur on systems with multiple functional units such as the 6600, 6700, CYBER 70 Models 74 and 76, and the CYBER 170 Models 175 and 176.

The following paragraphs describe the individual Post Mortem Dump subroutine calls.

PMDARRY

The format of the call to PMDARRY is shown in figure 10-1.

CALL PMDARRY(i,j,k,l,m,n,o)

i through o are integers indicating the limits on the first through seventh array subscripts respectively.

Figure 10-1. PMDARRY Call

PMDARRY causes dumps of arrays to be limited to elements where subscripts do not exceed i, j, k, l, m, n, and o for their respective dimensions; the integers i through o represent the first through seventh dimensions respectively.

One through seven arguments can be passed to PMDARRY. If fewer than seven arguments are passed, dumps are limited to the arrays shown in table 10-1. PMDARRY produces no immediate output until a dump is forced by some other PMD call or by a fatal execution error.

TABLE 10-1. Post Mortem Dump Arrays

Arguments Specified in CALL PMDARRY	PMDARRY Action
i	Only 1-dimensional arrays are dumped.
i,j	Only 1- and 2-dimensional arrays are dumped.
i,j,k	Only 1- through 3-dimensional arrays are dumped.
i,j,k,l	Only 1- through 4-dimensional arrays are dumped.
i,j,k,l,m	Only 1- through 5-dimensional arrays are dumped.
i,j,k,l,m,n	Only 1- through 6-dimensional arrays are dumped.
i,j,k,l,m,n,o	1- through 7-dimensional arrays are dumped.

The following special form of the execution control statement performs the same function as a call to PMDARRY:

```
LGO,*DA=i+j+k+l+m+n+o.
```

where i through o represent the first through seventh dimensions respectively.

If the call to PMDARRY and LGO,*DA is omitted, the effect is the same as CALL PMDARRAY(20,2,1,1,1,1,1); for example, all arrays (1 through 7 dimensions) are dumped, but only elements whose subscripts do not exceed (20,2,1,1,1,1,1) are included in the dump.

Once PMDARRAY has been called, the established conditions apply to all program units in the user program. Any number of PMDARRAY calls can be included; the most recent call determines the effective conditions.

Example:

```
DIMENSION RAY(10,10,10)
.
.
.
CALL PMDARRY(3,4,1)
```

Array elements are printed with the first subscript varying fastest and with a maximum of six values per line for REAL, INTEGER, and LOGICAL arrays, and a maximum of three values per line for DOUBLE PRECISION and COMPLEX arrays.

The following twelve elements of array RAY will be printed:

```
(1,1,1) (2,1,1) (3,1,1) (1,2,1) (2,2,1) (3,2,1)
(1,3,1) (2,3,1) (3,3,1) (1,4,1) (2,4,1) (3,4,1)
```

If all the requested elements of an array have the same value, PMD will print the message:

```
ALL REQUESTED ELEMENTS OF THIS ARRAY
    WERE...
```

If several consecutive elements of an array sub-block have the same value, PMD prints the message:

```
ALL THREE ELEMENTS WERE...
```

PMDDUMP

The format of the call to PMDDUMP is shown in figure 10-2.

```
CALL PMDDUMP
```

Figure 10-2. PMDDUMP Call

PMDDUMP causes a dump of variables in the calling routine, not at once, but when an abort occurs or when PMDLOAD or PMDSTOP is called. PMDDUMP and PMDLOAD or PMDSTOP need not be called from the same routine. The dump includes an analysis of all active routines that have called PMDDUMP. (An active routine is a routine that has been executed but is not necessarily in the traceback chain.) In addition, following an abort or call to PMDSTOP, all routines in the traceback chain are dumped. Up to ten subprograms can be dumped. If more than ten subprograms call PMDDUMP, the extra calls are ignored.

PMDLOAD

The format of the call to PMDLOAD is shown in figure 10-3.

```
CALL PMDLOAD
```

Figure 10-3. PMDLOAD Call

PMDLOAD causes an immediate dump of variables in the calling routine and in any routines that have called PMDDUMP. Program execution continues normally after the dump unless PMDLOAD is called 10 times, in which case the last call is treated as a call to PMDSTOP.

PMDSTOP

The format of the call to PMDSTOP is shown in figure 10-4.

```
CALL PMDSTOP
```

Figure 10-4. PMDSTOP Call

PMDSTOP causes an immediate dump of variables in the calling routine, all routines in the traceback chain, and any routines that have called PMDDUMP. The job is then aborted. Programs cannot recover from a call to PMDSTOP.

This section describes the FTN5 control statement options, the FORTRAN output listing formats, and the execution control statement.

FTN5 CONTROL STATEMENT

The FORTRAN compiler is called from the system library and executed by an FTN5 control statement. The FTN5 control statement calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This control statement can have any of the formats shown in figure 11-1. Some examples of FORTRAN control statements are as follows:

```
FTN5(ET=W,LO=R,S=0)

FTN5,I=INF,L=LIST.
```

Format 1:

```
FTN5(p[,p] . . . )
```

Format 2:

```
FTN5.
```

Format 3:

```
FTN5,p[,p] . . . •
```

Figure 11-1. FTN5 Control Statement

PARAMETERS

The optional parameters $p_1, \ldots, p_n$ must be separated by commas and can be in any order. If no parameters are specified, FTN5 is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the syntax for job control statements as defined in the operating system reference manual, with the added restriction that a comma is the only valid parameter delimiter. Columns following the right parenthesis or period can be used for comments; they are ignored by the compiler, but are printed on the dayfile.

Most parameters need not be specified because a default value will be used if the parameter is omitted. This default value (called the first default) has been chosen to be the most commonly desired value. First defaults are set when the system is installed; since installations can change first default values, the user should determine what values are in effect at the user's particular installation.

There is a second parameter value that is almost as commonly desired as the one chosen for the first default. Writing only the parameter name will select the second most commonly used value for the parameter (called the second default). Second defaults are not installation changeable.

Unrecognizable parameters cause compilation to terminate. Conflicting options either are resolved or cause compilation to terminate, depending on the severity of the conflict; this resolution is indicated in a dayfile entry.

The values of the B, DB, E, ET, G, I, LO, ML, PD, PN, PS, S, and X parameters are passed to COMPASS when intermixed COMPASS subprograms are present.

Parameters fall into two general classes: those with two possible settings, on or off; and those that have a specific value, such as a file name, optimization level, and so forth.

Binary Value Parameters

A binary value parameter has two possible settings, on or off. On is selected by writing the parameter name. Off is selected by writing parameter-name=0.

Specified Value Parameters

Specified value parameters provide the compiler with specific values for such things as file names, page size, and so forth. For parameters where multiple values are allowed, the values are separated by slashes. The specified value parameters are indicated in table 11-1.

Multiple Binary Value Parameters

Some binary value parameters are grouped together under a single parameter because they control related operations; for example, listing options. Such groupings of binary value parameters are called multiple binary value parameters. Like all parameters, first and second default values are defined to cover the most frequent usages. The form of a multiple binary value parameter is:

```
parameter=op[/op]...
```

where op is a binary value indicating on or off.

If options are selected for a multiple binary value parameter, they are processed as follows:

1. An initial value is selected for each option corresponding to the most commonly desired options.
2. The option list is scanned from left to right.
3. -op deselects the specified option.
4. op selects the specified option.
5. 0 deselects all previously selected options.

If the initial value set is close to what the user desires, the easiest way to select the values is by addition or deletion of specific options. For example, if DB=TB/SB/SL/ER/ID is wanted, the user would write FTN5,DB=-PMD/ID to remove PMD from the initial values and to add ID.

TABLE 11-1. DEFAULTS FOR FTN CONTROL STATEMENT

Parameter	First Default (parameter omitted)	Second Default (keyword only)	Initial Values
ANSI	ANSI=0	ANSI=T	
ARG	ARG=0	ARG=-COMMON/FIXED	ARG=0
B	B=LGO	B=BIN	
BL	BL=0	BL	
CS	CS=USER	CS=FIXED	
DB	DB=0 if opt=1, 2, or 3; DB=0/ER if opt=0	DB=TB/SB/SL/ER/PMD	DB=TB/SB/SL/ER/PMD
DO	DO=0	DO=OT	DO=0
DS	DS=0	DS	
E	E=OUTPUT	E=ERRS	
EC	EC	EC	
EL	EL=T	EL=F	
ET	ET=0	ET=F	
G	G=0	G=SYSTEXT	
GO	GO=0	GO	
I	I=INPUT	I=COMPILE	
L	L=OUTPUT	L=LIST	
LCM	LCM=D	LCM=I	
LO	LO=S/A	LO=S/A/R	LO=S/A
MD	MD=0	MD=T	
ML	ML=0	ML=0	
OPT	OPT=0	OPT=2	
PD	PD=6	PD=8	
PL	PL=5000	PL=50000	
PN	PN=0	PN	
PS	PS=60 if PD=6; PS=80 if PD=8	None	
PW	PW=136 (PW=72 for connected file)	PW=72	
QC	QC=0	QC	
REW	REW=0	REW=I/B	REW=I/B/E
ROUND	ROUND=A/S/M	ROUND=A/S/M/D	ROUND=0
S	S=SYSTEXT if G omitted S=0 if G specified	S=SYSTEXT if G omitted S=0 if G specified	
SEQ	SEQ=0	SEQ	
STATIC	STATIC=0	STATIC=0	
TM	TM=characteristics of compiling machine	TM=0	TM=0
X	X=OLDPL	X=OPL	

If the initial value set is not what is desired, the user should delete all of the initial values and add the desired values. For example, if only DB=PMD is wanted, the user would write FTN5,DB=0/PMD to deselect the initial value set and to add PMD.

Multiple binary value parameters are indicated in table 11-1 by a value in the initial value column.

Multiple Appearance of Parameters

The G and S parameters are the only parameters which can have multiple appearances in a FTN5 control statement. If any other parameter appears more than once, compilation terminates.

PARAMETER OPTIONS

Following are descriptions of the options for each of the FTN5 control statement parameters.

ANSI Diagnostics

The ANSI parameter specifies whether use of non-ANSI extensions to FORTRAN are to be diagnosed and if so, how severely. Valid options are:

omitted	Same as ANSI=0.
ANSI	Same as ANSI=T.
ANSI=0	No ANSI diagnostics are generated.
ANSI=T	ANSI errors are treated as trivial errors.
ANSI=F	Non-ANSI usages result in a fatal error. All ANSI warning diagnostics become fatal.

Refer to the EL parameter for an explanation of trivial and fatal diagnostics.

ARG Argument List Attributes

The ARG parameter is a multiple binary value parameter that specifies attributes of external procedure argument lists generated by the compiler. Valid options are:

omitted	Same as ARG=0.
ARG	Same as ARG=-COMMON/FIXED.
ARG=0	Same as ARG=-FIXED/-COMMON; both options are deselected.
ARG=op[/op]	

where op is one of the following:

COMMON	Argument lists generated for external procedures will be of the form required for interlanguage communication. Specification of COMMON implies -FIXED.
FIXED	All references in the FORTRAN program to a given external procedure have the same number of arguments (the compiler-generated argument lists will not contain a zero terminator).

Initial value is ARG=0. Specification of ARG=COMMON/FIXED is not permitted.

B Binary Output File

The B parameter specifies the name of the compiler output file. Valid options are:

omitted	Same as B=LGO.
B	Same as B=BIN.
B=0	No binary output file is produced. Cannot be specified with GO.
B=lfn	Compiler-generated binary code is output on the file lfn.

The B parameter conflicts with the QC parameter.

BL Burstable Listing

The BL parameter controls page ejects in the listing produced by the compiler. Valid options are:

omitted	Same as BL=0.
BL	Generates output listing that is easily separable into components by issuing page ejects between source listing, cross-reference-attributes map, and object code listing. Also ensures that each program unit listing contains an even number of pages, issuing a blank page at the end if necessary.
BL=0	Generates listings in compact format by minimizing page ejects.

CS Collating Sequence

The CS parameter specifies the weight table to be used for the evaluation of character relational expressions. Valid options are:

omitted	Same as CS=USER.
CS	Same as CS=FIXED.
CS=USER	User-specified weight table.
CS=FIXED	Fixed weight table. (See Collating Sequence Control in section 7.)

DB Debugging Options

The DB parameter is a multiple binary value parameter that selects debugging options. Valid options are:

omitted	Same as DB=0 if opt=1, 2, or 3; DB=0/ER if opt=0.
DB	Same as DB=TB/SB/SL/ER/PMD.
DB=0	All options are deselected.
DB=op[/op]...	

where op is one of the following:

TB	A full error traceback occurs upon detection of an execution time error. This option causes arguments to intrinsic functions to be passed by reference.
SB	Subscript bounds checking is performed.
SL	Character substring expressions are checked to ensure that the substring references are within the string.
ER	Selects object time reprieve of execution errors. A message identifying the program unit and line number containing the error is output.
ID	A line number table, symbol table, and special stylized object code (required by CYBER Interactive Debug) are generated along with the binary code. CYBER Interactive Debug uses the tables while processing the user program to determine variable locations, source line locations, and other useful debugging information. This option must be specified if the special FORTRAN features of CYBER Interactive Debug are to be used and debug mode has not been turned on by the DEBUG control statement. DB=-ID overrides a previous DEBUG control statement specification. DB=ID requires OPT=0.
ST	Same as ID except that the stylized object code is not generated.
PMD	Must be specified if the Post Mortem Dump facility is to be used. Symbol tables are written to separate files that are accessed by Post Mortem Dump so that a symbolic analysis of error conditions, variable names and values, and traceback information can be written to an output file.

Initial value is DB=TB/SB/SL/ER/PMD. If PMD is specified ARG=FIXED must not be selected.

DO Loop Control

The DO parameter is a multiple binary value parameter that specifies the manner in which DO loops are to be interpreted by the compiler. Valid options are:

omitted	Same as DO=0.
DO	Same as DO=OT.
DO=0	Trip count must be less than 131 071 and minimum trip defaults to zero.
DO=op [/op]	

where op is one of the following:

LONG	Permits the trip count to exceed 131 071.
OT	Sets the minimum trip count for DO loops to one. This option can result in faster program execution.

Most DO loops have a trip count of at least one and will execute correctly under either option. However, if a DO loop has a trip count of zero and DO=OT is specified, the program might not execute correctly. The effects of this parameter can be overridden by the C$ DO directive. Initial value is DO=0.

DS Directive Suppression

The DS parameter suppresses the recognition of C$ directives. Valid options are:

omitted	Same as DS=0.
DS	All C$ directives are treated as comments.
DS=0	All C$ directives are recognized and processed.

E Error File

The E parameter specifies the name of the file to receive error information. Valid options are:

omitted	Same as E=OUTPUT.
E	Same as E=ERRS.
E=lfn	In the event of an error of EL-specified severity or higher, the error line and diagnostic are written to lfn. If the L (full listing) parameter is specified, this information is also written to the file specified by the L parameters. E=0 is an error.

EC Extended Memory Usage

The EC parameter specifies that OPT=2 tables will use LCM or ECS, when available.

omitted	Same as EC.
EC	Use LCM or ECS for OPT=2 tables, if available.
EC=0	Do not use LCM or ECS for OPT=2 tables.

EL Error Level

The EL parameter indicates the severity level of errors to be printed on the output listing. The levels are ordered by increasing severity. Specification of a particular level selects that level and all higher levels. Valid options are:

omitted	Same as EL=T.
EL	Same as EL=F.
EL=T	Lists trivial errors. The syntax of these errors is correct but the usage is questionable.
EL=W	Lists warning errors. These are errors where the syntax is incorrect but the compiler has made an assumption (such as inserting a comma) and continued.
EL=F	Lists fatal errors. A fatal error prevents the compiler from processing the statement where the error occurred.
EL=C	Lists catastrophic errors. These errors are fatal to compilation; the compiler is unable to continue processing the current program unit. Compilation continues with a subsequent program unit.

ET Error Terminate

The ET parameter specifies the action to be taken by the compiler when compilation has completed. If an error of the specified level or higher occurs, the job skips to an EXIT (NOS) or EXIT(S) (NOS/BE). Valid options are:

Omitted	Same as ET=0.
ET	Same as ET=F.
ET=0	The job continues even if errors are encountered.
ET=T	Skips if errors of severity T or higher are detected.
ET=W	Skips if errors of severity W or higher are detected.
ET=F	Skips if errors of severity F or higher are detected.
ET=C	Skips if errors of severity C are detected.

Refer to the EL parameter for a description of error severity levels.

Level T and W errors result in an executable binary file. Level F and C errors result in a binary that will abort the loader.

G Get System Text File

The G parameter specifies the name of a file to be read to obtain a system text for intermixed COMPASS subprograms. Valid options are:

omitted	Same as G=0.
G	Same as G=SYSTEXT
G=0	No system text is loaded.
G=lfn	Loads the system text from file lfn.
G=lfn-recname	Loads the system text from record recname on file lfn. (The hyphen is required separator notation.)

Up to seven system texts can be specified, separated by slashes. Multiple occurrences of this parameter are permitted. A G=0 specification is ignored if any other G option is specified.

GO Automatic Execution

The GO parameter specifies automatic loading and executing. Valid options are:

omitted	Same as GO=0.
GO	The binary output file is loaded and executed after compilation.
GO=0	The binary output file is not loaded and executed after compilation.

The GO option conflicts with the QC and B=0 options.

I Input File

The I parameter specifies the name of the file containing the input source code. Valid options are:

omitted	Same as I=INPUT.
I	Same as I=COMPILE.
I=lfn	Source code to be compiled is contained in file lfn. Compilation ends when an end-of-section, end-of-partition, or end-of-information is encountered. I=0 is an error.

L List File

The L parameter specifies the name of the file where the compiler writes the source listing and any other requested listing information except diagnostics (see LO parameter). L=0 suppresses all listings except that directed to the E file. Valid options are:

omitted	Same as L=OUTPUT.
L	Same as L=LIST.
L=0	Listing suppressed.
L=lfn	Listing is on file lfn.

LCM Extended Memory (LCM or ECS Storage Access)

The LCM parameter specifies the requirements on address size for data in extended memory. Refer to section 2, LEVEL Statement, for further information. Valid options are:

omitted	Same as LCM=D.
LCM	Same as LCM=I.
LCM=D	Specifies direct mode addressing. Provides more efficient code for accessing data assigned to extended memory. Extended memory field length must not exceed 131 071 words.
LCM=I	Specifies indirect mode (21-bit) addressing. This is the most efficient mode when extended memory field length exceeds 131 071 words. If a single common block exceeds 131 071 words, an error results.
LCM=G	Specifies giant mode addressing. Required if any single common block is larger than 131 071 words. LCM=G automatically selects DO=LONG.

LO Listing Options

The LO parameter is a multiple binary value parameter that specifies the information that is to appear on the output listing file (L parameter). Multiple options can be specified. Valid options are:

omitted	Same as LO=S/A.
LO	Same as LO=S/A/R.
LO=0	No O, R, A, M, or S information appears on the output listing.

LO=op[/op]...

where op is one of the following:

O	Output object code (COMPASS mnemonics) is listed on the output file.
R	A cross-reference map (described later in this section) is written to the output file.
A	A list of program entities (variables, common blocks) and their attributes (data type, class, etc.) is written to the output file.
M	A map, showing the correlation of program entities and physical storage, is written to the output file.
S	A source listing of the program is written to the output file.

Initial value is LO=S/A. The effects of the LO parameter can be controlled by the C$ LIST directive, described in appendix E.

MD Machine Dependent Diagnostics

The MD parameter specifies whether or not the use of machine-dependent language features are to be diagnosed, and how severely. Valid options are:

omitted	Same as MD=0.
MD=0	Machine-dependent diagnostics are not generated.
MD=T	Machine dependencies are treated as trivial errors.
MD=F	Machine-dependent language feature results in a fatal error.

Refer to the EL parameter for explanation of trivial and fatal errors.

ML MODLEVEL Micro

The ML parameter specifies the value of the MODLEVEL micro used by COMPASS. Valid options are:

omitted	Same as ML=0.
ML	Same as ML=0.
ML=0	The current date, in the form yyddd (where yy is the year and ddd is the number of the day within the year), is used for the MODLEVEL micro.
ML=str	The string str is used for the MODLEVEL micro; str consists of 1 through 7 letters or digits.

OPT Optimization Level

The OPT parameter specifies the level of optimization performed by the compiler. Valid options are:

omitted	Same as OPT=0.
OPT	Same as OPT=2.
OPT=0	Minimum optimization is performed, resulting in fastest compilation. OPT=0 is required for DB=ID.
OPT=1	Intermediate optimization is performed.
OPT=2	High optimization is performed, resulting in slower compilation.
OPT=3	Potentially unsafe optimizations in addition to all OPT=2 optimizations are performed.

In optimizing mode, optimizations can be performed in two ways: by the compiler and by the user. User optimization includes not only the standard methods that represent good programming practice, but also certain specific methods that enable the compiler to optimize more effectively. Source code optimization and object code optimization are discussed in the following paragraphs:

OPT=0 Compilation

In the OPT=0 compilation mode, compile time evaluations are made of constant subexpressions; redundant instructions and expressions within a statement are eliminated.

OPT=1 Compilation

In the OPT=1 compilation mode, the following optimizations take place in addition to those in OPT=0:

1. Redundant instructions and expressions within a sequence of statements are eliminated.
2. PERT critical path scheduling is done to utilize the multiple functional units efficiently.
3. Subscript calculations are simplified, and values of simple integer variables are stored in machine registers throughout loop execution, for innermost loops satisfying all of the following conditions:

   Having no entries other than by normal entry at the beginning of the loop

   Having no exits other than by normal termination at the end of the loop

   Having no external references (user function references or subroutine calls; input/output, STOP, or PAUSE statement; or intrinsic function references) in the loop

   Having no IF or GOTO statement in the loop branching backward to a statement appearing previously in the loop

OPT=2 Compilation

In the OPT=2 compilation mode, the compiler collects information about the program unit as a whole and the following optimizations are attempted in addition to those in both OPT=0 and OPT=1:

1. Values of simple variables are not retained when they are not referenced by succeeding statements.
2. Invariant (loop-independent) subexpressions are evaluated prior to entering the loops containing them.
3. For all loops, the evaluation of subscript expressions containing a recursively defined integer variable (such as I when I=I+1 appears within the loop) is reduced from multiplication to addition.
4. Array addresses, values of simple variables in central memory, and subscript expressions are stored in machine registers throughout loop execution for all loops.
5. In all loops and in complicated sections of straight-line code, array references and subscript values are stored in machine registers.
6. In small loops, indexed array references are prefetched after safety checks are made to ensure that the base address of the array and its increment are reasonable and should not cause an out-of-bounds reference (mode 1 error).

OPT=3 Compilation

In OPT=3 compilation mode, the compiler performs certain optimizations which are potentially unsafe. The following optimizations are performed in addition to those provided by OPT=2:

1. In small loops, indexed array references are prefetched unconditionally without any safety checks. For example:

```
      REAL B(100,100)
      .
      .
      .
      DO 20 I = 1,100,10
   20 S = S + B(J,I)
```

   When the compiler prefetches the reference to B, the last reference to B in the loop is B(J,101) which might cause an out-of-bounds error at execution time if the array B is stored near the end of the field length.

2. When an intrinsic function is referenced, the compiler assumes that the contents of certain B registers are preserved for use following the function processing. For example:

```
      REAL A(10),C(10)
      .
      .
      .
      DO 10 I=1,N
   10 C(I) = EXP(A(I))
```

   The compiler might assign I and N to B registers during the loop.

In a loop, the registers available for assignment are determined by the presence or absence of external references. External references are user function references and subroutine calls, input/output statements, and intrinsic functions (SIN, COS, SQRT, EXP, and so forth).

When OPT=3 is not selected, the compiler assumes that any external reference modifies all the registers; therefore, it does not expect any register contents to be preserved across function calls.

If a math library other than the FORTRAN Common Library is used at an installation to supply intrinsic functions, the B register portion of the OPT=3 option must be deactivated by an installation option in order to ensure correct object code.

Source Code Optimization

To achieve maximum object code optimization regardless of optimization level, the user should observe the following practices for programming source code:

1. Since arrays are stored in column major order, DO loops (including implied DO loops in input/output lists) which manipulate multidimensional arrays should be nested so that the range of the DO loop indexing over the first subscript is executed first. Implied DO loop increments should be one whenever possible. For example:

```
      DIMENSION A(20,30,40), B(20,30,40)
      .
      .
      .
      DO 10 K = 1, 40
      DO 10 J = 1, 30
      DO 10 I = 1, 20
   10 A(I,J,K) = B(I,J,K)
```

2. The number of different variable names in subscript expressions should be minimized. For example:

```
X = A(I+1,I-1) + A(I-1,I+1)
```

   is more efficient than:

```
IP1 = I+1
IM1 = I-1
X = A(IP1,IM1) + A(IM1,IP1)
```

3. The use of EQUIVALENCE statements should be avoided, especially those including simple variables and arrays in the same equivalence class.

4. Common blocks should not be used as a scratch storage area for simple variables.

5. Program logic should be kept simple and straightforward; program unit length should be less than about 600 executable statements.

6. The use of dummy arguments (formal parameters) and variable dimensions should be avoided if possible; common or local variables should be used instead.

PD Print Density

The PD parameter specifies print density for all printable output (L and E files). The destination printer must be capable of supporting the specified density. For interactive connected files, PD options are suppressed. Valid options are:

omitted	Same as PD=6.
PD	Same as PD=8.
PD=6	Compiler output is printed at six lines per inch, single spaced.
PD=8	Compiler output is printed at eight lines per inch, single spaced.

PL Print Limit

The PL parameter specifies the maximum number of records (print lines) that the executing program can write to file OUTPUT. This parameter is operative only when appearing on an FTN5 statement used to compile a main program. Valid options are:

omitted	Same as PL=5000.
PL	Same as PL=50000.
PL=n	Output must not exceed n lines; n is a decimal integer consisting of one through nine digits for NOS/BE and SCOPE 2 or one through seven digits for NOS.

PN Pagination

The PN parameter specifies page numbering options for the compiler output listing. Valid options are:

omitted	Same as PN=0.
PN	Page numbering is continuous from program unit to program unit, including intermixed COMPASS output.
PN=0	Page numbers begin at 1 for each program unit.

PS Page Size

The PS parameter specifies the number of lines to be included on a printed page of the output listing. Valid options are:

omitted	Same as PS=60 if PD=6. Same as PS=80 if PD=8.
PS=n	Specifies the maximum number of printed lines; n must not be less than 4.

PW Page Width

The PW parameter specifies the width of an output line. Valid options are:

omitted	For a connected listing (L) or error (E) file, same as PW=72. For all other output files, same as PW=136.
PW	Same as PW=72.
PW=n	Printed lines are to contain n characters; n is a decimal integer and must not be less than 50 or greater than 136. Lines shorter than 136 characters are reformatted rather than truncated, as described later in this section under Listings.

QC Quick Syntax Check

The QC parameter specifies that the compiler is to perform a quick syntax check of the source program. Valid options are:

omitted	Same as QC=0.
QC	The compiler performs a full syntactic scan of the program, but no binary code is produced. No code addresses are provided if a reference map is requested. QC compilation is sub- stantially faster than normal compilation; but it must not be selected if the program is to be executed.
QC=0	Quick syntax check is not performed.

The QC option conflicts with the B, GO, and LO=O/M options.

REW Rewind Files

The REW parameter specifies the files to be rewound prior to compilation. Valid options are:

omitted	Same as REW=0.
REW	Same as REW=I/B.
REW=0	No files are rewound.
REW=op[/op]...	

where op is one of the following:

I	Rewinds the input file (specified by the I parameter).
E	Rewinds the error file (specified by the E parameter).
B	Rewinds the binary output file (specified by the B parameter).
L	Rewinds the output file (specified by the L parameter).

Initial value is REW=I/B/E.

ROUND Rounded Arithmetic Computations

The ROUND parameter specifies which arithmetic operations are to be performed using rounded arithmetic. This parameter controls only the in-line object code compiled for arithmetic expressions; it does not affect computations performed by library subroutines, intrinsic functions, or input/output routines. Valid options are:

omitted	Same as ROUND=A/S/M.
ROUND	Same as ROUND=A/S/M/D.
ROUND=0	No rounding is performed.
ROUND=op[/op]...	

where op is one of the following:

A	All addition operations are rounded.
S	All subtraction operations are rounded.
M	All multiplication operations are rounded.
D	All division operations are rounded.

Initial value is ROUND=0.

S System Text File

The S parameter specifies the name of the system text to be read by the compiler. Valid options are:

omitted	Same as S=SYSTEXT if G parameter is not specified.
	Same as S=0 if G parameter is specified.

S	Same as S=SYSTEXT if G parameter is not specified.
	Same as S=0 if G parameter is specified.
S=0	System text file is not loaded when COMPASS is called to assemble any intermixed COMPASS subprograms.
S=sname	Specifies the system text name to be sname and searches the global library set.
S=lib-sname	Searches the library named lib for the system text named sname. (The hyphen separating lib and sname is required.)

Multiple names can be specified by separating them with slashes, up to a maximum of seven names. Multiple occurrences of this parameter are permitted. An S=0 specification is ignored if any other S option is specified.

SEQ Sequenced Input

The SEQ parameter specifies source file sequencing format. Valid options are:

omitted	Same as SEQ=0.
SEQ	The source input file is in sequenced line format.
SEQ=0	The source input file is in standard FORTRAN format.

STATIC Static Load

The STATIC parameter specifies static inclusion of file buffers. Valid options are:

omitted	Same as STATIC=0.
STATIC	Inhibits dynamic file allocation at execution time by run-time library. Required library programs must be selected by calls to the routines described in section 7.
STATIC=0	Use of dynamic memory management at execution time by run-time library.

TM Target Machine

The TM option specifies attributes of the object time machine. This parameter is an installation option. (Not available on SCOPE 2.) Valid options are:

omitted	Attributes of the object time machine are assumed to be identical to those of the compile time machine.
TM	Same as TM=0.
TM=0	Object time machine is assumed to have none of the possible attributes; for example, no LCM.
TM=LCM	Object time machine is assumed to have large central memory available.

X External Text Name

The X parameter specifies the name of the file from which the COMPASS assembler reads the external text when it encounters an XTEXT directive in the intermixed COMPASS program. Valid options are:

omitted	Same as X=OLDPL.
X	Same as X=OPL.
X=lfn	COMPASS assembler reads external text from file lfn.

The X parameter is intended for use with intermixed COMPASS subprograms only.

FTN5 CONTROL STATEMENT EXAMPLES

Some examples of FTN5 control statements are as follows:

Example 1:

```
FTN5 (ET=F,EL=F,GO,L=SEE,LO=M/R,S=0)
```

selects the following options:

ET=F	On fatal compilation errors, skips to an EXIT (NOS) or EXIT,S (NOS/BE and SCOPE 2) control statement.
EL=F	Fatal diagnostics only are listed.
GO	Generated binary object file is loaded and executed at end of successful compilation.
L=SEE	Listed output appears on file SEE.
LO=M/R	Reference map and storage map are listed in addition to source listing and attributes list.
S=0	When COMPASS is called to assemble an intermixed COMPASS subprogram, it does not read in a system text file.

Example 2:

```
FTN5 (GO,DB=ID)
```

selects the following options:

GO	The program is loaded and executed after compilation.
DB=ID	Tables are generated for use by CYBER Interactive Debug.

Example 3:

```
FTN5.
```

selects default options. Refer to table 11-1 for a summary of the default options.

COMPILER LISTINGS

The listings produced by FORTRAN during compilation are determined by control statement parameters. The types of listings produced and the control statement parameters that influence them are as follows:

Source listing

Includes all source lines submitted for compilation as part of the source input file. The C$ LIST(S) directive can be used to suppress the listing of selected source lines. Listed lines are preceded by a line number. Information contained in the source listing is determined by the LO parameter.

Diagnostics

Includes informative, note, warning, ANSI, fatal diagnostics, and catastrophic as determined by the EL and ANSI parameters (appendix B). Catastrophic diagnostics cannot be suppressed. Diagnostics appear immediately after the source line where they were detected. (Some declarative processor diagnostics appear at the end of the declarative statements.)

Object code

Includes generated object code, listed as COMPASS assembly language instructions. Selected by the LO=O option.

Reference map

Includes compiler assigned locations, as well as other attributes, of all symbolic names, statement labels, and other program entities in each program unit. Contents are determined by the LO parameter.

Optimizer statistics

Includes a summary of optimizations performed by the compiler. Optimizations are determined by the OPT parameter.

Statistics

Includes program field length and CPU seconds used for compilation.

A header line at the top of each page of compiler output contains the program unit type and name, the computer used for compilation, and the target computer for which the code is being compiled, some of the control statement options, compiler version and mod-level, date, time, and page number. The source program is listed at 60 lines per page (including headers) unless a different value is specified by the PS parameter.

The files to which listings are written is determined by the L and E control statement parameters.

SHORT LINE LISTING FORMAT

When the page width specified by the PW parameter on the FTN5 control statement is less than 132, the output listing is reformatted so that source statements and error messages fit in a line of the specified width. Source statements are broken at the maximum line length and are resumed in the tenth printed column of the following line with >>>> appearing in columns three through six. Error messages are broken at the nearest blank and are resumed in the same manner as source lines.

When the compiler output listing file is connected to a terminal, the default for the PW parameter is the terminal line length. When the terminal line length is not determinable, a length of 72 characters is assumed.

If the PW value is equal to or greater than 126, the page header occupies one line. If the PW value is less than 126, the header is reformatted into two lines. In this case, the subtitle line is suppressed. Note that PW cannot be less than 50.

LISTING CONTROL DIRECTIVE

The C$ LIST comment directive provides control over the listings produced by the LO (listing options) parameter selected on the FTN5 control statement. The C$ LIST directive is described in appendix E.

REFERENCE MAP

The reference map is a dictionary of all programmer-created symbols appearing in a program unit. The symbol names are grouped by class and listed alphabetically within the groups. The reference map follows the source listing of the program and the diagnostics (if present), and precedes the object listing (if present).

The kind of reference map produced is determined by the LO parameter on the FTN5 control statement. The applicable reference map options are as follows:

A	A list of program entities (variables, common blocks) and their attributes (data type, class, and so forth) is written to the output file.
M	A map, showing the correlation of program entities and physical storage, is written to the output file.
R	A cross-reference map is written to the output file.

The initial values set for the LO parameter are as follows:

omitted	Selects S/A.
LO	Selects S/A/R.
LO=opt/ . . .	Selects S/A as initial values. All options then selected are added.
LO=0	Deselects all listing options.
LO=0/opt/ . . .	Deselects all listing options, then selects the specified options.

Examples:

LO=A	Selects S/A.
LO=R	Selects S/A/R.
LO=0/R	Selects R.

LO=M	Selects S/A/M.
LO=0/S	Selects S.
LO=S	Selects S/A.

L=0 forces LO=0, but LO=0 has no effect on L.

Fatal errors in the source program cause certain parts of the map to be suppressed; parts of the map might also be incomplete, or inaccurate. Fatal to execution (F) and fatal to compilation (C) errors cause the DO loop map to be suppressed, and assigned addresses will be different; symbol references might not be accumulated for statements containing syntax errors.

GENERAL FORMAT OF MAPS

Each class of symbol is preceded by a subtitle line that specifies the class and the properties listed. Formats for each symbol class are different, but printouts contain the following information:

The octal address associated with each symbol relative to the origin of the program unit or common block. All addresses will print as blank if QC is selected.

Properties associated with the symbol.

List of references to the symbol (for LO=R only).

All line numbers in the reference list refer to the line of the statement in which the reference occurs.

All numbers to the right of the name are decimal integers unless they are printed in the form O"..." to indicate octal.

Names of symbols generated by the compiler (such as system library routines called for input/output) do not appear in the reference map.

The following subsections describe the various sections of the reference map as they would appear for the full map, selected by LO=M/A/R. The sections that appear for a given option are indicated.

Variables Map

Variable names include local and COMMON variables and arrays, dummy arguments and, for FUNCTION subprograms, the defined function name when used as a variable. Figure 11-2 shows the variable map format.

```
--VARIABLE MAP--(LO=A/M/R)

-NAME--ADDRESS--BLOCK--PROPERTIES--TYPE--SIZE--REFERENCES-

name   addr      block    prop1/prop2   type   size   refs

    name      Variable name as it appears in FORTRAN source listing.  Variables are listed in alphabetical order.

    addr      Relative address assigned to variable name.
              If name is a member of a COMMON block, addr is relative to the start of block.
              If name is a non-SAVEd local variable, then addr is program relative.
              If name is a SAVEd local variable, then addr is relative to the /S$A$V$E/ block for this program unit.
              If name is a dummy argument, then addr is the offset into the program unit composite formal parameter
              list.  (Not necessarily the same as its position in a source program dummy argument list.)
              If addr is given as NONE, then the compiler (especially OPT=2) has determined that name does not need
              to be stored in memory.

    block     Name of COMMON block in which variable name appears.  If blank, name is a local variable.

              / /             Indicates name is in blank COMMON.

              DUMMY-ARG       Indicates variable name is a dummy argument (formal parameter) to this subprogram.

The following are obtained only with the LO=A or LO=M option:

    prop      Properties associated with variable name; indicated by the following keywords (listed in the format
              prop1/prop2. . .):

              UND             Variable name has not been defined.  A variable is defined if any of the following
                              conditions hold:

                              Appears in a COMMON or DATA statement.
                              Is equivalenced to a variable that is defined.
                              Appears on the left side of an assignment statement at the outermost parenthesis level.
                              Is the index variable in a DO loop.
                              Appears as a stand-alone actual parameter in a subroutine or function call.
                              Appears in an input list (READ, BUFFERIN, etc.).

                              Otherwise, the variable is considered undefined; however, variables which are used (in
                              arithmetic expressions, etc.) before they are defined (by an assignment statement or
                              subprogram call) are not flagged.
```

Figure 11-2. Variable Map (Sheet 1 of 2)

EQV	Variable name is equivalenced.
LEVn	Variable name is given a LEVEL due to the source program.
SAV	Variable name has the SAVE property.
UNUSED	Name appears only in dummy argument list(s) and/or in a nondimensioning type statement.
S	Name appears only once in the entire program unit. The user should check carefully for other names with similar spellings.

A name will not be flagged as *S*(STRAY) nor UNUSED if it is in COMMON, is a DO loop control index, or is used as a subroutine, function, or entry.

type Gives the mode associated with the variable name.

LOGICAL, INTEGER, REAL, COMPLEX, DOUBLE, CHARACTER or BOOLEAN.

In the case of CHARACTER, the form is:

CHAR*n	For specified length
CHAR*(*)	For adjustable length

size Number of elements of name, when name is dimensioned. For nonarray names, this field is blank. The size is given by UB-LB+1, where UB = upper bound, LB = lower bound. For adjustable dimensions ADJ.ARY. is given.

refs (appear only with R option)
References and definitions associated with variable name; listed by line number.
Certain references are followed by a usage suffix, chosen from:

/A	Argument: pass by reference actual argument in a subroutine or user-function call.
/C	Control: DO statement where name is the loop control variable.
/I	Initialized: in a DATA statement.
/R	Read: name appears as an input list item of a READ or ENCODE; as an internal file identifier of a WRITE, or DECODE statement; or as a BUFFER IN limit.
/S	Store: name appears as a store target in an assignment or ASSIGN statement; or receives a value as an IOSTAT= specifier or in an INQUIRE statement.
/U	Unit: used as an I/O unit designator, except an internal file designator.
/W	Write: value of name is written, by appearing as an output list item of a PRINT, PUNCH, or WRITE; or internal file identifier of READ or ENCODE; or as a BUFFER OUT limit.

Figure 11-2. Variable Map (Sheet 2 of 2)

Symbolic Constants Map

A symbolic constant is declared in a PARAMETER statement. The format of the symbolic constant map is shown in figure 11-3.

Procedure Map

Procedures include names of functions or subroutines called explicitly from a program or subprogram, names declared in an EXTERNAL statement, and names of intrinsic and statement functions appearing in the subprogram. Implicit external references, such as calls by certain FORTRAN source statements (READ, ENCODE, etc.) are not listed. The format of the procedure map is shown in figure 11-4.

Statement Label Map

The statement label map includes all statement labels defined in the program or subprogram. The format of the statement label map is shown in figure 11-5.

Entry Point Map

Entry point names include program and subprogram names and names appearing on ENTRY statements. The format of the entry point map is shown in figure 11-6.

Input/Output Unit Map

The input/output unit map includes constant UNIT designators. Standard or extended internal files are not included. The format of the input/output unit map is shown in figure 11-7.

NAMELIST Map

The namelist map contains the names of the namelist groups defined in the program unit. The format of the namelist map is shown in figure 11-8.

```
--SYMBOLIC CONSTANTS--(LO=A/M/R)

-NAME----TYPE-------VALUE------------------REFERENCES

name     type      value                    refs
```

name — Symbol name as it appears in the source. Listed in alphabetic order.

type — Same as type field in VARIABLES section.

value — Value assigned to name. Format depends on type:

LOGICAL	Either .TRUE. or .FALSE. .
BOOLEAN	O"nnn".
REAL, DOUBLE	O"nnn".
COMPLEX	Real half listed as O"nnn".
INTEGER	Integer value if magnitude less than 10000000000, otherwise O"nnn". Leading minus if < 0.
CHARACTER	Enclosed in quotes. If value does not fit in the columns provided, then the trailing quote is replaced by an ellipsis (. . .).

refs — Source line number where referenced. Suffix /S indicates definition line. Appears only when R option is specified.

Figure 11-3. Symbolic Constants Map

```
--PROCEDURES--(LO=A/M/R)

-NAME----TYPE----ARGS----CLASS-------REFERENCES-

name     type    arg      class       refs
```

The following are obtainable only with the LO=A or M option:

name — Symbol name as it appears in the source listing.

type — Gives the result mode for a function. One of the following:

Blank if class is SUBROUTINE or EXTERNAL.
GENERIC if appropriate.
Otherwise, one of the type designators listed in VARIABLES section.

args — Number of arguments. If the number is variable (MAX, MIN, etc.), VAR is given as number of arguments. UNKNOWN if external.

class — One of the following:

SUBROUTINE	
DUMMY SUBR	Dummy argument subroutine
INTRINSIC	
STAT FUNC	Statement function
FUNCTION	Nonintrinsic external function
DUMMY FUNC	Dummy argument function
EXTERNAL	None of the above

refs — (with LO=R option only). Line number on which name is referenced. Reference might be suffixed with:

/D Declarative statement, or definition line of a statement function.

/A Argument: pass by reference actual argument in a subroutine or user-function call.

Figure 11-4. Procedures Map

```
-STATEMENT LABELS-(LO=A/M/R)

-LABEL------ADDRESS------PROPERTY----DEF--REFERENCES-

label       addr          prop          def  refs
```

label — Statement label from FORTRAN source program. Statement labels are listed in numerical order.

addr — Program-relative address assigned to statement label. When no meaningful address can be assigned, one of the following flags will appear:

UNDEF	Statement label is not defined; refs lists all occurrences of the undefined label. Undefined labels also generate a diagnostic.
NO REFS	label is not referenced by any statements. This label can be safely removed from the FORTRAN source program.
INACTIVE	label has been deleted by optimization.
blank	no address is available usually due to source program fatal error, or QC option.

prop — One of the following:

FORMAT	Statement label is a FORMAT.
DO-TERM	Statement label appeared in a DO statement.
NON-EX	Label appeared on a nonexecutable statement. If addr is not *NO REFS*, then the program is incorrect, and a diagnostic will have been issued.
blank	Label is a normal control label.

def — Source line number where label was defined. *UNDEF* if not defined.

refs — Line numbers on which label was referenced. (Appears only with LO=R option.) Usage suffixes are as follows:

/A	Assign statement
/D	DO statement
/R	Input or DECODE format
/W	Output or ENCODE format

Figure 11-5. Statement Label Map

```
--ENTRY-POINTS--(LO=A/M/R)

-NAME-----ADDRESS-----ARGS-----REFERENCES

name      addr          args      refs
```

name — Entry point name as defined in FORTRAN source.

addr — Relative address assigned to the entry point.

args — Number of dummy arguments for entry name.

refs — In subprograms only, line number of RETURN statements and ENTRY definition. If line number is followed by /D, this implies the line number on which the name is defined. All RETURN statement refs are to the main entry point. Appears only on R maps.

Figure 11-6. Entry Point Map

```
--I/O UNITS--(LO=A/M/R)

-NAME-----PROPERTIES-----REFERENCES

name     prop1/prop2     refs

    name        The value of the constant UNIT designator.  If the value is an integer, 0≤ val ≤ 999, then name is
                TAPEnnn.

    prop        Properties listed include only the ones detectable in the current program-unit:

                FMT     A formatted operation appeared.
                BIN     A nonformatted operation appeared.
                DIR     A direct access operation.
                SEQ     A sequential operation.
                BUF     A BUFFER IN or BUFFER OUT statement.
                AUX     An auxiliary I/O statement.

    refs        Appears only with R option.
                Source line number of statements which explicitly refer to unit name.  Usage suffixes are:

                /R      Read operation.
                /W      Write operation, including ENDFILE.
                blank   OPEN, CLOSE, BACKSPACE, REWIND, INQUIRE.
```

Figure 11-7. Input/Output Unit Map

```
--NAMELISTS--(LO=A/M/R)

-NAME-----ADDRESS-----REFERENCES-

name     addr          refs

    name        Namelist group name as defined in FORTRAN source.

    addr        Relative address assigned to name.

    refs        Line numbers of references to name  (with LO=R option only).
                Line number will be followed by /D, /R, /W.
                /D = line number on which namelist is defined.
                /R = line number on which namelist appears in an input operation.
                /W = line number on which namelist appears in an output operation.
```

Figure 11-8. Namelist Map

DO Loop Map

The DO loop map includes all DO loops that appear in the program unit, including implied DO loops not in DATA statements, and lists their properties. This map is suppressed if fatal errors have been detected in the program unit or if QC was specified on the FTN5 control statement. Loops are listed in order of appearance in the program. This map appears only when LO=M is selected. The format of the DO loop map is shown in figure 11-9.

Common and Equivalence Map

The common and equivalence map shows the storage layout for common blocks, and the equivalence-induced storage overlap for all variables. This map appears only when LO=M is selected. The format of the common and equivalence map is shown in figure 11-10.

Equivalence-induced storage overlap classes are indicated by enclosing parentheses: the first item in an equivalence class is preceded by a left parenthesis, and the last item is followed by a right parenthesis. Entries under LOCAL EQUIVALENCE include items that are not declared in a COMMON statement.

Stray Names

If a program contains items with questionable or illegal attributes, the reference map will specify the following attributes:

STRAY	Indicates variable names that appear only once in the entire program unit.
NO-REFS	Indicates statement labels that are not referenced by any statement in the program unit.

--DO-LOOPS--

-LABEL---ADDRESS---INDEX---PROPERTIES---FROM---TO

label addr index prop first-last

label Statement label defined as end of loop, or I/O for implied DO loops in I/O statements.

addr Relative address assigned to the start of the loop body.

index Variable name used as control index for loop, as defined by DO statement.

prop Various keywords can appear, describing optimization properties of the loop:

XREF Loop not optimized because it contains references to an external subprogram (including compiler-generated references to library routines).

OPEN Loop not optimized because it can be reentered from outside its range.

OUTER Loop not optimized because other loops are contained inside it.

EXIT Loop not optimized because it contains references to statement labels outside its range.

first-last Line numbers of the first and last statements of the loop.

Figure 11-9. DO Loop Map

UNDEF Indicates statement labels that are referenced but are not defined in the program unit.

If the *STRAY* attribute appears, the following trivial diagnostic is issued:

nnn STRAY NAMES. SEE MAP.

Program Statistics

At the end of each program unit, statistics are printed in octal and decimal. The format of the statistics map is shown in figure 11-11.

If LO=A is not specified, only the diagnostic counts are printed.

DEBUGGING USING THE REFERENCE MAP

When debugging a new program, the reference map can be used to find names that have been punched incorrectly as well as other items that will not show up as compilation errors. The basic technique consists of using the compiler as a verifier and correcting the fatal errors until the program compiles.

Using the listing, the LO=R/M reference map, and the original flowcharts, the following information should be checked by the programmer:

Names incorrectly punched

Stray name flag in the variable map

Functions that should be arrays

Functions that should be in-line instead of external

Variables or functions with incorrect type

Unreferenced format statements

Unused formal parameters

Ordering of members in common blocks

Equivalence classes

When debugging a program, the reference map can be used to understand the structure of the program. Questions concerning the loop structure, external references, common blocks, arrays, equivalence classes, input/output operations, and so forth, can be answered by checking the reference map.

The sample program shown in figure 11-12 is compiled with LO=A/M/R to produce a full reference map. The reference map is shown in figure 11-13.

```
--COMMON+EQUIVALENCE--(LO=M/A/R)

/block/ LEVEL lev, SIZE = size units sav.

    item [item] . . .

    item [item] . . .
      .
      .
      .

LOCAL EQUIVALENCE

    item [item] . . .

    item [item] . . .
      .
      .
      .
```

block	Name of the common block being described.
lev	Storage level of the block (1, 2, or 3).
size	Total number of storage units occupied by the block.
units	CHARS for a block containing character variables. WORDS for a block with noncharacter variables.
sav	SAVE if the block is saved, otherwise blank.
item	Describes the storage position of a variable or array. Each item consists of three fields: name first : last
name	Symbolic name of the item.
first	Number of the storage unit occupied by the first element of name.
last	Number of the storage unit occupied by the last element of the name.

First and last are given in decimal. The first position in a block is numbered one. For a 2-word scalar, last = first + 1. The item descriptors are printed left-to-right in order by ascending first then descending last. When name is of type character, first and last are given in character storage units, and they are separated by a colon. Otherwise, they are in words, separated by a dash.

Figure 11-10. Common Equivalence Map

```
--STATISTICS--

    PROGRAM-UNIT LENGTH     Length of program including code, storage for local variables, arrays, constants, temporaries, etc., but excluding buffers and common blocks.

    CM STORAGE USED         Maximum memory used during the compilation.

    COMPILE TIME            Compilation time of subprogram unit.

    nnn { ANSI    }
        { TRIVIAL }
        { WARNING } ERROR(S) IN program
        { FATAL   }
```

Figure 11-11. Program Statistics Map

```
      PROGRAM MAPS(INPUT,OUTPUT,TAPE3=INPUT,TAPE2=OUTPUT)
      CHARACTER*4 FILE
      INTEGER SIZE1,S1,SIZE2,S2,STRAY
      EQUIVALENCE (SIZE1,S1), (SIZE2,S2)
      NAMELIST /PARAMS/ SIZE1,SIZE2
      DATA S1,S2 /12,12/
C
100   READ(3,PARAMS)
      WRITE(2,PARAMS)
      WRITE(2, '(" SAMPLE PROGRAM TO ILLUSTRATE COMPILER MAPS")')
      CALL PASCAL(S1)
      WRITE(2, '(" THE FOLLOWING WILL HAVE NO HEADING")')
      CALL NOHEAD(S2)
      STOP
      END
C
      BLOCK DATA
      COMMON /ANARRAY/ X(22)
      INTEGER X
      DATA X(22) /1/
      END
C
      SUBROUTINE PASCAL(SIZE)
      INTEGER L(22), SIZE
      COMMON /ANARRAY/ L
C
      WRITE(2, '(" PASCALS TRIANGLE")')
      ENTRY NOHEAD
      M = MIN0(21,MAX0(2,SIZE-1))
      DO 2 K=21,22-M,-1
      L(K) = 1
      DO 1 J=K,21
1     L(J) = L(J) + L(J+1)
2     WRITE(2, '(1X,22I6)') (L(J),J=K,22)
      RETURN
      END
```

Figure 11-12. Program MAPS

```
--VARIABLE MAP--(LO=M/A/R)                                          A=ARGLIST, C=CTRL CF DC, I=DATA INIT,
 -NAME---ADDRESS--BLOCK-----PROPERTIES-------TYPE---------SIZE---REFERENCES-   R=READ, S=STORE, U=I/O UNIT, W=WRITE

  FILE     NONE             UNUSED/*STRAY*   CHAR*4              2
  SIZE1     105B            EQV              INTEGER             3       4       5
  SIZE2     106B            EQV              INTEGER             3       4       5
  STRAY    NONE             UNUSED/*STRAY*   INTEGER             3
  S1        105B            EQV              INTEGER             3       4       6/I   11/A
  S2        106B            EQV              INTEGER             3       4       6/I   13/A


--PROCEDURES--(LO=M/A/R)                                            D=DEF LINE CF STMT FUNC
 -NAME----TYPE------ARGS----CLASS-----REFERENCES-                   A=ACTUAL ARGUMENT

  NOHEAD                1    SUBROUTINE   13
  PASCAL                1    SUBROUTINE   11


--STATEMENT LABELS--(LO=M/A/R)                                      A=ASSIGN STMT, C=DO STMT,
 -LABEL--ADDRESS---PROPERTIES-----DEF--REFERENCES-                  R=READ, W=WRITE

    100  *NO REFS*                 8


--ENTRY POINTS--(LO=M/A/R)                                          D=DEFINITION
 -NAME---ADDRESS--ARGS------REFERENCES-

  MAPS       208     0          1/D


--NAMELISTS--(LO=M/A/R)                                             D=DEFINITION, R=READ, W=WRITE
 -NAME---ADDRESS---REFERENCES-

  PARAMS     768


--COMMON+EQUIVALENCE--(LO=M/A/R)



--LOCAL EQUIVALENCE--

     ( SIZE2<1>  S2<1> )( SIZE1<1>  S1<1> )


--I/O UNITS--(LO=M/A/R)                                             R=READ, W=WRITE
 -NAME--- PROPERTIES------------REFERENCES-

  TAPE2   FMT/SEQ                  9/W   10/W   12/W
  TAPE3   FMT/SEQ                  8/R
```

Figure 11-13. Reference Map Example (Sheet 1 of 3)

```
--STATISTICS--

  PROGRAM-UNIT LENGTH            107B  =  71
  CM STORAGE USED              60300B  =  24768
  COMPILE TIME                  0.10B SECONDS
```

Block Data Subprogram:

```
--VARIABLE MAP--(LO=M/A/R)                                          A=ARGLIST, C=CTRL OF DO, I=DATA INIT,
 -NAME---ADDRESS--BLOCK-----PROPERTIES-------TYPE---------SIZE---REFERENCES-   R=READ, S=STORE, U=I/O UNIT, W=WRITE

  X           0B  /ANARRAY/                 INTEGER        22       3      4      5/I

--COMMON+EQUIVALENCE--(LO=M/A/R)


  /ANARRAY/ LEVEL = 1, SIZE = 22 WORDS CM
      X<1-22>

--STATISTICS--

  PROGRAM-UNIT LENGTH             0B  =  0
  CM LABELLED COMMON LENGTH      26B  =  22
  CM STORAGE USED              60300B  =  24768
  COMPILE TIME                  0.02B SECONDS
```

Subroutine PASCAL:

```
--VARIABLE MAP--(LO=M/A/R)                                          A=ARGLIST, C=CTRL OF DO, I=DATA INIT,
 -NAME---ADDRESS--BLOCK-----PROPERTIES-------TYPE---------SIZE---REFERENCES-   R=READ, S=STORE, U=I/O UNIT, W=WRITE

  J         1278                            INTEGER              11/C   12     12     12     13     13/C   13
  K         1258                            INTEGER               9/C   10     11/C   13     13/C
  L           0B  /ANARRAY/                 INTEGER        22     3      4      10/S   12     12     12/S   13/W
  M         1248                            INTEGER               8/S    9/C
  SIZE         1  DUMMY-ARG                  INTEGER               2/A    3      8/A


--PROCEDURES--(LO=M/A/R)                                            D=DEF LINE OF STMT FUNC
 -NAME----TYPE------ARGS----CLASS-----REFERENCES-                   A=ACTUAL ARGUMENT

  MAXO    INTEGER   VAR     INTRINSIC     8/A
  MINO    INTEGER   VAR     INTRINSIC     8
```

Figure 11-13. Reference Map Example (Sheet 2 of 3)

```
--STATEMENT LABELS--(LO=M/A/R)                                    A=ASSIGN STMT, C=CC STMT,
 -LABEL--ADDRESS---PROPERTIES-----DEF--REFERENCES-                R=READ, W=WRITE

      1   INACTIVE DO-TERM          12      11/D
      2   INACTIVE DO-TERM          13       9/D


--ENTRY POINTS--(LO=M/A/R)                                        D=DEFINITION
 -NAME---ADDRESS--ARGS------REFERENCES-

  NOHEAD    1038     0          7/D
  PASCAL      58     1          2/D


--DO LOOPS--(LO=M/A/R)
 -LABEL--ADDRESS----PROPERTIES-------------INDEX---FROM-----TO

      2  *NO REFS*  XREF/OUTER               K         9      13
      1  *NO REFS*                          J        11      12
  I/O    *NO REFS*  XREF                     J        13      13


--COMMON+EQUIVALENCE--(LO=M/A/R)


  /ANARRAY/ LEVEL = 1, SIZE = 22 WORDS CM
      L<1-22>


--I/O UNITS--(LO=M/A/R)                                           R=READ, W=WRITE
 -NAME--- PROPERTIES-----------REFERENCES-

  TAPE2   FMT/SEQ                      6/W   13/W


--STATISTICS--

  PROGRAM-UNIT LENGTH            1348  =  92
  CM LABELLED COMMON LENGTH       268  =  22
  CM STORAGE USED              603008  = 24768
  COMPILE TIME                 0.142 SECONDS
```

Figure 11-13. Reference Map Example (Sheet 3 of 3)

OBJECT LISTING

The structure of the object code produced by FORTRAN differs depending on which optimization mode is selected (OPT=0, 1, 2, 3 control statement option).

The FORTRAN compiler produces object code in units called USE blocks. (See the COMPASS reference manual.) These blocks include not only the code produced by compilation of the executable statements in the user's program, but also storage for variables, constants, and compiler-generated temporary entities, as well as other special purpose areas.

Also discussed in this section is the arrangement in memory of user code, library routines, and common blocks after the program is loaded.

NOTE

The information in this subsection is intended only as a guide to interpret object code listings and core dumps. The information does not represent a guaranteed interface specification. Users should avoid any programming techniques which depend on a detailed knowledge of compiler-generated code.

The following description of the arrangement of code and data within main programs, subroutines, and functions does not include the arrangement of data within common blocks because this arrangement is specified by the programmer. However, the diagram of a typical memory layout (in section 9) illustrates the position of blank common and labeled common blocks.

PROGRAM UNIT STRUCTURE

The code within program units is arranged in the following blocks in the order given:

START. A table of file names specified in the PROGRAM statement (main program). Code for primary entry point initialization and for saving A0 (subprograms).

CODE. Code generated by compiling executable statements and code for alternate entry points (ENTRY statement).

LITERL. Contains the following subblocks:

- CON. Storage for read-only constants.

FORMAT. Storage for static FORMAT statements.

TEMPS. Contains the following subblocks:

- ST. Statement temporary values.
- CT. Character statement function argument temporary values.
- IT. Global optimization temporary values.
- OT. Scheduler temporary values.
- VD. Nonconstant formal array-bound expressions and formal character passed-length values.
- LC. Local copies of selected scalar formal parameter values (OPT=2 and 3 only).

APLIST. Actual parameter lists for subprograms called.

IOAPL. Actual parameter lists for I/O subroutines called (this format differs from APLIST.). Array of character descriptors referred to in I/O actual parameter lists (CL. subblock).

NAMLST. Execution time dimension descriptors (RD. block) and descriptors for referenced NAMELIST groups.

VARS. Storage for local variables and arrays not mentioned in a SAVE statement. Each local equivalence class of type CHARACTER begins on a word boundary. Also some compiler-generated temporary cells including:

- DI. DO loop variable increment values.
- DC. DO loop trip count values.

SUB. Address substitution lists for formal parameters.

SUB0. Instructions to be modified for level 0 formal parameter references.

NAMING CONVENTIONS

The names of some system-supplied entities are changed by the compiler to prevent ambiguity for the assembler.

Register Name Conflicts

Variable names which are identical to COMPASS register names can be a source of confusion to the user when reading COMPASS listings, although the assembler can differentiate between the two.

System-Supplied Procedure Names

The name of an intrinsic function called by value is suffixed with a decimal point. The entry point is the symbolic name of the intrinsic function and a decimal point suffix. Examples are: EXP., COS., and CSQRT. The names of all intrinsic functions called by value appear in section 7. The functions in section 7 are not called by value if the name appears either in an EXTERNAL statement, or if the DB=TB option is specified on the FTN5 control statement, or if they are other than mathematical functions.

If the function name appears in an INTRINSIC statement, the entry point is the function name suffixed with an equal sign. Otherwise, the pass by name entry point is the function name with no suffix.

The subroutines listed in section 7 are called by reference.

Listing Format

The object code produced for each program unit is listed following the reference map (if any) for that program unit. The O parameter of the C$ LIST directive (described in appendix E) controls the listing of lines of object code. Object code generated by source lines falling between C$ LIST,O=0 and C$ LIST,O=1 is not listed.

Optimization can cause code to move so that object code for other statements is listed, or code from the desired statements is not listed.

Certain information which can be obtained from the source listing or reference map is not reproduced in the object listing. This information includes:

- Storage allocation for variables and arrays
- Namelist group definitions
- Data initialization translations
- Loader directives

EXECUTION CONTROL STATEMENT

Optional parameters can be included on the control statement that calls into execution a program compiled by FORTRAN. This control statement is normally either the name of the file to which the binary object code was written (LGO is the default) or an EXECUTE statement specifying the name of the main entry point of the program (the name used on the PROGRAM statement or START. if the PROGRAM statement was omitted). The parameters that can be included on this control statement are of four kinds: file names, print limit specification (PL), user parameters, and Post Mortem Dump options.

FILE NAME SUBSTITUTION

FORTRAN 5 provides a method of substituting file names at execution time. File names declared on the PROGRAM statement are associated with files of the same name unless the user substitutes a different name. For example, with the PROGRAM statement:

```
PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)
```

the execution time file names would be:

- INPUT
- OUTPUT
- TAPE1
- TAPE2

Note that specification of file names on the PROGRAM statement is optional; the same file names would occur if the statement PROGRAM TEST1 were used.

However, file names on the PROGRAM statement can be changed for the execution of a program by substitution in the execution control statement. A one-to-one correspondence exists between parameters on this statement and parameters in the PROGRAM statement. For example, using the preceding PROGRAM statement, an execution control statement of the form:

```
LGO(,,DATA,ANSW)
```

would cause the following files to be used:

File name	File used
INPUT	INPUT
OUTPUT	OUTPUT
TAPE1	DATA
TAPE2	ANSW

If a file name in the PROGRAM statement is equivalenced, the logical file name is the file to the right of the equals sign. A corresponding file name in the execution control statement is ignored. For example, using the program statement:

```
PROGRAM TEST3(INPUT,OUTPUT,
*TAPE1=OUTPUT,TAPE2,TAPE3)
```

and the execution control statement:

```
LGO(,,DATA,ANSW)
```

would cause the following substitutions:

File name	File used
INPUT	INPUT
OUTPUT	OUTPUT
TAPE1	OUTPUT
TAPE2	ANSW
TAPE3	TAPE3

The user should not substitute the same file name for two different file names on the PROGRAM statement.

PRINT LIMIT SPECIFICATION

A parameter can be specified on the execution control statement to regulate the maximum number of records that can be written at execution time on the file OUTPUT. The *PL parameter has the same form as the PL parameter specified at compilation time on the FTN5 control statement. If specified on the execution control statement, the PL parameter overrides the value specified either explicitly or by default at compilation time.

The print limit parameter (specified either at compilation time or at execution time) is operative only on files with the name OUTPUT in the first word of their corresponding file information table. Thus, if a file name declared in the PROGRAM statement is superseded at execution time by the file name OUTPUT as described previously, the print limit parameter will be operative on the original file name. Conversely, if the file name OUTPUT is superseded at execution time by another file name, the effect of the print limit parameter is nullified. Some examples of *PL parameter usage are as follows:

```
LGO(*PL=2000)

EXECUTE(,FILE1,OUTPUT,FILE2,*PL=1000)
```

USER PARAMETERS

User parameters that specify values to be accessible from the user program can be included on the execution control statement. These parameters must appear after any file names, *PL specification, and PMD parameters. The format of a user parameter is as follows:

```
name [=value]
```

name Parameter name consisting of 1 through 7 numbers or letters.

value Optional parameter value; string consisting of numbers, letters, or the character *. Can also be a literal, containing any character, delimited by dollar signs ($).

An example of an execution control statement containing user parameters is as follows:

```
LGO (INFILE,*PL=1000,V1=25,V2=ABCD)

LGO (X=$@@$)
```

User parameters can be accessed from the FORTRAN program using the GETPARM subroutine call described in section 7. (Under NOS, user parameters on the execution control statement must be in product set format. Refer to the NOS reference manual for information on product set format.)

POST MORTEM DUMP PARAMETERS

The Post Mortem Dump facility, described in section 10, provides a symbolic dump of program variables, an error analysis, and traceback information on the occurrence of a program abort. Parameters can be specified on the execution control statement to control Post Mortem Dump output and to specify limits on array subscripts.

Post Mortem Dump Output Parameter

The Post Mortem Dump output parameter specifies the destination and format of the dump. This parameter has the following format:

```
*OP=list
```

The option list consists of one or more of the following, not separated by separators:

A All active routines are included in the dump. An active routine is one that has been executed but is not necessarily in the traceback chain.

F Valid for interactive jobs only; Post Mortem Dump output is sent to file PMDUMP when the job is executed with file OUTPUT connected. The following message appears at the terminal at the time of the dump:

```
*POST PROCESSOR OUTPUT WILL BE
    FOUND ON THE FILE PMDUMP*
```

T Valid for interactive jobs only; a condensed form of the dump is sent to the terminal. File OUTPUT must be connected.

If the *OP parameter is omitted, dumps are sent to file PMDUMP when executing from a terminal with file OUTPUT connected. When the dump occurs, the following message appears at the terminal:

```
*POST PROCESSOR OUTPUT WILL BE FOUND ON
    THE FILE PMDUMP.*

*FOR A SUMMARY AT THE TERMINAL,

*RERUN JOB, REPLACING LGO CARD BY
    LGO,*OP=T.
```

An example of a Post Mortem Dump output control parameter is as follows:

```
LGO (*OP=AF)
```

Subscript Limit Specification

Subscript limits can be specified on the execution control statement to control the printing of arrays by Post Mortem Dump facility. This has the same effect as a PMDARRY call (section 10). The subscript limit parameter has the following format:

```
*DA=i+j+k+l+m+n+o
```

The integers i, j, k, l, m, n, and o specify the maximum values of the subscripts of arrays to be printed; i through o represent the first through seventh subscripts respectively. Subscript limits can be omitted from the list to control printing as follows:

If o is omitted, 7-dimensional arrays are not printed.

If n and o are omitted, 6- through 7-dimensional arrays are not printed.

If m through o are omitted, 5- through 7-dimensional arrays are not printed.

If l through o are omitted, 4- through 7-dimensional arrays are not printed.

If k through o are omitted, 3- through 7-dimensional arrays are not printed.

If i only is specified, only 1-dimensional arrays are printed.

For example, if the control statement:

```
LGO (*DA=2+5)
```

is used, only 1- and 2-dimensional arrays will be dumped. If the statement

```
DIMENSION ARAY(20,20)
```

appears in the source program, then the following elements will be printed by Post Mortem Dump:

```
ARAY(1,1), ARAY(2,1)
ARAY(1,2), ARAY(2,2)
ARAY(1,3), ARAY(2,3)
ARAY(1,4), ARAY(2,4)
ARAY(1,5), ARAY(2,5)
```

EXAMPLES 12

The first part of this section contains sample deck structures, including control statements, illustrating compilation and execution of FORTRAN programs. The second part contains sample executable programs illustrating various features of FORTRAN. Examples of input and output are included.

SAMPLE DECK STRUCTURES

Following are some typical deck structures that can be used for compiling and executing FORTRAN programs. Refer to the operating system reference manual for details of control statements.

FORTRAN SOURCE PROGRAM WITH CONTROL STATEMENTS

Figure 12-1 shows a deck structure for compiling and executing a FORTRAN program that contains a function and a subroutine.

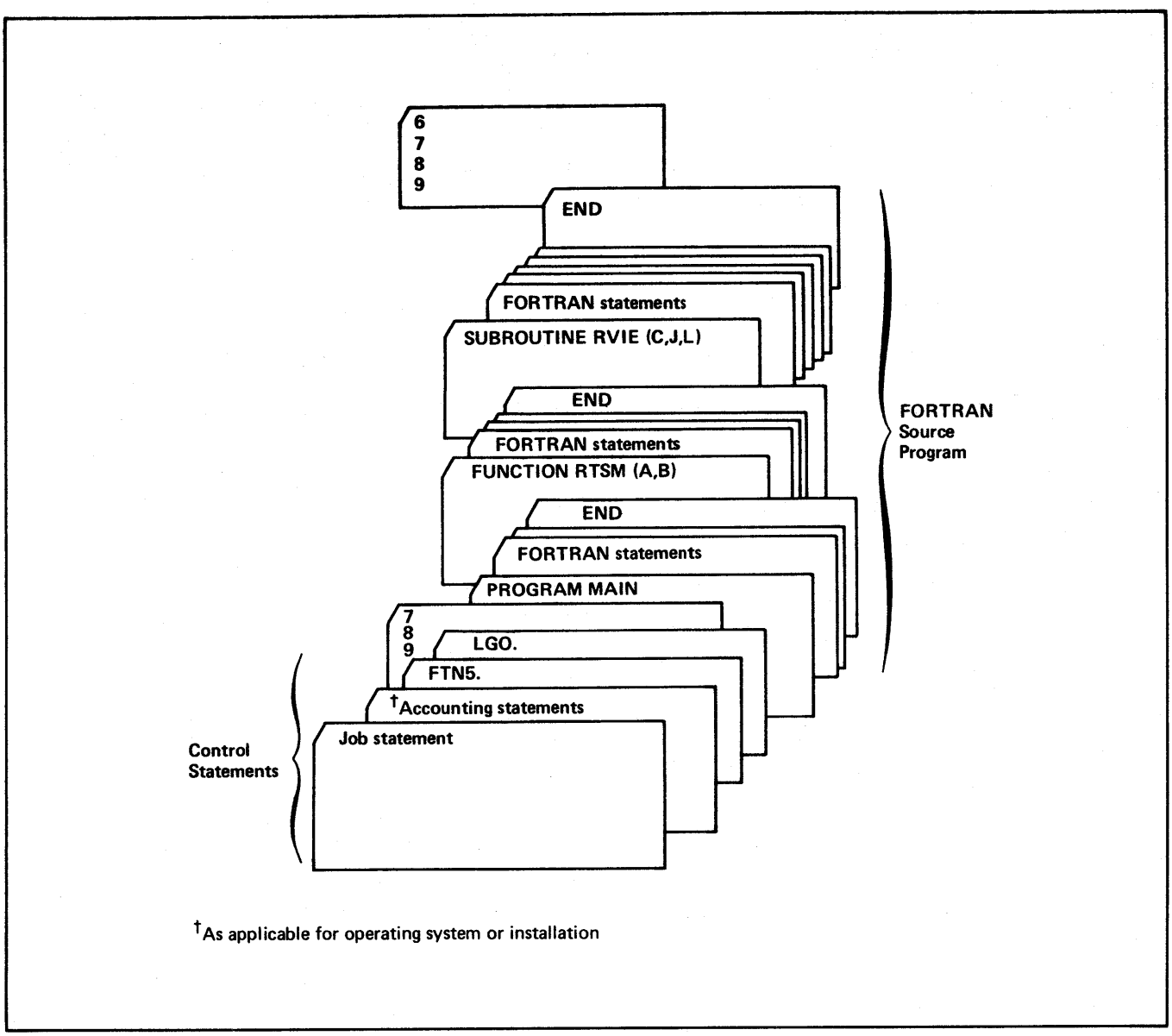


Figure 12-1. FORTRAN Source Program With Control Statements

COMPILATION ONLY

Figure 12-2 shows a deck structure for compiling a program; the program is not executed after compilation.

OPT=0 COMPILATION

Figure 12-3 illustrates a deck structure for compiling a program in OPT=0 mode. No binary object file is produced and no execution occurs.

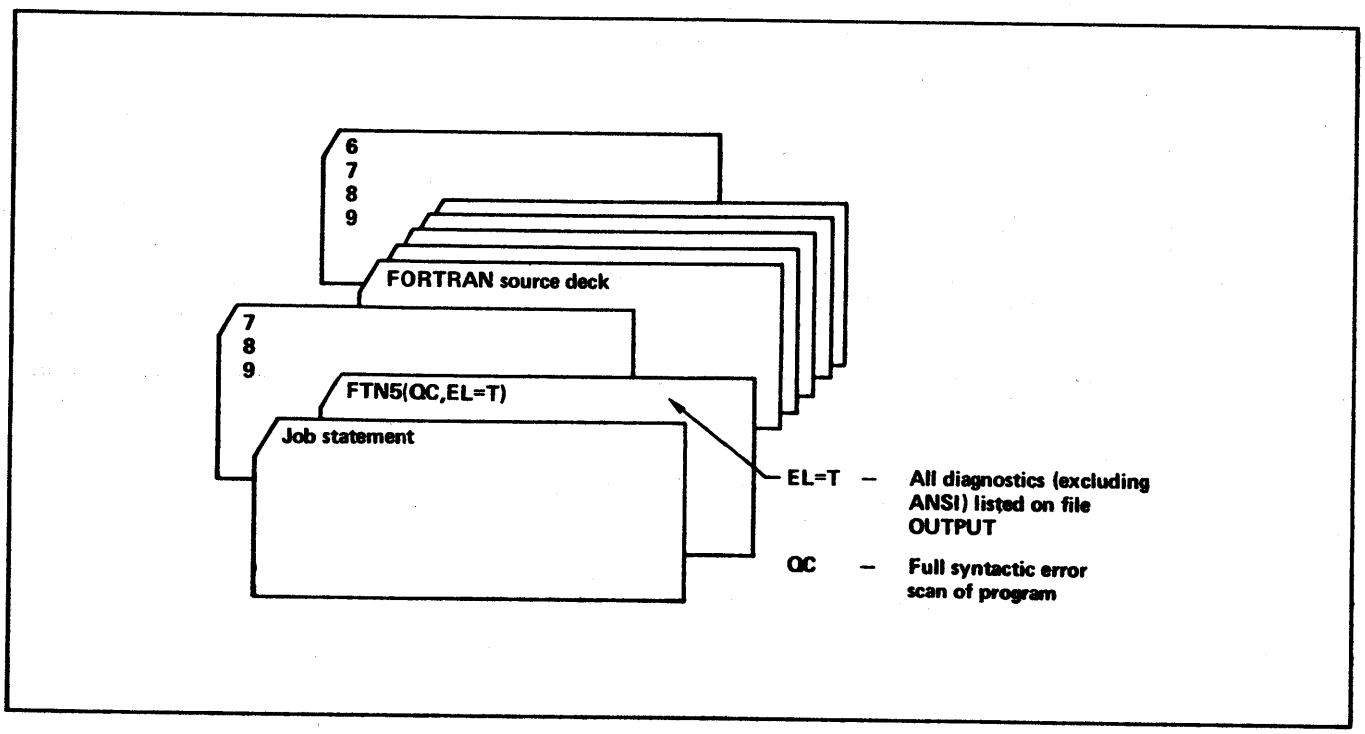


Figure 12-2. Compilation Only

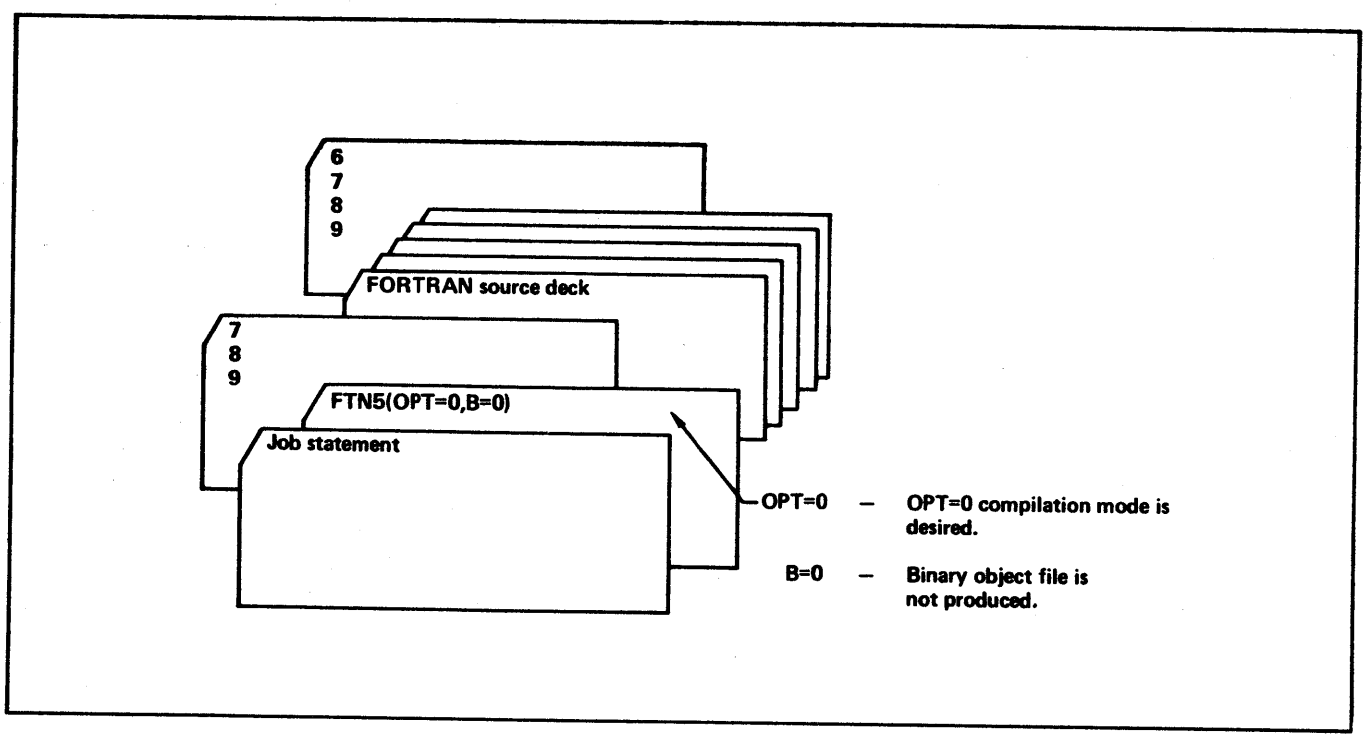


Figure 12-3. OPT=0 Compilation

COMPILATION AND EXECUTION

Figure 12-4 illustrates a deck structure for compiling and executing a program that reads data from cards.

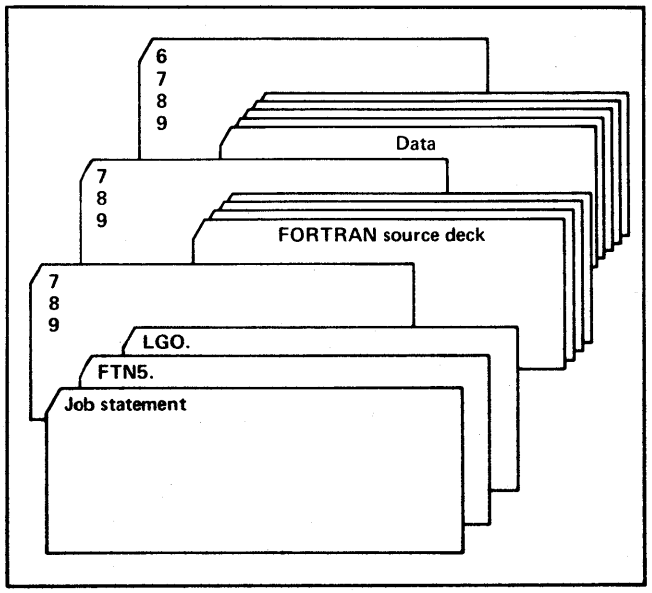


Figure 12-4. Compilation and Execution

FORTRAN COMPILATION WITH COMPASS ASSEMBLY AND EXECUTION

Figure 12-5 illustrates a deck structure containing a FORTRAN and a COMPASS program unit. The FORTRAN and COMPASS source decks can be in any order. COMPASS source decks must begin with a line containing the word:

IDENTΔ

in columns 11 through 16. Columns 1 through 10 of the ident line must be blank.

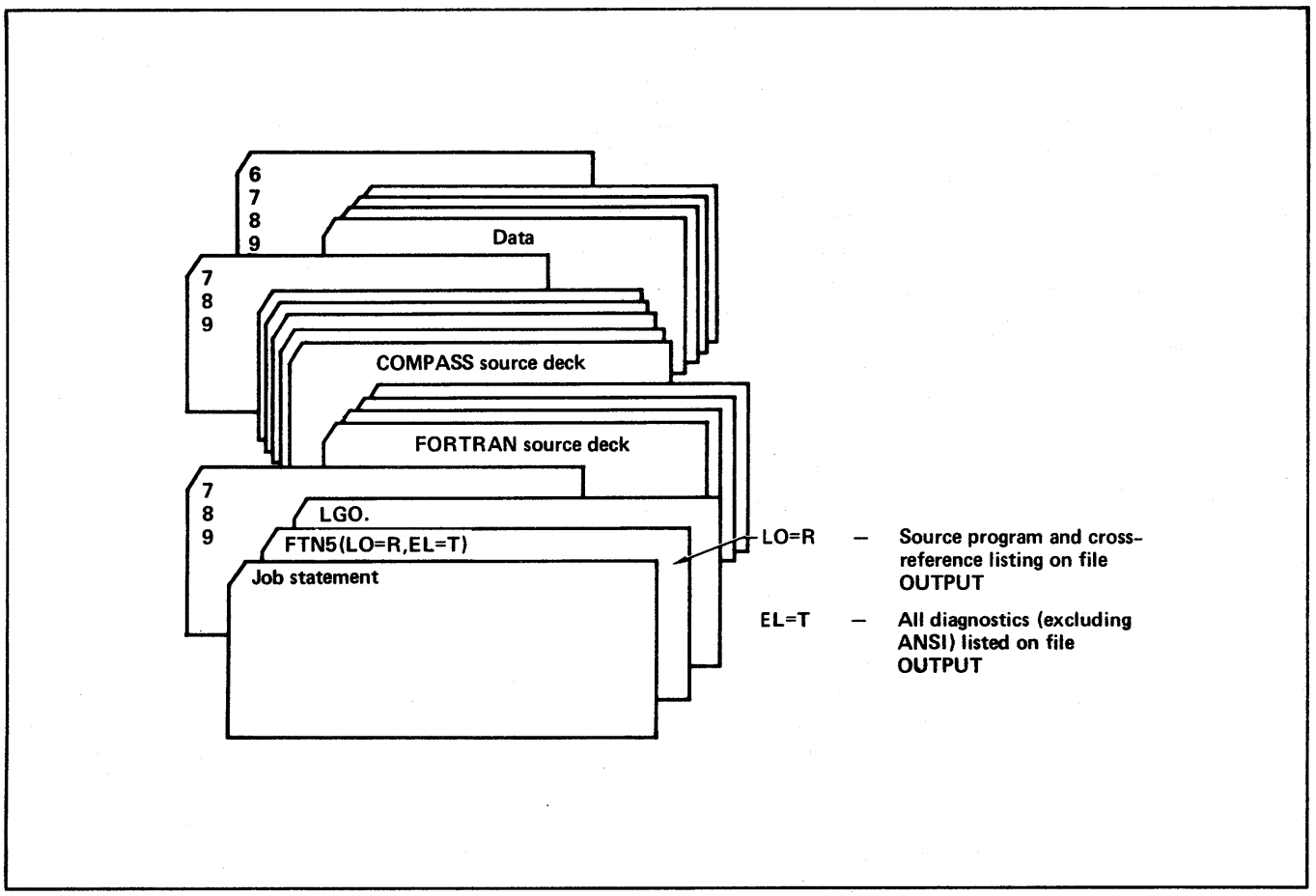


Figure 12-5. Compilation With COMPASS Assembly and Execution

COMPILATION AND EXECUTION WITH FORTRAN SUBROUTINE AND COMPASS SUBPROGRAM

Figure 12-6 illustrates a deck structure containing a FORTRAN subroutine, and a COMPASS subprogram, showing the COMPASS IDENT and ENTRY statements. In this example, the LGO statement specifies the output file (as described in section 11).

COMPILATION WITH BINARY CARD OUTPUT

Figure 12-7 illustrates a deck structure to compile and produce a binary object deck.

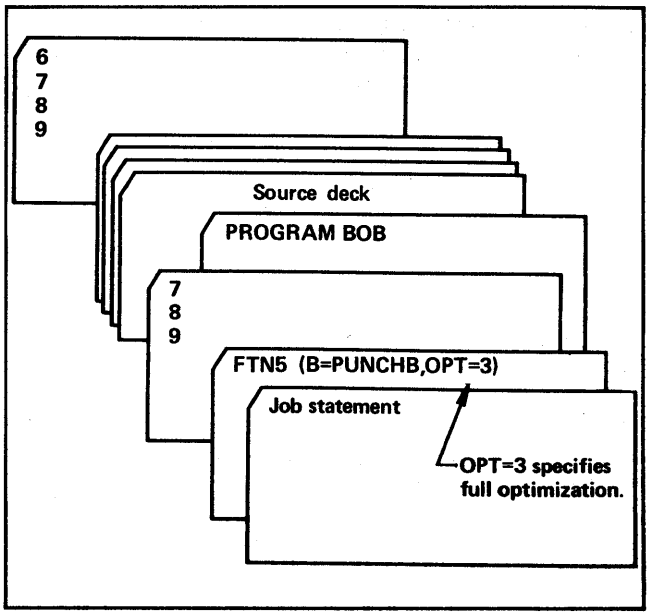


Figure 12-7. Compilation With Binary Card Output

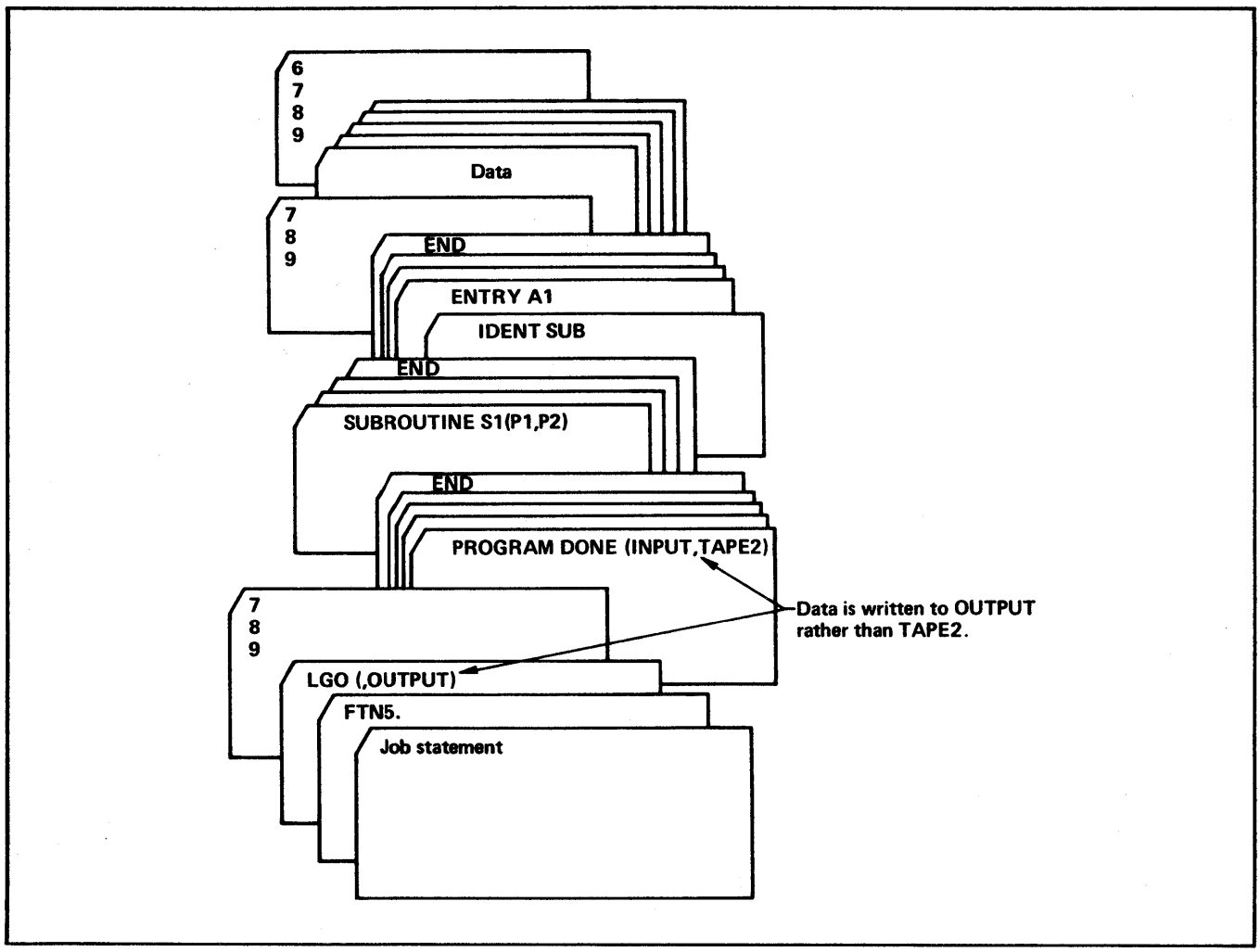


Figure 12-6. Compilation and Execution With FORTRAN Subroutines and COMPASS Subprogram

LOADING AND EXECUTION OF BINARY PROGRAM

Figure 12-8 illustrates a deck structure to load and execute a binary object program. The MAP(OFF) statement suppresses the load map.

COMPILATION AND EXECUTION WITH RELOCATABLE BINARY DECK

Figure 12-9 illustrates a deck structure to compile a FORTRAN program and load and execute a binary program along with the FORTRAN program.

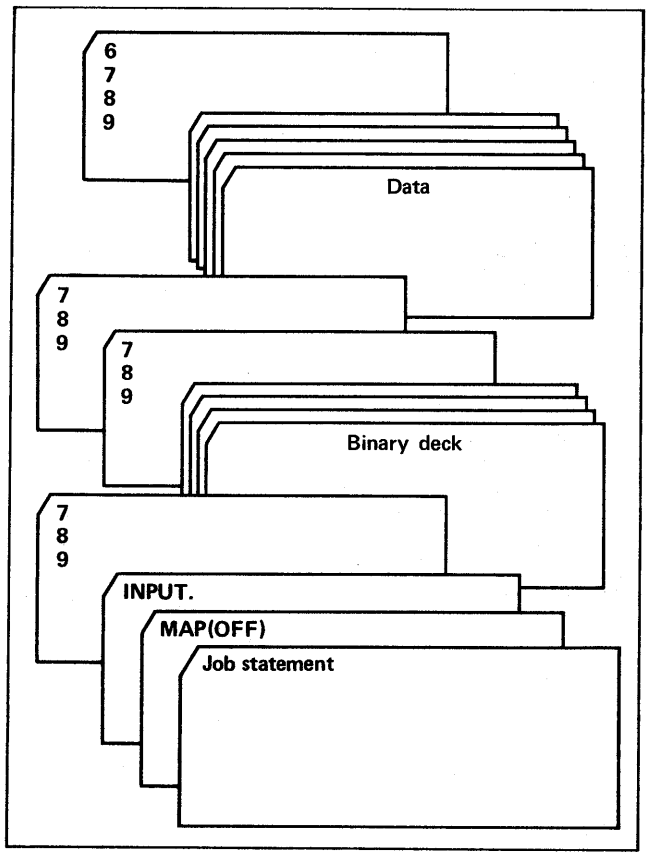


Figure 12-8. Loading and Execution of Binary Program

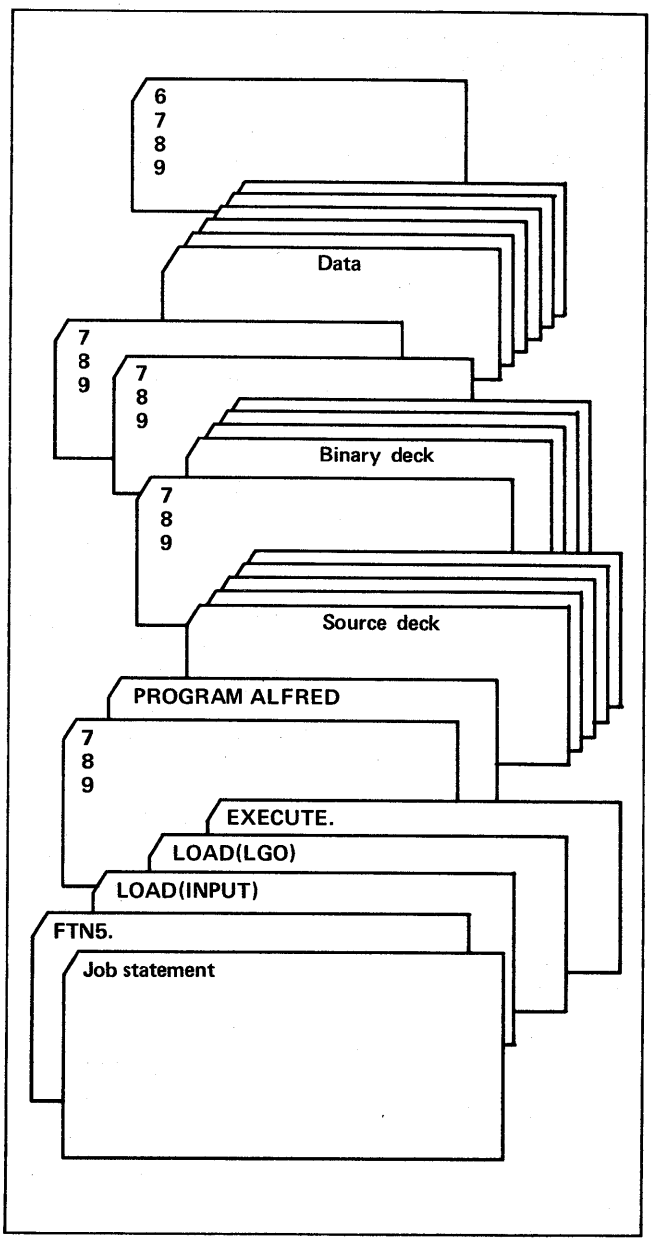


Figure 12-9. Compilation and Execution With Relocatable Binary Deck

COMPILATIONS AND TWO EXECUTIONS WITH DIFFERENT DATA DECKS

Figure 12-10 illustrates a deck structure to compile a program and to execute the program twice with two different data decks. Output from the two executions is sent to separate output files.

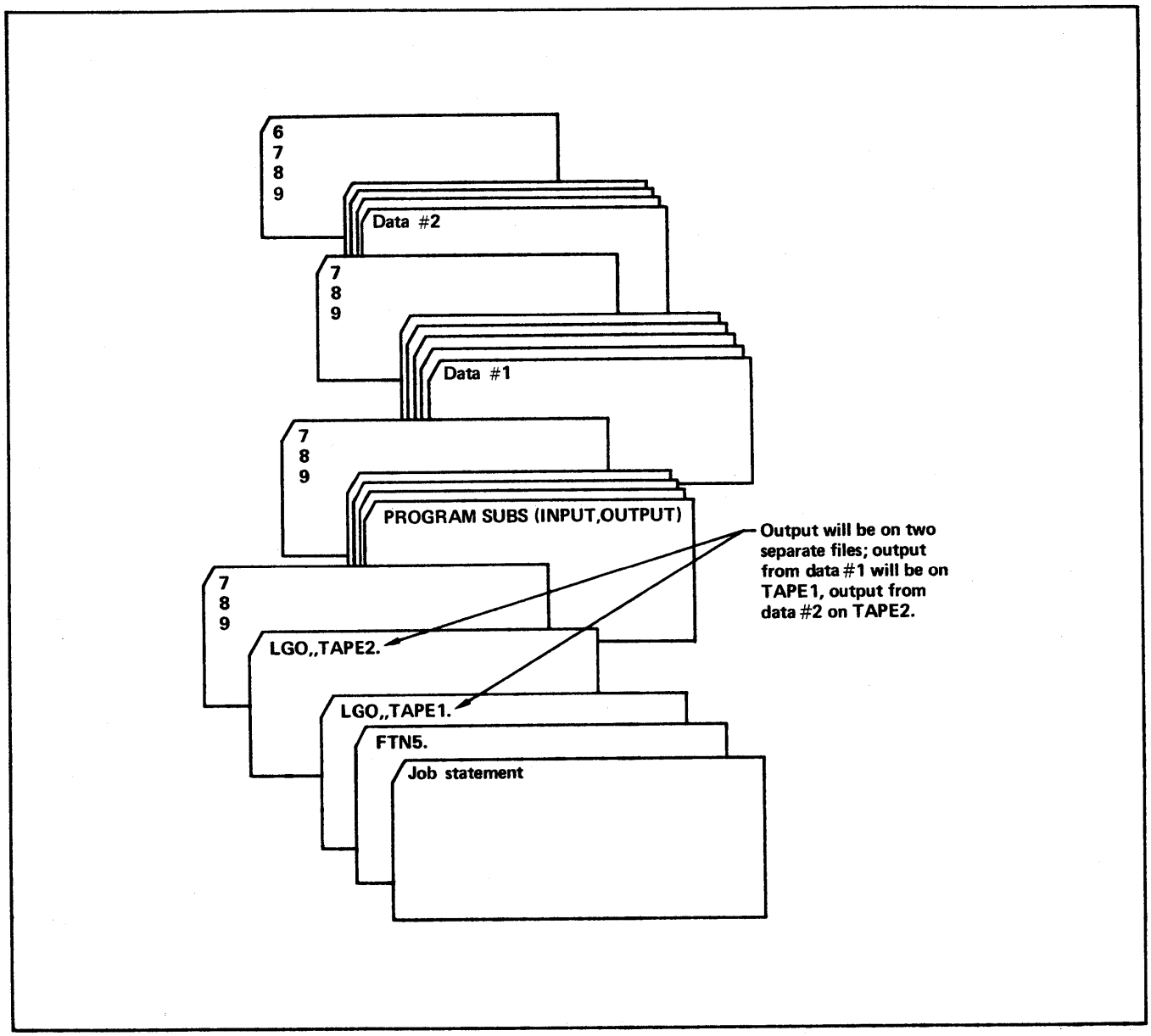


Figure 12-10. Compilation and Execution With Different Data Decks

PREPARATION OF OVERLAYS

Figure 12-11 illustrates a deck structure to compile, load and execute a program containing overlays.

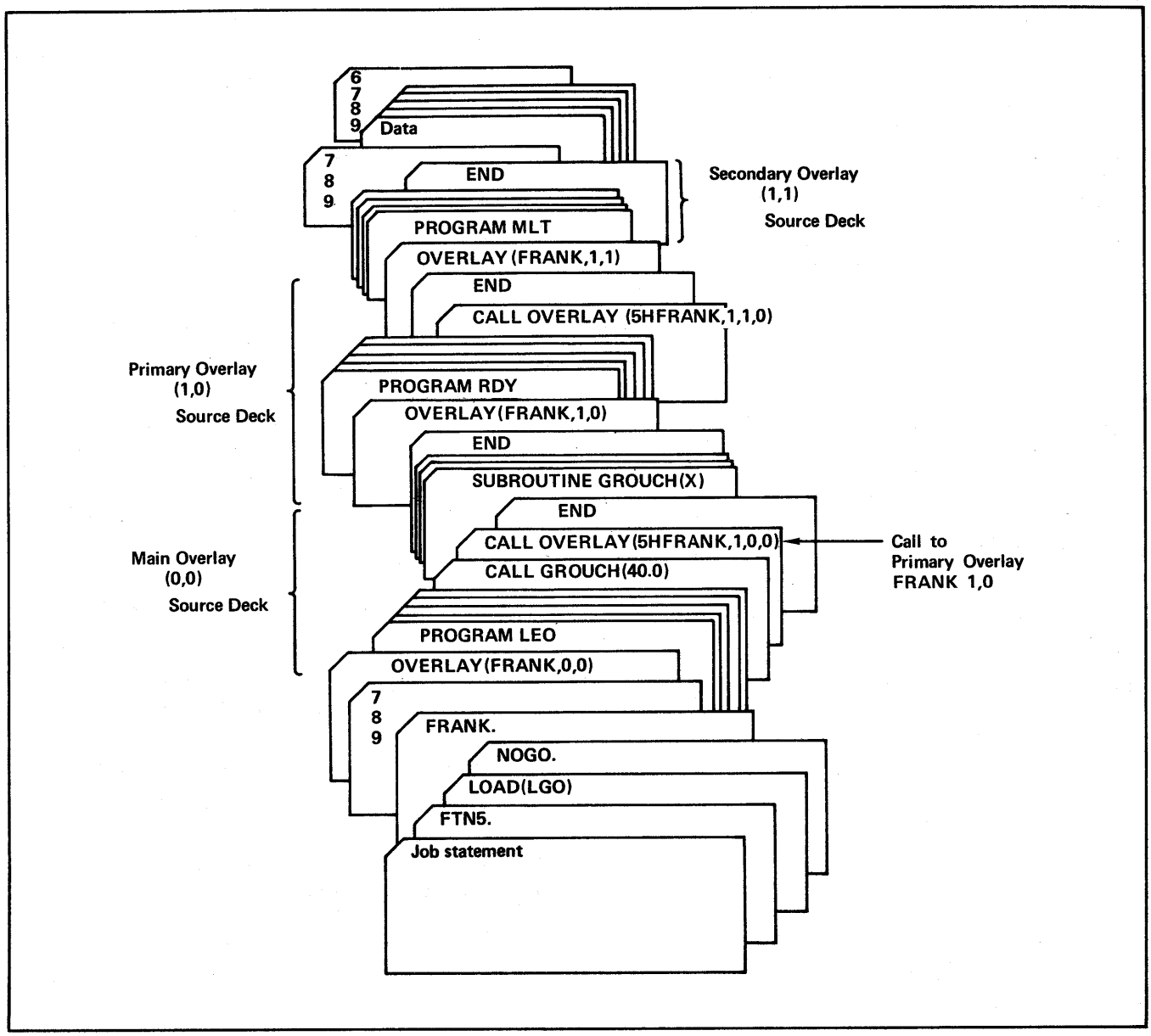


Figure 12-11. Preparation of Overlays

COMPILATION AND TWO EXECUTIONS WITH OVERLAYS

Figure 12-12 illustrates a deck structure to compile an overlay and to execute the overlay two times.

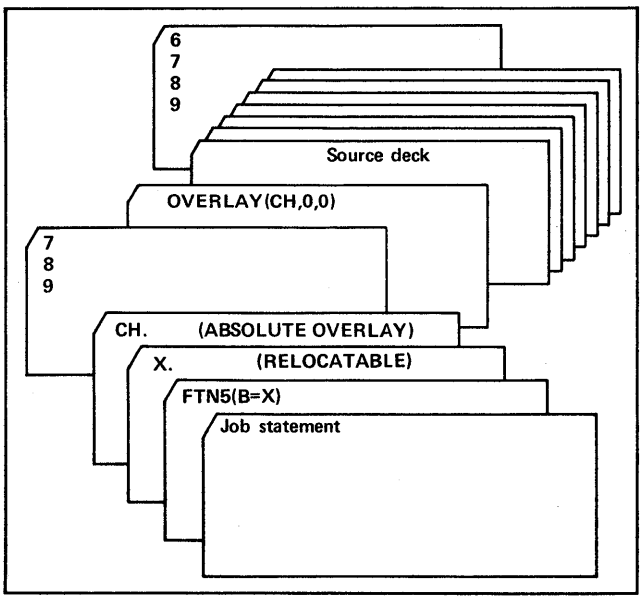


Figure 12-12. Compilation and Two Executions With Overlays

SAMPLE PROGRAMS

This subsection shows sample FORTRAN programs which illustrate various features of the FORTRAN language.

PROGRAM OUT

Program OUT, shown in figure 12-13, illustrates the following FORTRAN features:

Control statements for batch execution

WRITE and PRINT statements

Carriage control

PROGRAM statement

The control statement:

BIRD, T10.

is the job statement. A job statement must precede every job. BIRD is the job name. T10 specifies a maximum of 10 seconds of central processor time (can be either octal or decimal, depending on installation option).

The statement:

FTN5.

specifies the FORTRAN compiler. The default parameters (described in section 10) are used. Since no alternative files are specified on the FTN5 control statement, the FORTRAN compiler reads from file INPUT and outputs to files OUTPUT and LGO. Listings, diagnostics, and maps are written to OUTPUT and the relocatable object code is written to LGO.

The statement:

LGO.

causes the binary object code to be loaded and executed.

The statement:

7/8/9

separates control statements from the remainder of the job deck (file INPUT). This statement contains a 7, 8, and 9 multipunched in column 1; it follows the control statements in every batch job.

The OPEN statement (line 2) associates unit 6 with file OUTPUT.

The WRITE statement (line 6) outputs the variable INK to file OUTPUT. The format specification is included in the WRITE statement. If the following PRINT statement had been used instead of WRITE:

```
PRINT '(I5, "= INK OUTPUT BY PRINT",
*"STATEMENT")', INK
```

the OPEN statement would not be needed. The specification uses quotes to delimit the literal and the carriage control character 1 to cause the line to be printed at the top of a page.

Lines 6 and 7 print the variable INK. In both output statements, a blank carriage control character is specified to cause single spacing. Line 6 uses the specification 1X

```
BIRD,T10.
FTN5.
LGO.
7/8/9 IN COLUMN 1
      PROGRAM OUT
      OPEN (6, FILE='OUTPUT')
      PRINT 100
100   FORMAT ("1 THIS WILL PRINT AT THE TOP OF A PAGE")
      INK= 2000 + 4000
      WRITE (6,'(1X,I4," = INK OUTPUT BY WRITE STATEMENT")') INK
      PRINT '(1H ,I4, 30H = OUTPUT FROM PRINT STATEMENT)', INK
      STOP
      END
6/7/8/9 IN COLUMN 1
```

Figure 12-13. Program OUT With Control Statements

to produce a blank in column 1; line 7 uses the specification 1H for the same effect.

The 6/7/8/9 card contains the characters 6, 7, 8, and 9 multipunched in column 1. It is the last card in every job deck (INPUT file), indicating to the system the end of the job.

Output from program OUT is shown in figure 12-14.

```
 THIS WILL PRINT AT THE TOP OF A PAGE
6000 = INK OUTPUT BY WRITE STATEMENT
6000 = OUTPUT FROM PRINT STATEMENT
```

Figure 12-14. Program OUT Output

PROGRAM B

Program B, shown in figure 12-15, generates a table of 64 characters. The internal bit configuration of any character can be determined by its position in the table. Each character occupies six bits.

Features illustrated in this example include:

Octal constants

Simple DO loop

PRINT statement

FORMAT with /,I,X and A editing

Character constant as a format specifier

The PRINT statement (line 2) has no output list; it prints out the heading at the top of the page using the information provided by the format specification. The 1 is the carriage control character, and the two slashes cause one line to be skipped before the next string is printed. The slash at the end of the format specification skips another line before the program output is printed.

The DO loop (lines 4 through 6) generates numbers 0 through 7 (note that a DO index can be zero). The PRINT statement (line 5) prints 0 through 7 (the value of J) on the left and the 8 characters in NCHAR on the right. The first iteration of the DO loop prints NCHAR as it appears on line 3. The octal value 01 is a display code A, 02 is a B, 03 is a C, etc. Line 6 adds the octal constant 1010101010101010000O to NCHAR; when this is printed on the second iteration of the DO loop, the octal value 10 is printed as a display code H, 11 as I, 12 as J, etc. Compare these values with the character set listed in appendix A.

Output from program B is shown in figure 12-16.

```
          TABLE OF INTERNAL VALUES

             01234567

          0 :ABCDEFG
          1 HIJKLMNO
          2 PQRSTUVW
          3 XYZ01234
          4 56789+-*
          5 /()$= ,.
          6 #[]%"_!&
          7 '?<>@\^;
```

Figure 12-16. Program B Output

PROGRAM STATES

Program STATES, shown in figure 12-17, reads employee names and home states, ignoring all but the first two letters of the state name. If the state name starts with the letters CA, the name is printed. This program illustrates character handling.

The first PRINT statement (line 3) directs the printer to start a new page, print the heading NAME, and skip 3 lines.

The READ statement (line 5) reads the last name into LNAME, first name into FNAME, home state into STATE, and tests for end-of-file.

```
      PROGRAM STATES
      CHARACTER*10 FNAME, LNAME, STATE
      PRINT 1
1     FORMAT (1H1, 5X, 4HNAME, ///)
3     READ (*, '(3A)', END=99) LNAME,
     X FNAME, STATE
C
C  IF FIRST TWO CHARACTERS OF STATE ARE CA
C  PRINT LAST NAME AND FIRST NAME
C
      IF (STATE(1:2) .EQ. 'CA') THEN
       PRINT '(5X, 2A)', LNAME, FNAME
      ENDIF
      GO TO 3
   99 STOP
      END
```

Figure 12-17. Program STATES

```
      PROGRAM B
      PRINT '("1TABLE OF INTERNAL VALUES",//,"    01234567",/)'
      NCHAR= 0"00 01 02 03 04 05 06 07 00 00"
      DO 3 J = 0,7
      PRINT '(I3, 1X, A8)', J, NCHAR
3     NCHAR= NCHAR + 0"10 10 10 10 10 10 10 10 00 00"
      STOP
      END
```

Figure 12-15. Program B

The relational operator .EQ. tests to determine if the first two letters read into variable STATE match the two letters of the constant 'CA'. If a match occurs, FNAME and LNAME are printed.

Sample input and output for program STATES are shown in figure 12-18.

Input:

```
BROWN,      PHILLIP M.CA
BICARDI,    R. J.       KENTUCKY
CROWN,      SYLVIA      CAL
HIGENBERF,ZELDA         MAINE
MUNCH,      GARY G.     CALIF
SMITH       SIMON       CA
DEAN,       ROGER       GEORGIA
RIPPLE      SALLY       NEW YORK
JONES       STAN        OREGON
HEATH       BILL        NEW YORK
```

Output:

```
NAME

BROWN,      PHILLIP M.
CROWN,      SYLVIA
MUNCH,      GARY G.
SMITH       SIMON
```

Figure 12-18. Sample Input and Output for Program STATES

PROGRAM EQUIV

Program EQUIV, shown in figure 12-19, places values in variables that have been equivalenced and prints these values using the NAMELIST statement. The following features are illustrated:

EQUIVALENCE statement
NAMELIST statement

Line 2 equivalences two real variables X and Y; the two variables share the same location in storage, which can be referred to as either X or Y. Any change made to one

```
      PROGRAM EQUIV
      EQUIVALENCE (X,Y), (Z,I)
      NAMELIST /OUT/ X, Y, Z, I
      OPEN (6, FILE='OUTPUT')
      X= 1.
      Y= 2.
      Z= 3.
      I= 4
      WRITE (6,OUT)
      STOP
      END
```

Figure 12-19. Program EQUIV

variable changes the value of the others in an equivalence group as illustrated by the output of the WRITE statement, in which both X and Y have the value 2.. The storage location shared by X and Y contains first 1. (X=1.), then 2. (Y=2.).

The real variable Z and the integer variable I are equivalenced, and the same location can be referred to as either real or integer. Since integer and real internal formats differ, however, the output values will not be the same.

For example, the storage location shared by Z and I contained first 3.0 (real value), then 4 (integer value). When I is output, no problem arises; an integer value is referred to by an integer variable name. However, when this same integer value is referred to by a real variable name, the value 0.0 is output, because the internal formats of real and integer values differ. The integer and real internal formats are shown in figure 12-20.

Although a value can be referred to by names of different types, the internal bit configuration does not change. An integer value output as a real variable has a zero exponent and its value is zero.

When variables of different types are equivalenced, the value in the storage location must agree with the type of the variable name, or unexpected results might be obtained.

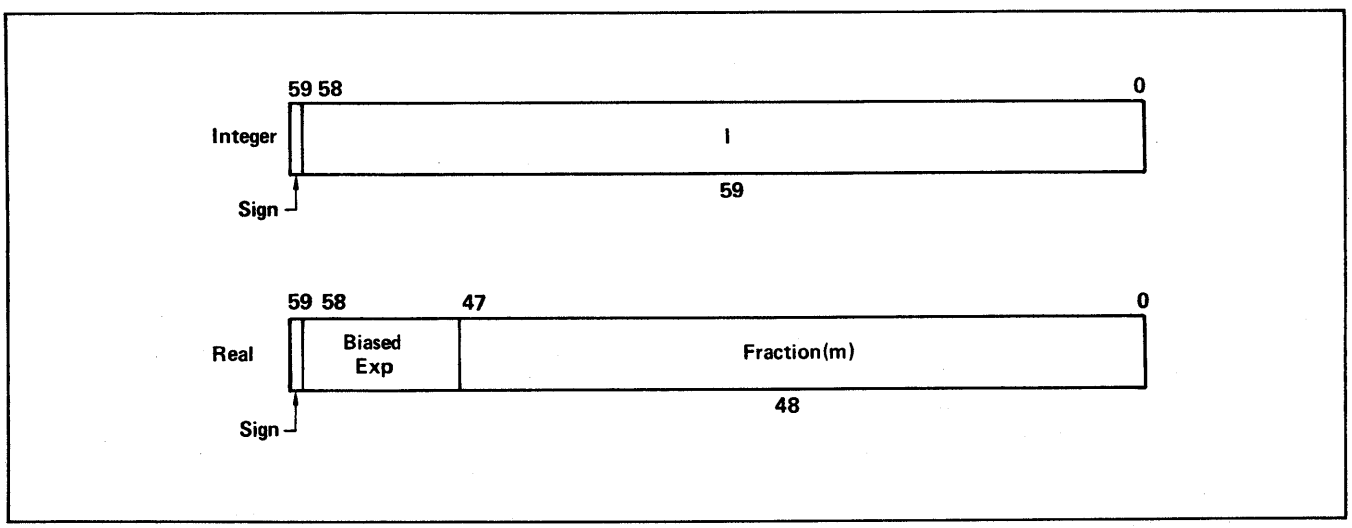


Figure 12-20. INTEGER and REAL Internal Formats

This NAMELIST WRITE statement (line 10) outputs both the name and the value of each member of the NAMELIST group OUT defined in the statement NAMELIST/OUT/X,Y,Z,I. The NAMELIST group is preceded by the group name, OUT, and terminated by the characters $END. Output is shown in figure 12-21.

```
 $OUT
 X       = .2E+01,
 Y       = .2E+01,
 Z       = 0.0,
 I       = 4,
 $END
```

Figure 12-21. Program EQUIV Output

PROGRAM COME

Program COME, shown in figure 12-22, places variables and an array in common and declares another variable and array equivalent to the first element in common. It places the numbers -1 through -12 in each element of the array IA and outputs values in common using the NAMELIST statement. Features illustrated include:

COMMON and EQUIVALENCE statements

NAMELIST statement

Negative subscript

Negative DO loop parameters

Variables are stored in common in the order of appearance in the COMMON statement: A,B,C,D,F,G,H. All variables with the exception of G are declared integer. G is implicitly typed real.

The EQUIVALENCE statement assigns the first element of the arrays IA and E to the same storage location as the variable A. The subscript of IA has a lower bound of -12. Since A is in common, E and IA will be in common. Variables and array elements are assigned storage as shown in figure 12-23.

```
      PROGRAM COME
      COMMON A,B,C,D, F,G,H
      INTEGER A,B,C,D,E(3,4),F, H,IA(-12:-1)
      EQUIVALENCE (A, E, IA)
      NAMELIST /V/ A,B,C,D,E,F,G,H,IA
C
      OPEN (6, FILE='OUTPUT')
      DO 2 J=-1, -12, -1
2     IA(J)= J
      WRITE (6,V)
C
      STOP
      END
```

Figure 12-22. Program COME

The DO loop places values -1 through -12 in IA using a negative DO index. The first element of IA (indexed by -12) shares the same location as the first element of E. This location is also shared by A. IA(-11) is equivalent to E(2,1) and B; IA(-10) is equivalent to E(3,1) and C, and so forth.

Any change made to one member of an equivalence group changes the value of all members of the group. When -12 is stored in IA(-12), both E(1,1) and A have the value -12. When -11 is stored in IA(-11), B and E(2,1) have the value -11. Although B and E(2,1) are not explicitly equivalenced to IA(-11), equivalence is implied by their position in common.

The implied equivalence between the array elements and variables is illustrated by the output shown in figure 12-24.

The NAMELIST statement is used for output. A NAMELIST group, V, containing the variables and arrays A,B,C,D,E,F,G,H,IA is defined. The NAMELIST WRITE statement, WRITE(6,V), outputs all the members of the group in the order of appearance in the NAMELIST statement. Array E is output on one line in the order in which it is stored in memory. There is no indication of the number of rows and columns (3,4).

G is equivalent to E(3,2) and yet the output for E(3,2) is 6 and G is 0.0. G is type real and E is type integer. When two names of different types are used for the same element, their values will differ because the internal bit configuration for type real and type integer differ. (Refer to Program EQUIV.)

Output from program COME is shown in figure 12-24.

Relative Address	0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
	I											
	E(1,1)	E(2,1)	E(3,1)	E(1,2)	E(2,2)	E(3,2)	E(1,3)	E(2,3)	E(3,3)	E(1,4)	E(2,4)	E(3,4)
	A	B	C	D	F	G	H					
	IA(-12)	IA(-11)	IA(-10)	IA(-9)	IA(-8)	IA(-7)	IA(-6)	IA(-5)	IA(-4)	IA(-3)	IA(-2)	IA(-1)

Figure 12-23. Storage Layout for Variables in Program COME

```
$V
A       = -12,
B       = -11,
C       = -10,
D       = -9,
E       = -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1,
F       = -8,
G       = 0.0,
H       = -6,
IA      = -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2,  -1,
$END
```

Figure 12-24. Program COME Output

PROGRAM LIBS

Program LIBS, shown in figure 12-25, illustrates the following features:

Use of FORTRAN library subroutines and intrinsic functions

EXTERNAL used to pass a library subroutine name as a parameter to another library routine

INTRINSIC used to pass an intrinsic function name as a parameter to another library routine

Division by zero

LEGVAR function used to test for overflow or divide error conditions

```
      PROGRAM LIBS
C
      CHARACTER*10 TODAY, CLOCK, DATE, TIME
      EXTERNAL DATE
      INTRINSIC SQRT, SIN
C
      TODAY= DATE()
      CLOCK= TIME()
C
      PRINT 2, TODAY, CLOCK
2     FORMAT ('1TODAY= ', A, ' CLOCK= ', A)
C
      TYME= SECOND()
      CALL RANGET (SEED)
      Y= FUNC(SQRT)
      Y1= FUNC(SIN)
C
      PRINT 3, TYME, Y, Y1, SEED, SEED
3     FORMAT (' THE ELAPSED CPU TIME IS',G14.5,' SECONDS.'//,' SQRT(2.4
     * )/PI = ',G14.5,/' SIN(2.4)/PI = ',G14.5,/' THE INITIAL VALUE OF T
     *HE RANF SEED IS',O22,', OR',/G30.15,' IN G30.15 FORMAT.')
C
      Y= 0.0
      WOW= 7.2/Y
      IF (LEGVAR(WOW) .NE. 0) PRINT 4, WOW
4     FORMAT (1H0,50(2H*-)/' DIVIDE ERROR, WOW PRINTS AS:',G10.2)
      STOP
      END
      FUNCTION FUNC(F)
      FUNC= F(2.4)/3.14159
      RETURN
      END
```

Figure 12-25. Program LIBS

The following functions and subroutines are used in LIBS:

DATE

TIME

SECOND

RANGET

SQRT

SIN

DATE is a library function which returns the date entered by the operator from the console.

SQRT is an intrinsic function that calculates the square root of its argument. SIN is an intrinsic function that calculates the sine of its argument. These functions are declared INTRINSIC so that they can be passed as arguments to a subprogram.

The PRINT statement in line 10 prints the date and time. The arguments TODAY and CLOCK are declared character with length 10 because the DATE and TIME functions each return 10 characters. The leading and trailing blanks appear with the 10 characters returned by the subroutine DATE, because the operating system formats the date in this manner. (The date format is system and installation dependent.) The value returned by TIME is changed by the system once a second, and the position of the digits remains fixed; a leading blank always appears.

When SECOND is called (line 13), the variable name TYME is used. A variable name cannot be spelled the same as an intrinsic function name if that intrinsic function is used in the same program unit. If program LIBS had not called the function TIME, a variable name could be spelled TIME.

Library subroutine RANGET returns the seed used by the random number generator RANF. If RANGET is called after RANF has been used, RANGET will return the value currently being processed by the random number generator. With the library subroutine RANSET, this same value could be used to initialize the random number generator at a later date.

The PRINT statement in line 18 prints out the values returned by the routines SECOND, FUNC, and RANGET.

Lines 25 through 27 illustrate the use of the library function LEGVAR within an IF statement to test the validity of division by zero. LEGVAR checks the variable WOW. This function returns a result of -1 if the variable is indefinite, +1 if it is out of range, and 0 if it is normal. Comparing the value returned by LEGVAR with 0 shows that the number is either indefinite or out of range. The output R shows the variable is out of range.

NOTE

This example will not work on a CYBER 76/176 or 7600 machine because division by zero causes an immediate program interrupt before LEGVAR can be called.

The line of *- on the output is produced by the FORMAT specification in statement number 4: 50(2H*-).

Output from program LIBS is shown in figure 12-26.

PROGRAM ADD

Program ADD, shown in figure 12-27, illustrates the use of internal files. Any character variable or array can be treated as an internal file. Input and output for internal files is performed by formatted READ and WRITE statements. Program ADD uses a formatted READ statement to read data from an internal file.

Read

A formatted READ statement for an external file places the image of each record read into an input buffer. Compiler routines convert the character string in the record into floating-point, integer, or logical values, as specified by the FORMAT statement, and store these values in the locations associated with the variables named in the list.

With internal files, the specified file (character variable, substring, or array) is used as the input buffer. The record length is equal to the length, in characters, of the variable if the file is a character variable, of a single array element if the file is an array, or of the substring.

With external files, when the format specification indicates a new record is to be processed (by a slash or the final right parenthesis of the FORMAT statement), a new record is read into the input buffer.

With internal files, when the format specification indicates a new record is to be processed (by a slash or final right parenthesis), the next element of the array is used as the input buffer.

```
TODAY=  79/08/17. CLOCK=  12.12.21.
THE ELAPSED CPU TIME IS      3.1010      SECONDS.

SQRT(2.4   )/PI =      .49312
SIN(2.4)/PI =      .21501
THE INITIAL VALUE OF THE RANF SEED IS  1717127432147741З155, OR
         .17099839404402З      IN G30.15 FORMAT.
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
DIVIDE ERROR. WOW PRINTS AS:          R
```

Figure 12-26. Program LIBS Output

```
      PROGRAM ADD
      DIMENSION IN(79)
      CHARACTER CARD*79, FM(3)*6
      DATA FM/'(7911)', '(3912)', '(2613)'/
C
      OPEN (5, FILE='INPUT')
      OPEN (6, FILE='OUTPUT')
10    READ (5, '(I1,A)', END=100) KEY, CARD
      N= MAX(1, MIN(KEY, 3))
      LIM= 79/N
C
      READ (CARD, FM(N)) (IN(I),I=1,LIM)
C
      ITOT=0
      DO 40 I=1,LIM
40    ITOT= ITOT + IN(I)
      WRITE (6,12) ITOT, LIM, N, CARD, (IN(I),I=1,LIM)
12    FORMAT (/16,20H IS THE TOTAL OF THE ,I3,20H NUMBERS ON THE CARD/
     1 I2,A79,/16H THE NUMBERS ARE/(20I4))
      GO TO 10
100   STOP
      END
```

Figure 12-27. Program ADD

Write

A formatted WRITE statement for external files causes the output buffer to be cleared. Data in the WRITE statement list is converted into a character string according to the format specified in the format specification, and placed in the output buffer. When the format specification indicates the end of a record with either a slash or the final right parenthesis, the character string is passed from the output buffer to the output system; the output buffer area is reset, and the next string of characters is placed in the buffer.

The WRITE statement for internal files is processed by compiler routines in the same way as for external files, but with the internal file specified within the WRITE statement used as the output buffer. The number of words per record in the array is determined by the length of an element.

In the sample program, the format of data on input is specified in column 1 of each input card. If column 1 contains a one or zero or blank, each of the remaining columns contains a data item. If column 1 is a two, each pair of the remaining columns is a data item. If column 1 contains a number equal to or greater than 3, each triplet of the remaining columns is a data item. Based on the information in column 1, the correct format specification is selected. The program then totals and prints out the items in each input record.

CARD is a character variable 79 characters long, which is to receive the characters in columns 2 through 80 of the input record. IN is dimensioned 79 to receive the converted input items. FM is a character array which contains three elements, each six characters long. The DATA statement (line 4) loads a format specification into each element of FM.

The READ statement in line 8 reads the first column of an input record into KEY under I format and the remaining 79 characters into CARD under A format. When an end-of-file is encountered, control transfers to statement 100, a STOP statement.

Line 9 ensures that the value of KEY is between 1 and 3; this value is stored in N.

Line 10 calculates the number of values to be transferred to IN.

The READ statement in line 12 transmits the characters in CARD to IN, converting them to integers according to the format specification stored in FM; N selects the array element containing the correct format specification.

Lines 14 through 20 sum the values in IN, print the input and output values, and branch back to process the next input record.

Sample input and output records for program ADD are shown in figure 12-28.

Input:

```
213225547669887755332103322456668779655412333221123654789654123655478965412360028
302144566998774566322144556666552336552221444556633255666998856665547788548870029
555666632236665523322144556669988776552221444556112233033244566699887745588960300
102345666889998877899655544445566665533222111233023333669985555222114444777885031
```

Output:

```
 1900 IS THE TOTAL OF THE 39 NUMBERS ON THE CARD
213225547669887755332103322456668779655412333221123654789654123655478965412360028
THE NUMBERS ARE
 13  22  55  47  66  98  87  75  53  32  10  33  22  45  66  68  77  96  55  41
 23  33  22  11  23  65  47  89  65  41  23  65  54  78  96  54  12  36   2

14380 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
302144566998774566322144556666552336552221444556633255666998856665547788548870029
THE NUMBERS ARE
 21 445 669 987 745 663 221 445 566 665 523 365 522 214 445 566 332 556 669 988
566 655 477 885 488 702

13840 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
355666632236665523322144556669988776552221444556112233033244566699887745588960300
THE NUMBERS ARE
556 666 322 366 655 233 221 445 566 699 887 765 522 214 445 561 122 330 332 445
666 998 877 455 889 603

  370 IS THE TOTAL OF THE 79 NUMBERS ON THE CARD
102345666889998877899655544445566665533222111233023333669985555222114444777885031
THE NUMBERS ARE
  0   2   3   4   5   6   6   6   8   8   8   9   9   8   8   7   7   8   9   9
  6   5   5   5   4   4   4   4   5   5   6   6   6   5   5   3   3   2   2   2
  1   1   1   2   3   3   0   2   3   3   3   3   6   6   9   9   8   5   5   5
  5   2   2   2   1   1   4   4   4   4   7   7   7   8   8   5   0   3   1
```

Figure 12-28. Program ADD Input and Output

PROGRAM PASCAL

Program PASCAL, shown in figure 12-29, produces a table of binary coefficients (Pascal's triangle). The following features are illustrated:

Nested DO loops

Implied DO loop

The DO loop in lines 6 and 7 initializes the integer array LROW to 1. The PRINT statement in line 8 prints a heading and the the first two rows of the triangle.

The nested DO loops (lines 11 through 15) calculate the remaining elements of the triangle. These statements illustrate the technique of going backward through an array by using a negative incrementation parameter.

Each pass through the inner DO loop generates one row of the triangle. The row elements are written in line 14 using an implied DO loop.

Output from program PASCAL is shown in figure 12-30.

```
      PROGRAM PASCAL
C
C THIS PROGRAM PRODUCES A PASCAL TRIANGLE WITH 15 ROWS
C
      INTEGER LROW(15)
      DO 10 I=1,15
10    LROW(1)= 1
      PRINT '("1 PASCAL TRIANGLE "//1X, 15,/1X, 215)', LROW(15),
     * LROW(14), LROW(15)
C
      DO 50 J = 14, 2, -1
      DO 40 K=J,14
40    LROW(K)= LROW(K) + LROW(K+1)
      PRINT '(1X, 1515)', (LROW(M), M=J-1,15)
50    CONTINUE
C
      STOP
      END
```

Figure 12-29. Program PASCAL

```
PASCAL TRIANGLE

     1
     1     1
     1     2     1
     1     3     3     1
     1     4     6     4     1
     1     5    10    10     5     1
     1     6    15    20    15     6     1
     1     7    21    35    35    21     7     1
     1     8    28    56    70    56    28     8     1
     1     9    36    84   126   126    84    36     9     1
     1    10    45   120   210   252   210   120    45    10     1
     1    11    55   165   330   462   462   330   165    55    11     1
     1    12    66   220   495   792   924   792   495   220    66    12     1
     1    13    78   286   715  1287  1716  1716  1287   715   286    78    13     1
     1    14    91   364  1001  2002  3003  3432  3003  2002  1001   364    91    14     1
```

Figure 12-30. Program PASCAL Output

PROGRAM PIE

Program PIE, shown in figure 12-31, calculates an approximation of the value of π. This program illustrates the use of the intrinsic function RANF.

The random number generator, RANF, is called twice during each iteration of the DO loop, and the values obtained are stored in the variables X and Y.

The DATA statement (line 2) initializes the variable circle with the value 0.0.

Each time RANF is called, a random number, uniformly distributed over the range 0 through 1, is returned. A random number is stored in X and in Y.

The IF statement and the arithmetic expression 4.0*CIRCLE/1000.0. calculate an approximation of the value of π. The value of π is calculated using Monte Carlo techniques. The IF statement counts those points whose distance from the point (0., 0.) is less than or equal to one. The ratio of the number of points within the quarter circle to the total number of points approximates 1/4 of π. The value PI is printed by the list directed output statement PRINT*, 'PI=', PI.

PROGRAM X

Program X, shown in figures 12-32 and 12-33, references a function EXTRAC which squares the number passed as an argument. This program illustrates the following features:

Referencing user-defined functions

Function type

Program X illustrates that a function type must agree with the type associated with the function name in the calling program.

In the example shown in figure 12-32, the first letter of the function name EXTRAC is E and the function is therefore implicitly typed real. EXTRAC is referenced, and the value 7 is passed to the function as an argument. However, the function subprogram is explicitly defined integer, INTEGER FUNCTION EXTRAC(K), and the conflicting types produce erroneous results.

```
      PROGRAM PIE
      DATA CIRCLE /0.0/
C
      DO 1 I = 1,1000
      X= RANF()
      Y= RANF()
      IF (X*X + Y*Y .LE. 1.0) CIRCLE= CIRCLE + 1.0
1     CONTINUE
C
      PI= 4.0*CIRCLE/1000.0
      PRINT*, ' PI = ', PI
C
      STOP
      END
```

Output:

```
PI =  3.148
```

Figure 12-31. Program PIE and Output

The argument 7 is type integer which agrees with the type of the dummy argument K in the subprogram. The result 49 is correctly computed. However, when this value is returned to the calling program, the integer value 9 is returned to the real name EXTRAC; and an integer value in a real variable produces an erroneous result. (Refer to program EQUIV.)

This problem arises because the programmer and the compiler regard a program from different viewpoints. The programmer often considers a complete program to be one unit, whereas the compiler treats each program unit separately. To the programmer, the statement:

```
INTEGER FUNCTION EXTRAC(K)
```

defines the function EXTRAC integer. The compiler, however, compiles integer function EXTRAC and the main program separately. In the subprogram, EXTRAC is declared integer; in the main program it is declared real. Information (in this instance the type of the function) which the main program needs regarding a subprogram, must be supplied in the main program.

There is no way for the compiler to determine if the type of a program unit agrees with the type of the name in the calling program; therefore, no diagnostic help can be given for errors of this kind.

In figure 12-33, EXTRAC is declared integer in the calling program, and the correct result is obtained.

```
      PROGRAM X
C  IF EXTRAC IS DECLARED TYPE INTEGER THE RESULT IS 49, OTHERWISE IT IS
C  ZERO
C
      K= EXTRAC(7)
      PRINT '(''1K = '', I5)', K
      STOP
      END
C
```

Function EXTRAC:

```
      INTEGER FUNCTION EXTRAC (K)
      EXTRAC= K*K
      RETURN
      END
```

Output:

```
K =     0
```

Figure 12-32. Program X, Function EXTRAC, Output: INTEGER Declaration Omitted From Main Program

```
      PROGRAM X
C  IF EXTRAC IS DECLARED TYPE INTEGER THE RESULT IS 49, OTHERWISE IT IS
C  ZERO
C
      INTEGER EXTRAC
      K= EXTRAC(7)
      PRINT '(''1K = '', I5)', K
      STOP
      END
C
```

Function EXTRAC:

```
      INTEGER FUNCTION EXTRAC (K)
      EXTRAC= K*K
      RETURN
      END
```

Output:

```
K =    49
```

Figure 12-33. Program X, Function EXTRAC, Output: INTEGER Declaration Included in Main Program

PROGRAM ADIM

Program ADIM, shown in figure 12-34, illustrates the use of adjustable dimensions to allow a subroutine to operate on arrays of various sizes. The following features are included in this example:

Passing an array to a subroutine as a parameter

Specifying an array name, with no dimension information, in an argument list

Specifying an array with a negative lower subscript bound

Two arrays, X and Z, are dimensioned and placed in common. Z is dimensioned (-2:3). This means that Z has six elements; the lower subscript bound is -2 and the upper subscript bound is 3. The elements are: Z(-2), Z(-1), Z(0), Z(1), Z(2), Z(3).

The array Y is dimensioned (6) and is explicitly typed real. It is not in common.

In subroutine IOTA, the adjustable dimension for array A is indicated by M. Whenever the main program calls IOTA, it can provide the name and the dimensions of the array; since A and M are dummy arguments, IOTA can be called repeatedly with different dimensions replacing M at each call. IOTA contains a DO loop which stores consecutive integers into the array A.

The main program calls subroutine IOTA three times. In the first call, the first argument is array X and the second argument is the number of elements in the array, 12. Consecutive integers are stored into the 12 elements of X.

In the second call to IOTA, the arguments (Y,6) are passed. Consecutive integers are stored into the six elements of Y.

In the third call to IOTA, the arguments (Z,6) are passed. The subscript bounds specified in the subroutine need not be the same as the ones specified in the calling program. Although Z is dimensioned (-2:3) in the main program, it can be dimensioned (6) in IOTA.

The PRINT statements output the arrays X, Y, and Z. The second PRINT statement illustrates the use of a negative DO index to output the array Z. The output is shown in figure 12-35.

```
      PROGRAM ADIM
      COMMON X(4,3), Z(-2:3)
      REAL Y(6)
C
      CALL IOTA (X,12)
      CALL IOTA (Y,6)
      CALL IOTA (Z,6)
C
      PRINT 100, X, Y, Z
100   FORMAT ('1ARRAY X = ',12F6.0/' ARRAY Y = ', 6F6.0,
     * /' ARRAY Z = ',6F6.0)
C
      DO 8 I = -2,3
8     Z(I)= I
      PRINT 110, Z
110   FORMAT (' ARRAY Z = ',6F6.0)
C
      STOP
      END
C
      SUBROUTINE IOTA (A,M)
C
C  IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
C  STARTING AT 1
C
      DIMENSION A(M)
      DO 1 I = 1,M
1     A(I)= I
      RETURN
      END
```

Figure 12-34. Program ADIM and Subroutine IOTA

```
ARRAY X =     1.    2.    3.    4.    5.    6.    7.    8.    9.   10.   11.   12.
ARRAY Y =     1.    2.    3.    4.    5.    6.
ARRAY Z =     1.    2.    3.    4.    5.    6.
ARRAY Z =    -2.   -1.    0.    1.    2.    3.
```

Figure 12-35. Program ADIM Output

PROGRAM ADIM2

ADIM2, shown in figure 12-36, is an extension of program ADIM. Subroutine IOTA is used; in addition, another subroutine and two functions are used. The following features are illustrated:

Parameter statement

Negative array subscripts

Negative DO parameters

Use of an expression for an array dimension

Multiple entry points

Adjustable dimensions

EXTERNAL statement

Passing values through COMMON

Use of intrinsic functions ABS and REAL

Calling functions through several levels

Passing a subprogram name as an argument

Program ADIM2 illustrates the method of a main program calling subprograms and subprograms calling each other. Since the program is necessarily complex, each subprogram is described separately followed by a description of the main program.

Subroutine SET

Subroutine SET places the value V into every element of the array A. The dimension of A is specified by M.

Subroutine SET has an alternate entry point INC. When SET is entered at ENTRY INC, the value V is added to each element of the array A. The dimension of A is specified by M.

The first DO loop in subroutine SET clears the array to zero.

Subroutine IOTA

Subroutine IOTA is as described for program ADIM except that the input array A is given negative upper and lower subscript bounds. The DO loop uses negative control variables and places consecutive negative integers in A.

Subroutine PVAL

Function PVAL references a function specified by the calling program to return a value to the calling program. This value is forced to be positive by the intrinsic function ABS.

The main program first calls PVAL with the statement AA=PVAL(M,AVG), passing the integer M (assigned the value 12 in the PARAMETER statement) and the function AVG as parameters. The type of the argument in the main program (INTEGER M) agrees with the corresponding dummy argument (ISIZE) in the subprogram.

The value of PVAL is computed in line 7. This value will be returned to the main program through the function name PVAL. Two functions are referenced by this statement; the intrinisic function ABS and the user-written function AVG. The actual arguments M and AVG replace ISIZE and WAY. The second time PVAL is called, the actual arguments M and MULT replace ISIZE and WAY.

Function AVG

This function computes the average of the first J elements of common. J is a value passed by the main program through the function PVAL.

This function subprogram is an example of a main program and a subprogram sharing values in common. The main program and function AVG declare common to be a total of 12 words. Values placed in common by the main program are available to the function subprogram.

The number of values to be averaged is passed to function PVAL by the statement AA=PVAL(12,AVG) and function PVAL passes this number (in ISIZE) to function AVG: PVAL=ABS(WAY(ISIZE)).

AVG uses a PARAMETER statement to assign symbolic names to the constants 4 and 3. These constants are then used in an expression that calculates the dimension for A. The expression itself is used as the dimension for A. AVG declares a total of 12 locations for common.

Lines 4 through 6 sum the 12 elements and divide by the number of elements to calculate the average. The intrinsic function REAL is used to convert the integer 12 to a real number to avoid mixed mode arithmetic, although in this case mixed mode is permissible and produces the same result.

The average is returned to the statement PVAL=ABS(WAY(ISIZE)) in function PVAL.

Function MULT

MULT multiplies the first and twelfth words in COMMON and subtracts the product from the average (computed by the function AVG) of the first J/2 words in common.

The declaration COMMON ARRAY (-1:10) assigns 12 elements to ARRAY and places it in common. The 12 elements are referenced by a subscript in the range -1 through 10. Line 8 multiplies the first element (ARRAY(-1)) by the twelfth element (ARRAY(10)) and subtracts the average (computed by function AVG) of the first J/2 elements in common.

Main Program: ADIM2

The main program calls the subroutines and functions described.

The array Y has six elements, with subscript bounds of (-2:3). MULT and AVG appear in an EXTERNAL statement so that they can be passed to subprograms as arguments.

Lines 12 through 16 call the user-written subprograms SET, IOTA, and PVAL; CALL INC calls subroutine SET through the alternate entry point INC. The calls to PVAL pass a symbolic constant and a function name. Results are returned to AA and AM, respectively.

The namelist PRINT statement outputs the values calculated by the subprograms. The output is shown in figure 12-37.

```
      PROGRAM ADIM2
C
C  THIS PROGRAM USES ADJUSTABLE DIMENSIONS, NEGATIVE ARRAY BOUNDS,
C  AND MANY SUBPROGRAM CONCEPTS
C
      PARAMETER (I=4, J=3, K=-2, M=12, N=6)
      COMMON X(I,J)
      REAL Y(K:J)
      EXTERNAL MULT, AVG
      NAMELIST /V/ X, Y, AA, AM
C
      CALL SET (Y, M, 0.)
      CALL IOTA(X, M)
      CALL INC (X, M, -5.0)
      AA= PVAL(M, AVG)
      AM= PVAL(M, MULT)
      PRINT V
      STOP
      END



C
      SUBROUTINE SET (A, M, V)
C
C  SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
C
      DIMENSION A(*)
      DO 1 I = 1,M
1     A(I)= 0.0
C
      ENTRY INC(A,M,V)
C
C  INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
C
      DO 2 I = 1,M
2     A(I)= A(I) + V
      RETURN
      END
```

Figure 12-36. Program ADIM2

```
C
      SUBROUTINE IOTA (A,M)
C
C  IOTA PUTS CONSECUTIVE NEGATIVE INTEGERS STARTING AT -1 INTO EVERY
C ELEMENT OF THE ARRAY A
C
      DIMENSION A(-M:-1)
      DO 1 I = -1, -M, -1
1     A(I)= I
      RETURN
      END



C
      FUNCTION PVAL (ISIZE, WAY)
C
C  PVAL COMPUTES THE ABSOLUTE VALUE OF THE REAL VALUE OF A FUNCTION
C  PASSED TO PVAL.  ISIZE IS AN INTEGER WHICH PVAL PASSES TO THE
C  FUNCTION
C
      PVAL= ABS(WAY(ISIZE))
      RETURN
      END



C
      FUNCTION AVG(J)
C
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON
C
      PARAMETER (M=4, N=3)
      COMMON A(M*N)
      AVG= 0.
      DO 1 I = 1,J
1     AVG= AVG + A(I)
      AVG= AVG/REAL(J)
      RETURN
      END


C
      REAL FUNCTION MULT(J)
C
C  MULT MULTIPLIES THE FIRST AND TWELFTH ELEMENTS OF COMMON AND
C  SUBTRACTS FROM THIS THE AVERAGE (COMPUTED BY THE FUNCTION AVG)
C  OF THE FIRST J/2 WORDS IN COMMON
C
      COMMON ARRAY(-1:10)
      MULT= ARRAY(10)*ARRAY(-1) - AVG(J/2)
      RETURN
      END
```

Figure 12-36. Program ADIM2 (Sheet 2 of 2)

```
$V
X       = -.17E+02, -.16E+02, -.15E+02, -.14E+02, -.13E+02,
          -.12E+02, -.11E+02, -.1E+02, -.9E+01, -.8E+01,
          -.7E+01, -.6E+01,

Y       = 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,

AA      = .115E+02,

AM      = .1165E+03,

$END
```

Figure 12-37. Program ADIM2 Output

PROGRAM CIRCLE

Program CIRCLE, shown in figure 12-38, finds the area of a circle which circumscribes a rectangle with short sides of length 3 and long sides of length 4. This example illustrates the use of FUNCTION subprograms and of statement functions. The program contains an error.

Program CIRCLE:

```
      PROGRAM CIRCLE
      A= 4.0
      B= 3.0
      AREA= 3.1416/4.0 * DIM(A,B)**2
      PRINT 1, AREA
1     FORMAT (' AREA = ', G20.10)
      STOP
      END
```

Function DIM:

```
C
      FUNCTION DIM(X, Y)
      DIM= SQRT(X*X + Y*Y)
      RETURN
      END
```

Output:

```
 AREA =       .7854000000
```

Figure 12-38. Program CIRCLE, Function DIM, Output

Figure 12-39 shows a rectangle and circumscribed circle. The area of a circle is given by R^2, which is approximated by the FORTRAN expression:

```
3.1416/4.0*D**2
```

where R is the radius and D is the diameter of the circle.

The user-written function DIM computes the diameter of the rectangle given the lengths of the sides using the relation:

```
DIM=SQRT(X*X + Y*Y)
```

The result shown in figure 12-38 is incorrect. The area of a circle circumscribing a rectangle with sides 3 and 4 is clearly greater than .785.

The error occurred because the function DIM has the same name as an intrinsic function. If the name of an intrinsic function is used for a user-written function, the user-written function is ignored.

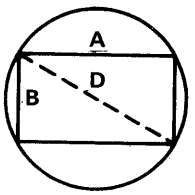


Figure 12-39. Rectangle and Circumscribed Circle

There are several ways of correcting this error:

Change the function name so that it is not the same as an intrinsic function name.

Declare DIM external; in this case, the user-written external function will be used.

Write the function DIM as a statement function; the function name can be the same as an intrinsic function name, and the user-written function is used. This is the most efficient method. Since FORTRAN compiles statement functions in-line, the program executes much faster because no function call is used. This solution is limited to functions of a single statement.

A corrected version of the program, in which DIM is written as a statement function, is shown in figure 12-40.

Program CIRCLE:

```
      PROGRAM CIRCLE
      DIM(X,Y)= SQRT(X*X + Y*Y)
      A= 4.0
      B= 3.0
      AREA= 3.1416/4.0*DIM(A,B)**2
      PRINT 1, AREA
1     FORMAT ('1AREA IS ',G20.10)
      STOP
      END
```

Output:

```
 AREA IS       19.63500000
```

Figure 12-40. Program Circle With Correction and Output

PROGRAM BOOL

Program BOOL, shown in figures 12-41 and 12-42, illustrates some problems that can occur when Boolean constants are used in expressions. The program in figure 12-41 contains no type declaration for the variables A, B, C, D, and E. The program in figure 12-42 declares these variables type Boolean.

Boolean constants include octal, hexadecimal, and Hollerith constants. When Boolean constants are used in expressions with operands of another type, no mode conversion occurs, and the result has the type of the other operand. Boolean operands used in arithmetic expressions are treated as type integer. For example, referring to figure 12-41, the statement:

```
B = O"10" + O"10"
```

is evaluated using integer arithmetic. Furthermore, for subsequent operations, the result of integer arithmetic is treated as true integer. Thus, in the above example, the expression on the right is evaluated using integer arithmetic; and the integer result is converted to real before the value is stored in B. Comparing the values produced for A and B illustrates this effect.

With floating-point arithmetic, whenever the left 12 bits of the computer word are all zeros or all ones, the value of that number is zero. (See Program EQUIV.) This explains why the output value of A in figure 12-41 is zero.

Program BOOL:

```
      PROGRAM BOOL
      LOGICAL F
      NAMELIST /OUT/A,B,C,D,E,F
      A= O"20"
      B= O"10" + O"10"
      C= B + O"10"
      I= 5
      D= I + O"10"
      E= B + I + O"10"
      F= A .EQ. O"77"
      PRINT OUT
      STOP
      END
```

Output:

```
$OUT
A       = 0.0,
B       = .16E+02,
C       = .16E+02,
D       = .13E+02,
E       = .21E+02,
F       = T,
$END
```

Figure 12-41. Program BOOL and Output

The remaining expressions are evaluated as follows:

```
C=B+O"10"
```

Floating-point arithmetic is used to evaluate the expression; the octal constant O"10" is used without type conversion, making its value zero. Note in the output from BOOL the values of B and C are equal.

```
D=I+O"10"
```

No problem arises in the preceding expression as it is evaluated with integer arithmetic; then the result is converted to real and stored in D.

```
E=B+I+O"10"
```

The compiler, in scanning the preceding expression left to right, encounters the real variable B and uses real arithmetic to evaluate the expression. Again, the octal constant O"10" has the real value of zero.

If the expression were written as:

```
E=O"10"+I+B  or  E=I+O"10"+B
```

the first two terms would be added using integer arithmetic; then that result would be converted to real and added to B. In this case, the octal constant O"10" would effectively have the value eight.

This is similar to the mode conversion which occurs in:

```
X=Y*3/5  or  Z=3/5*Y
```

These expressions would give different values for X and Z. More information on the evaluation of mixed mode expressions is presented in section 3.

```
F=A .EQ. O"77"
```

Real arithmetic is used to compare the value because A is a type real name. The value in A and the constant O"77" both have all zeros in the leftmost 12 bits; both have value zero for real arithmetic; therefore, the value assigned to F is .TRUE.

To avoid the confusion illustrated in this example, simply use a type Boolean declaration for variable names whose values come from octal, hexadecimal, or Hollerith constants. Figure 12-42 shows the same program with the names A, B, C, D, and E all as type Boolean.

All these examples use octal constants; however, the same problem occurs with Hollerith, especially when it is right-justified, and with hexadecimal. The following program segment illustrates the point:

```
      .
      .
      .
      REAL ANS
      .
      .
      .
      READ 2, ANS
    2 FORMAT(R3)
      IF(ANS .EQ. R"NO ") PRINT 3
    3 FORMAT ('ΔNEGATIVE RESPONSE')
      .
      .
      .
```

Program BOOL:

```
      PROGRAM BOOL
      LOGICAL F
      BOOLEAN A,B,C,D,E
      NAMELIST /OUT/A,B,C,D,E,F
      A= O"20"
      B= O"10" + O"10"
      C= B + O"10"
      I= 5
      D= I + O"10"
      E= B + I + O"10"
      F= A .EQ. O"77"
      PRINT OUT
      STOP
      END
```

Output:

```
$OUT
A       = O"20",
B       = O"20",
C       = O"30",
D       = O"15",
E       = O"35",
F       = F,
$END
```

Figure 12-42. Program BOOL With Correction and Output

PROGRAM EASY IO

Program EASY IO, shown in figure 12-43, illustrates the use of list directed input/output.

List directed input/output eliminates the need for fixed data fields. It is especially useful for input since the user need not be concerned with punching data in specific columns. List directed input does not require the user to name each item as does NAMELIST input.

Used in combination, list directed input and NAMELIST output simplify program design. Such a program is easy to write, even for persons just learning the language; knowledge of the format specifications is not required. This feature is particularly useful when FORTRAN programs are being run from a remote terminal.

Program EASY IO calculates the area and radius of a circle inscribed in a triangle, given the lengths of the sides of the triangle. A list directed READ statement is used for input, and NAMELIST is used for output. Figure 12-44 shows some sample input and output.

The user can enter the three input values in whatever way is convenient, such as: one item per line (or card), one item per line with each item followed by a comma, all items on a single line with spaces separating each item, all items on a line with a comma and several spaces separating each item, or any combination of the foregoing. Furthermore, even though all input items are real, the decimal point is not required when the input value is a whole number.

```
      PROGRAM EASY IO
C
C  GIVEN THE SIDES OF A TRIANGLE, COMPUTE THE AREA AND RADIUS OF THE
C  INSCRIBED CIRCLE
C
      REAL SIDES(3)
      EQUIVALENCE (SIDES(1),A), (SIDES(2),B), (SIDES(3),C)
      NAMELIST /OUT/ SIDES, AREA, RADIUS
3     READ (*, *, END=50) SIDES
      S= (A + B + C)/2.0
      AREA= SQRT(S*(S-A) * (S-B) * (S-C))
      RADIUS= AREA/S
      WRITE (*, OUT)
      GO TO 3
50    STOP
      END
```

Figure 12-43. Program EASYIO

Input:

```
3 4 5
6,7,8
3*1
4
5
6
```

Output:

```
$OUT
SIDES    = .3E+01, .4E+01, .5E+01,
AREA     = .6E+01,
RADIUS   = .1E+01,
$END

$OUT
SIDES    = .6E+01, .7E+01, .8E+01,
AREA     = .203331625675 89E+02,
RADIUS   = .193649167310 37E+01,
$END

$OUT
SIDES    = .1E+01, .1E+01, .1E+01,
AREA     = .433012701892 22E+00,
RADIUS   = .288675134594 81E+00,
$END

$OUT
SIDES    = .4E+01, .5E+01, .6E+01,
AREA     = .992156741649 22E+01,
RADIUS   = .132287565553 23E+01,
$END
```

Figure 12-44. Sample Input and Output for Program EASYIO

PROGRAM BLOCK

Program BLOCK, shown in figure 12-45, illustrates block IF structures.

Block IF structures allow the user to specify alternate paths of execution, based on the outcome of IF tests. Block IF structures eliminate the need for branching when IF tests are performed. This feature can make programs simpler and more readable.

Program BLOCK reads an integer into the variable K, and two sets of real numbers into the arrays A and B. K is tested and the following action is taken:

K=1	Calculate C(I)=A(I)**2 + B(I)**2.
K=2	Calculate C(I)=A(I)*B(I).
All other values of K	Set array C to zero.

These tests could be performed by conventional methods, using logical IF and GO TO statements. However, with block IF structures the program is much clearer.

The program includes a block IF statement (line 7), and ELSE IF statement (line 11), and an ELSE statement (line 15). These statements provide for three alternate paths of execution. After the appropriate block has been executed, control transfers to the WRITE statement following END IF. The program then branches back to process the next input record.

Sample input and output are shown in figure 12-46.

```
      PROGRAM BLOCK
      PARAMETER (M=5)
      DIMENSION A(M), B(M), C(M)
      NAMELIST /OUT/ K, A, B, C
C
2     READ (*, *, END=100) K, A, B
      IF (K .EQ. 1) THEN
      DO 5 I = 1,M
5     C(I)= A(I)**2 + B(I)**2
C
      ELSE IF (K .EQ. 2) THEN
         DO 10 I = 1,M
10       C(I)= A(I)*B(I)
C
      ELSE
         DO 15 I = 1,M
15       C(I)= 0.0
C
      END IF
         WRITE (*, OUT)
         GO TO 2
100   STOP
      END
```

Figure 12-45. Program BLOCK

Input:

```
5 9.0 9.0 8.0 8.0 7.0 5.0 3.0 3.0 2.0 6.0
1 1.0 0.0 0.0 7.0 7.0 4.0 0.0 0.0 0.0 0.0
4 4.0 4.0 4.0 7.0 8.0 5.0 0.0 0.0 3.0 2.0
3 3.0 3.0 2.0 2.0 1.0 6.0 8.0 0.0 1.0 1.0
```

Output:

```
$OUT
K       = 5,
A       = .9E+01, .9E+01, .8E+01, .8E+01, .7E+01,
B       = .5E+01, .3E+01, .3E+01, .2E+01, .6E+01,
C       = 0.0, 0.0, 0.0, 0.0, 0.0,
$END
$OUT
K       = 1,
A       = .1E+01, 0.0, 0.0, .7E+01, .7E+01,
B       = .4E+01, 0.0, 0.0, 0.0, 0.0,
C       = .17E+02, 0.0, 0.0, .49E+02, .49E+02,
$END
$OUT
K       = 4,
A       = .4E+01, .4E+01, .4E+01, .7E+01, .8E+01,
B       = .5E+01, 0.0, 0.0, .3E+01, .2E+01,
C       = 0.0, 0.0, 0.0, 0.0, 0.0,
$END
$OUT
K       = 3,
A       = .3E+01, .3E+01, .2E+01, .2E+01, .1E+01,
B       = .6E+01, .8E+01, 0.0, .1E+01, .1E+01,
C       = 0.0, 0.0, 0.0, 0.0, 0.0,
$END
```

Figure 12-46. Sample Input and Output for Program BLOCK

PROGRAMS ONE AND TWO

Programs ONE and TWO, shown in figure 12-47, illustrate internal file usage.

Program ONE writes a single record to an internal file. The array A and the variables B and C are declared type character of length 10. The character variable ALPHA, to be used as the internal file, has length 40. The DATA statement loads character data into A, B, and C.

The WRITE statement defines ALPHA to be an internal file and writes the values of A, B, and C to the file according to the format specification (2A4, A5, A6). The following formatting is performed:

Characters ABCD from A(1) are transmitted to positions 1 through 4 of ALPHA.

Characters KLMN from A(2) are transmitted to positions 5 through 8 of ALPHA.

Characters UVWXY from B are transmitted to positions 9 through 13 of ALPHA.

Characters Z12345 from C are transmitted to positions 14 through 19 of ALPHA.

Positions 20 through 40 of ALPHA are blank filled.

Program TWO is identical to program ONE except that ALPHA is dimensioned 2 and the format specification is changed to cause two records to be written to ALPHA. The characters in A(1) and A(2) are transmitted to ALPHA(1) as before. The slash, however, causes subsequent data to be transmitted to ALPHA(2). Unused portions of both records are blank filled.

Example 1:

```
      PROGRAM ONE
      CHARACTER A(2)*10,B*10,C*10,ALPHA*40
      DATA A,B,C /'ABCDEFGHIJ','KLMNOPQRST','UVWXY','Z123456'/
      WRITE (ALPHA,'(2A4,A5,A6)') A,B,C
      PRINT 2,ALPHA
2     FORMAT ('1CONTENTS OF ALPHA = ', /1X, A40)
      STOP
      END
```

Output:

```
 CONTENTS OF ALPHA =
 ABCDKLMNUVWXYZ12345Δ - - - - - - - - - - Δ
                SECONDS EXECUTION TIME.
```

A single record is written to the internal file ALPHA.

Example 2:

```
      PROGRAM TWO
      CHARACTER A(2)*10,B*10,C*10,ALPHA(2)*40
      DATA A,B,C /'ABCDEFGHIJ','KLMNOPQRST','UVWXY','Z123456'/
      WRITE  (ALPHA,'(2A4/A5,A6)') A,B,C
      PRINT 2,ALPHA
2     FORMAT ('1CONTENTS OF ALPHA = ', /1X, 2A40)
      STOP
      END
```

Output:

```
 CONTENTS OF ALPHA =
 ABCDKLMNΔ - - - - - - - - - - - - - - - - - Δ | UVWXYZ12345Δ - -
```

record 1 **record 2**

Two records are written to the internal file ALPHA.

Figure 12-47. Programs ONE and TWO

PROGRAM PMD2

Program PMD2, shown in figure 12-48, illustrates the use of the Post Mortem Dump facility. In this example, the dump is triggered by a program abort.

Program PMD2 consists of a main program and a subroutine. The main program contains an error: in the CALL statement, the subroutine name SETCOM is misspelled as SETCM. This error causes the program to abort when the statement CALL SETCM is executed.

Subroutine SETCOM tests the logical variable L. If L contains the value .TRUE., data is read from unit 1 into the array B. If L contains the value .FALSE., B is set to zero.

Note that the program contains no calls to Post Mortem Dump routines. In this case, if the program aborts and DB=PMD was selected, a dump occurs automatically.

The Post Mortem Dump output for program PMD2 is shown in figure 12-49. The dump includes an error analysis, a description of current file status, and an analysis of variables in the main program (in which the error occurred).

```
      PROGRAM PMD2
C
C  THIS PROGRAM CONTAINS AN ERROR WHICH ACTIVATES POST MORTEM DUMP
C  IF DB=PMD IS SELECTED
C
      CHARACTER*10 FILE, IFG
      LOGICAL LVAR
      COMMON /CBLOCK/ ARR(3,3)
C
      OPEN (UNIT=6,FILE='OUTPUT')
      LVAR = .TRUE.
      CALL SETCM (LVAR, IFG)
      WRITE (6,*) IFG, ARR
      STOP
      END
C
      SUBROUTINE SETCOM (L, IFG)
      LOGICAL L
      CHARACTER*10 IFG
      COMMON /CBLOCK/ B(3,3)
C
      IF (L) THEN
      IFG = 'FIRST'
      READ (1,END=999) ((B(I,J),I=1,3),J=1,3)
      ELSE
      IFG = 'SECOND'
      DO 10 I=1,3
      DO 10 J=1,3
10    B(I,J) = 0.0
      ENDIF
999   RETURN
      END
```

Figure 12-48. Program PMD2

```
FTN POST MORTEM DUMP                                    ERROR  REPORT                    79/08/20. 12.57.28.

*** YOUR JOB HAS THE FOLLOWING NON-FATAL LOAD ERROR(S):
                    UNSATISFIED EXTERNAL REF -- SETCM

/// EXECUTION WAS TERMINATED BECAUSE YOUR PROGRAM CALLED A MISSING ROUTINE AT LINE NUMBER   12 OF   PROGRAM  PMD2

... ARRAYS WILL BE PRINTED BY  DEFAULT  PARAMETERS (   20,    2,    1,    1,    1,    1,    1)

... YOUR PROGRAM REQUIRED  26300B WORDS TO LOAD, 10315B WORDS TO RUN

... FILE STATUS AT TIME OF TERMINATION

        FILE NAME  FORTRAN NAMES  LAST OP       STATUS       FILE TYPE  BLOCKING TYPE  REC TYPE  RECORD COUNT
        -OUTPUT      TAPE6         OPENED                       SQ           C             Z           0

... VARIABLES IN   PROGRAM  PMD2
        NAME        TYPE       RELOCATION      CURRENT VALUE                          COMMENTS  NAME

        ARR         REAL         /CBLOCK/      ARRAY                                            ARR
    ... DIMENSIONED AS - ARR(1:3,1:3)
*** THE NEXT ITEM IS NEVER DEFINED
        FILE        CHARACT                    PMD2   :AW                                       FILE
        IFG         CHARACT                    #::E/DU5A%                                       IFG
        LVAR        LOGICAL                      .TRUE.                                         LVAR

... ARRAYS IN   PROGRAM  PMD2

    REAL    ARRAY ARR(1:3,1:3)
 (ARR(N,1))
  N=1       NOT INITIALIZED     NOT INITIALIZED     NOT INITIALIZED
 (ARR(N,2))
  N=1       NOT INITIALIZED     NOT INITIALIZED     NOT INITIALIZED

... TRACEBACK SUCCESSFULLY COMPLETED

/// END OF ERROR REPORT
```

Figure 12-49. Post Mortem Dump Output for Program PMD2

PROGRAM PMD

Program PMD, shown in figure 12-50, illustrates the use of the Post Mortem Dump. In this example, Post Mortem Dump calls are used to trigger a dump. Post Mortem Dump routines illustrated are:

PMDARRY

PMDLOAD

PMDDUMP

Program PMD consists of a main program, a subroutine, and a function subprogram. These program units perform some simple operations on values stored in an array. The call to PMDARRY in the main program specifies that only 1-dimensional arrays are to be dumped and that dumps of arrays are to be limited to the first five elements, although the arrays are dimensioned 50. The call to PMDLOAD in line 11 causes a dump of variables in the main program and in any routines that have called PMDDUMP.

Subroutine SUBT and function SQRS each contain a call to PMDDUMP. After these calls are executed, the call to PMDLOAD in the main program causes variables in SUBT and SQRS to be dumped following the variables of the main

```
      PROGRAM PMD
      DIMENSION A(50), B(50), C(50)
      DATA A/50*2.0/, B/50*4.0/
C
      CALL PMDARRY(5)
C
      DO 10 I = 1,50,2
10    A(I) = A(I) + B(I)
      CALL SUBT (A,B,C,50)
C
      CALL PMDLOAD
C
      STOP
      END
C
      SUBROUTINE SUBT (X,Y,Z,M)
      DIMENSION X(M), Y(M), Z(M)
      DO 16 I = 1,M
16    Z(I) = SQRS(X(I),Y(I))
C
      CALL PMDDUMP
C
      RETURN
      END
C
      FUNCTION SQRS(R,S)
      SQRS = R*R + S*S
C
      CALL PMDDUMP
C
      RETURN
      END
```

Figure 12-50. Program PMD

The Post Mortem Dump output is shown in figure 12-51. The dump includes an analysis of variables and traceback information for each program unit.

PROGRAM DBUG

Program DBUG, shown in figure 12-52, illustrates the use of CYBER Interactive Debug (CID) to conduct an interactive debug session (not supported on SCOPE 2). The CID commands illustrated are:

SET,BREAKPOINT

GO

PRINT

QUIT

Program DBUG stores numbers into an array A and stores a character string into a variable CHAR. The program is compiled and executed interactively in debug mode.

The terminal session for NOS/BE is shown in figure 12-53 (CID and system output are in uppercase, user input is in lowercase). The DEBUG control statement establishes debug mode. When the program is compiled in debug mode, special tables are generated for use by CID. The execution control statement LGO initiates the debug session. CID responds with:

```
CYBER INTERACTIVE DEBUG
?
```

allowing the user to enter CID commands. The SET,BREAKPOINT command sets a breakpoint that causes execution to be suspended when line 9 is reached. The GO command initiates execution of the program. The message:

```
*B #1, AT L.9
?
```

indicates that a breakpoint has suspended execution at line 9 and that CID is waiting for user input. Note that execution is suspended before the statement in line 9 is executed (the PRINT command shows that CHAR still contains the value assigned by the DATA statement).

The GO command is then entered to resume program execution. The message:

```
*T #17, END IN L.10
?
```

is a trap message indicating that the program has terminated at line 10 and that CID commands can be entered.

The QUIT command ends the debug session. Debug mode, however, remains in effect until DEBUG(OFF) is entered.

```
FTN POST MORTEM DUMP                                    ERROR  REPORT                    79/08/20. 12.59.04.

*** YOUR JOB HAS THE FOLLOWING NON-FATAL LOAD ERROR(S):
                  UNSATISFIED EXTERNAL REF -- SETCM

/// EXECUTION WAS INTERRUPTED BECAUSE YOUR PROGRAM CALLED PMDLOAD AT LINE NUMBER   11 OF   PROGRAM  PMD
... ARRAYS WILL BE PRINTED BY REQUESTED PARAMETERS (    5,    0,    0,    0,    0,    0,    0)
... YOUR PROGRAM REQUIRED  27200B WORDS TO LOAD, 11105B WORDS TO RUN

... VARIABLES IN   PROGRAM  PMD
       NAME        TYPE       RELOCATION      CURRENT VALUE                          COMMENTS  NAME

       A           REAL                       ARRAY                                            A
    ... DIMENSIONED AS - A(1:50)
       B           REAL                       ARRAY                                            B
    ... DIMENSIONED AS - B(1:50)
       C           REAL                       ARRAY                                            C
    ... DIMENSIONED AS - C(1:50)
       I           INTEGER                                51      = 1R%                       I

... ARRAYS IN   PROGRAM  PMD

   REAL    ARRAY A(1:50)
 (A(N))
  N=1      6.00000000000       2.00000000000       6.00000000000       2.00000000000       6.00000000000

   REAL    ARRAY B(1:50)
 (B(N))
  N=1      4.00000000000       4.00000000000       4.00000000000       4.00000000000       4.00000000000

   REAL    ARRAY C(1:50)
 (C(N))
  N=1      52.0000000000       20.0000000000       52.0000000000       20.0000000000       52.0000000000
-------------------------------------------------------------------------------------------------------------

FTN POST MORTEM DUMP                                    FUNCTION  SQRS                    79/08/20. 12.59.04.

... CURRENT SITUATION IN  FUNCTION  SQRS

... VARIABLES IN  FUNCTION  SQRS
       NAME        TYPE       RELOCATION      CURRENT VALUE                          COMMENTS  NAME

       R           REAL         F.P.   1      2.00000000000                                    R
       S           REAL         F.P.   2      4.00000000000                                    S

... CALLED FROM LINE NUMBER    4 OF SUBROUTINE SUBT
-------------------------------------------------------------------------------------------------------------
```

Figure 12-51. Post Mortem Dump Output For Program PMD (Sheet 1 of 2)

```
FTN POST MORTEM DUMP                          SUBROUTINE SUBT                    79/08/20. 12.59.04.

... CURRENT SITUATION IN SUBROUTINE SUBT

... VARIABLES IN SUBROUTINE SUBT
        NAME        TYPE       RELOCATION      CURRENT VALUE                        COMMENTS  NAME

        I           INTEGER                            51      = 1R%                          I
        M           INTEGER      F.P.   4              50      = 1RJ                          M
        X           REAL         F.P.   1      ARRAY                                       X
    ... DIMENSIONED AS - X(1:50)
        Y           REAL         F.P.   2      ARRAY                                       Y
    ... DIMENSIONED AS - Y(1:50)
        Z           REAL         F.P.   3      ARRAY                                       Z
    ... DIMENSIONED AS - Z(1:50)

... ARRAYS IN SUBROUTINE SUBT

    REAL    ARRAY X(1:50)
VARIABLE SPAN IN SUBSCRIPTS 1
 (X(N))
  N=1       6.00000000000       2.00000000000       6.00000000000       2.00000000000       6.00000000000

    REAL    ARRAY Y(1:50)
VARIABLE SPAN IN SUBSCRIPTS 1
                  ALL REQUESTED ELEMENTS OF THIS ARRAY WERE  4.00000000000

    REAL    ARRAY Z(1:50)
VARIABLE SPAN IN SUBSCRIPTS 1
 (Z(N))
  N=1       52.0000000000       20.0000000000       52.0000000000       20.0000000000       52.0000000000

... CALLED FROM LINE NUMBER    9 OF   PROGRAM  PMD

/// END OF ERROR REPORT
```

Figure 12-51. Post Mortem Dump Output For Program PMD (Sheet 2 of 2)

```
1          PROGRAM DBUG        74/74   OPT=0


     1                    PROGRAM DBUG
     2                    DIMENSION A(-1:4)
     3                    CHARACTER*5 CHAR
     4                    DATA CHAR /'ABCDE'/
     5               C
     6                    DO 12 I=-1,4
     7               12   A(I) = I
     8               C
     9                    CHAR = 'XYZ12'
    10                    STOP
    11                    END
```

Figure 12-52. Program DBUG

```
COMMAND- debug                                    Activate debug mode

COMMAND- ftn5,i=prog,l=list                       Compile program

      57300 CM STORAGE USED.
      0.112 CP SECONDS.
COMMAND- lgo                                      Initiate debug session

CYBER INTERACTIVE DEBUG
?set,breakpoint,l.9                               Set a breakpoint at line 9

?go                                               Begin program execution

*B #1, AT L.9                                     Breakpoint detected at line 9, execution suspended
?print*, a,char                                   Display values of A and CHAR

 -1. 0. 1. 2. 3. 4. ABCDE
?go                                               Resume execution

*T #17, END IN L.10                               Program terminates at line 10
?
     STOP
      12000B MAX FL DURING EXECUTION.
       .167 CP SECONDS EXECUTION TIME
print*, char                                      Display value of CHAR

 XYZ12
?quit                                             Terminate debug session

  DEBUG TERMINATED
COMMAND- debug(off)                               Turn off debug mode

COMMAND-
```

Figure 12-53. Debug Session

PROGRAM GOTO

Program GOTO, shown in figure 12-54, illustrates the computed GO TO feature.

Program GOTO reads records containing a single integer each and keeps a running total of the number of integers falling within the intervals 0 through 25, 26 through 50, 51 through 75, and 76 through 100. If the integer does not fall within any of these intervals an appropriate message is printed. When all records have been read, the total for each interval is printed.

In the computed GO TO statement in line 13, the control index is an expression (NUM + 24)/25. If the input value NUM is in the range 1 through 100, the value of the expression is in the range 1 through 4. The computed GO TO transfers control to the label 20, 30, 40, or 50, if the value of the expression is 1, 2, 3, or 4 respectively. The appropriate counter is then incremented. If the value of the expression is less than 1 or greater than 4, control passes to the PRINT statement following the GO TO.

Sample input and output are shown in figure 12-55.

Input:

```
56
30
110
2
25
-10
0
100
81
```

Output:

```
NUMBER  110 IS OUT OF CORRECT RANGE
NUMBER  -10 IS OUT OF CORRECT RANGE
0  - 25 :    3
26 - 50 :    1
51 - 75 :    1
76 - 100:    2
```

Figure 12-55. Sample Input and Output for Program GOTO

```
      PROGRAM GOTO
C
C  PROGRAM GOTO READS INTEGERS RANGING FROM 1 TO 100, DIVIDES THEM INTO
C  FOUR GROUPS, AND DETERMINES THE NUMBER IN EACH GROUP
C
      NGRP1 = 0
      NGRP2 = 0
      NGRP3 = 0
      NGRP4 = 0
C
10    READ (*, *, END=100) NUM
      IF (NUM .EQ. 0) NUM = 1
      GO TO (20,30,40,50), (NUM + 24)/25
      PRINT '( " NUMBER ",I4," IS OUT OF CORRECT RANGE")', NUM
      GO TO 10
20    NGRP1 = NGRP1 + 1
      GO TO 10
30    NGRP2 = NGRP2 + 1
      GO TO 10
40    NGRP3 = NGRP3 + 1
      GO TO 10
50    NGRP4 = NGRP4 + 1
      GO TO 10
C
100   PRINT 200, NGRP1, NGRP2, NGRP3, NGRP4
200   FORMAT (' 0  - 25 :', I4,/1X,'26 - 50 :', I4,/1X,'51 - 75 :', I4,
     * /1X, '76 - 100:',I4)
      STOP
      END
```

Figure 12-54. Program GOTO

PROGRAM ASK

Program ASK, shown in figure 12-56, illustrates the OPEN, INQUIRE, and CLOSE statements. The program creates a file, writes information to the file, inquires about the status of the file, and closes the file.

The OPEN statement in lines 9 and 10 creates a file named N123 and associates the file with unit 2. File N123 is declared to be a direct access file with a record length of 100 words.

The DO loop in lines 12 through 15 writes 5 records to file N123. One record is written on each pass through the loop. Each record consists of ten consecutive words from the array BUF followed by blank fill. Since N123 is a direct access file, the REC parameter is specified on the WRITE statement to assign a number to each record. A counter K is incremented on each pass through the loop, and the value of K is used for the record number.

The INQUIRE statement in line 16 performs an inquire on unit 2. INQUIRE returns information in the variables supplied for the specified parameters. The variables O and E are declared type logical because INQUIRE returns a logical value (T or F) for the EXIST and OPENED parameters. Variables N, A, S, F, and B are declared type character because INQUIRE returns a character string for the NAME, ACCESS, SEQUENTIAL, and FORM parameters.

Program output is shown in figure 12-57. The FORMAT statement formats the output so that it is self-explanatory. Note that sequential access is not permitted on file N123. The file is opened for unformatted output (default for direct access files), the next record is 6 (5 records have been written), and blanks within a record are ignored (default). The NAME, ACCESS, and RECL parameters reflect information specified on the OPEN statement.

The CLOSE statement in line 26 specifies the STATUS='DELETE' parameter so that the file is destroyed after execution of the CLOSE. If this statement were omitted, an implicit CLOSE(2,STATUS='KEEP') would occur.

```
 UNIT EXISTS? T
 UNIT ASSOCIATED WITH FILE? T
 FILE NAME IS N123
 ACCESS METHOD IS DIRECT
 SEQUENTIAL ACCESS PERMITTED? NO
 OPENED FOR UNFORMATTED I/O
 RECORD LENGTH IS  100
 NEXT RECORD IS      6
```

Figure 12-57. Program ASK Output

```
      PROGRAM ASK
      LOGICAL E, O
      CHARACTER*10 N,A,S,F*11
      DIMENSION BUF(50)
C
      DO 10 I = 1,50
10    BUF(I) = I
C
      OPEN (2, ERR=99, FILE='N123', STATUS='NEW', ACCESS='DIRECT',
     * RECL=100 )
      K=1
      DO 15 I = 1,41,10
      WRITE (2, REC=K, ERR=99) (BUF(J),J=I,I+9)
15    K = K + 1
C
      INQUIRE (UNIT=2, ERR=99, EXIST=E, OPENED=O, NAME=N, ACCESS=A,
     *        SEQUENTIAL=S, FORM=F, RECL=L, NEXTREC=M)
C
      PRINT 50, E,O,N,A,S,F,L,M
50    FORMAT ('1', 'UNIT EXISTS? ', L1, /' UNIT ASSOCIATED WITH FILE? ',
     * L1, /' FILE NAME IS ', A, /' ACCESS METHOD IS ', A,
     * /' SEQUENTIAL ACCESS PERMITTED? ', A,
     * /' OPENED FOR ', A, ' I/O', /' RECORD LENGTH IS ', I5,
     * /' NEXT RECORD IS ', I5)
C
      CLOSE (2, ERR=99, STATUS='DELETE')
      STOP
99    PRINT*, ' FATAL I/O ERROR'
      STOP
      END
```

Figure 12-56. Program ASK

PROGRAM SCORE

Program SCORE, shown in figure 12-58, reads student names and test scores from input records and calls subroutine AVG to compute the average of the scores on each record and to determine which of the students qualify for honors. Program SCORE illustrates the use of an alternate return.

Each input record contains a name and four test scores. After reading a record, the main program calls subroutine AVG which computes the average of the four scores. The actual arguments passed to AVG are an array ISCORE containing the four scores, an integer variable N containing the number of scores, a real variable XLIM, a real variable AV in which AVG returns the computed average, and two statement labels indicated by *8 and *10.

The variables XLIM and N are initialized by the DATA statement in line 4.

Subroutine AVG computes the average of the values in ISCORE and tests the average against XLIM to determine if the student qualifies for honors. The IF statement in line 8 performs the test and returns control to the statement label represented by the first asterisk in the SUBROUTINE statement (label 8) if the test has a value that is true. If the test is not true, control passes to the next statement which returns control to the statement label represented by the second asterisk in the SUBROUTINE statement (label 10).

In the main program, the statement labeled 8 prints the string "HONORS". The statement labeled 10 prints the name and the computed average; the + carriage control character causes these values to appear on the same line as "HONORS".

The program continues to process input records until an end-of-file is detected, at which time control passes to the statement labeled 99 and execution terminates.

Sample input and output for program SCORE are shown in figure 12-59.

Input:

```
SMITH      98 85 89 92
JONES      75 83 80 89
DOE        85 92 95 89
DOAKES     85 89 80 91
```

Output:

```
SMITH          91.00 HONORS
JONES          81.75

DOE            90.25 HONORS
DOAKES         86.25
```

Figure 12-59. Sample Input and Output for Program SCORE

```
      PROGRAM SCORE
      CHARACTER*10 NAME
      DIMENSION ISCORE(4)
      DATA XLIM/90.0/, N/4/
C
6     READ (*, 100, END=12) NAME, (ISCORE(I),I=1,4)
100   FORMAT (A10, 4I3)
      CALL AVG (ISCORE, N, XLIM, AV, *8, *10)
8     PRINT '(21X, "HONORS")'
10    PRINT '("+", A, 3X, F6.2, /)', NAME, AV
      GO TO 6
C
12    STOP
      END
C
      SUBROUTINE AVG(IARR, N, XLIM, AV, *, *)
      DIMENSION IARR(N)
C
      SUM = 0
      DO 20 I = 1,N
20    SUM = SUM + IARR(I)
      AV = SUM/N
      IF (AV .GE. XLIM) RETURN 1
      RETURN 2
      END
```

Figure 12-58. Program SCORE and Subroutine AVG

STANDARD CHARACTER SETS

A

CONTROL DATA operating systems offer the following variations of a basic character set:

- CDC 64-character set
- CDC 63-character set
- ASCII 64-character set
- ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed. The standard character sets are shown in table A-1.

Depending on another installation option, NOS and NOS/BE assume an input deck has been punched either in 026 or 029 mode, regardless of the character set in use. Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified alternate mode remains in effect throughout the job unless reset by another alternate mode specification.

Graphic character representation on a terminal or printer depends on the installation character set and the device type. CDC graphic characters in table A-1 are applicable to BCD terminals. ASCII subset graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

Under SCOPE 2, the alternate modes are: 026, 029, and blank.

The 026 and 029 modes are specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The 26 and 29 codes convert 026 and 029 coded input to display code. Blank entries in columns 79 and 80 indicate that the following section is coded or binary and the next card should be checked according to these alternatives:

If the next card is a free-form flag card, the section following is free-form binary. (See the SCOPE 2 reference manual.)

If the next card has 7/9 punched (only) in column 1, the following section is SCOPE 2 binary. (See the SCOPE 2 reference manual.)

In any other case, the following section is coded with the last requested conversion mode.

When a 63-character set is in use, display code 00 under A or R edit descriptor conversion in a formatted I/O statement, ENCODE statement, or DECODE statement is converted to display code 55_8 (blank). No conversions occur when a 64-character set is in use.

FORTRAN programs can be written to handle 96-character or 128-character ASCII. In general, NOS handling of 96-character or 128-character ASCII involves 6-bit and 12-bit codes, with characters represented in a single display code or double display code combination. The NOS character codes are shown in table A-2. In general, NOS/BE and INTERCOM handling of 96-character or 128-character ASCII involves 8-bit and 12-bit codes, with the 8-bit ASCII code right-justified in a 12-bit field. The ASCII character set is shown in table A-3. See the appropriate operating system manual (NOS reference manual volume 1, NOS/BE reference manual, or the SCOPE 2 reference manual) for details.

The collation weight tables referenced at the end of section 7 are given in table A-4.

TABLE A-1. FORTRAN AND STANDARD CHARACTER SETS

FORTRAN	Display Code (octal)	CDC			ASCII		
		Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
: (colon)	00†	: (colon)††	8-2	00	: (colon) ††	8-2	072
A	01	A	12-1	61	A	12-1	101
B	02	B	12-2	62	B	12-2	102
C	03	C	12-3	63	C	12-3	103
D	04	D	12-4	64	D	12-4	104
E	05	E	12-5	65	E	12-5	105
F	06	F	12-6	66	F	12-6	106
G	07	G	12-7	67	G	12-7	107
H	10	H	12-8	70	H	12-8	110
I	11	I	12-9	71	I	12-9	111
J	12	J	11-1	41	J	11-1	112
K	13	K	11-2	42	K	11-2	113
L	14	L	11-3	43	L	11-3	114
M	15	M	11-4	44	M	11-4	115
N	16	N	11-5	45	N	11-5	116
O	17	O	11-6	46	O	11-6	117
P	20	P	11-7	47	P	11-7	120
Q	21	Q	11-8	50	Q	11-8	121
R	22	R	11-9	51	R	11-9	122
S	23	S	0-2	22	S	0-2	123
T	24	T	0-3	23	T	0-3	124
U	25	U	0-4	24	U	0-4	125
V	26	V	0-5	25	V	0-5	126
W	27	W	0-6	26	W	0-6	127
X	30	X	0-7	27	X	0-7	130
Y	31	Y	0-8	30	Y	0-8	131
Z	32	Z	0-9	31	Z	0-9	132
0	33	0	0	12	0	0	060
1	34	1	1	01	1	1	061
2	35	2	2	02	2	2	062
3	36	3	3	03	3	3	063
4	37	4	4	04	4	4	064
5	40	5	5	05	5	5	065
6	41	6	6	06	6	6	066
7	42	7	7	07	7	7	067
8	43	8	8	10	8	8	070
9	44	9	9	11	9	9	071
+ (plus)	45	+	12	60	+	12-8-6	053
- (minus)	46	-	11	40	-	11	055
* (asterisk)	47	*	11-8-4	54	*	11-8-4	052
/ (slash)	50	/	0-1	21	/	0-1	057
((left paren)	51	(	0-8-4	34	(	12-8-5	050
) (right paren)	52	)	12-8-4	74	)	11-8-5	051
$ (currency)	53	$	11-8-3	53	$	11-8-3	044
= (equals)	54	=	8-3	13	=	8-6	075
blank	55	blank	no punch	20	blank	no punch	040
, (comma)	56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
. (decimal point)	57	. (period)	12-8-3	73	. (period)	12-8-3	056
	60	≡	0-8-6	36	#	8-3	043
	61	[	8-7	17	[	12-8-2	133
	62	]	0-8-2	32	]	11-8-2	135
	63	% ††	8-6	16	% ††	0-8-4	045
" (quote)	64	≠	8-4	14	" (quote)	8-7	042
	65	←	0-8-5	35	_ (underline)	0-8-5	137
	66	v	11-0	52	!	12-8-7	041
	67	∧	0-8-7	37	&	12	046
' (apostrophe)	70	↑	11-8-5	55	' (apostrophe)	8-5	047
	71	↓	11-8-6	56	?	0-8-7	077
	72	<	12-0	72	<	12-8-4	074
	73	>	11-8-7	57	>	0-8-6	076
	74	≤	8-5	15	@	8-4	100
	75	≥	12-8-5	75	\	0-8-2	134
	76	¬	12-8-6	76	~ (circumflex)	11-8-7	136
	77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

†Twelve zero bits at the end of a 60-bit word in a zero byte record are an end-of-record mark rather than two colons.

††In installations using a 63-graphic set, display code 00_8 has no associated graphic or card code; display code 63_8 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_8).

TABLE A-2. CODES (6/12-BIT) FOR NOS

Display Code (6/12-Bit Octal)	Char.	ASCII Code (7-Bit Octal)	ASCII Code (Hexadecimal)	Display Code (6/12-Bit Octal)	Char.	ASCII Code (7-Bit Octal)	ASCII Code (Hexadecimal)
00†	:	072	3A	7604	d	144	64
01	A	101	41	7605	e	145	65
02	B	102	42	7606	f	146	66
03	C	103	43	7607	g	147	67
04	D	104	44	7610	h	150	68
05	E	105	45	7611	i	151	69
06	F	106	46	7612	j	152	6A
07	G	107	47	7613	k	153	6B
10	H	110	48	7614	l	154	6C
11	I	111	49	7615	m	155	6D
12	J	112	4A	7616	n	156	6E
13	K	113	4B	7617	o	157	6F
14	L	114	4C	7620	p	160	70
15	M	115	4D	7621	q	161	71
16	N	116	4E	7622	r	162	72
17	O	117	4F	7623	s	163	73
20	P	120	50	7624	t	164	74
21	Q	121	51	7625	u	165	75
22	R	122	52	7626	v	166	76
23	S	123	53	7627	w	167	77
24	T	124	54	7630	x	170	78
25	U	125	55	7631	y	171	79
26	V	126	56	7632	z	172	7A
27	W	127	57	7633	{	173	7B
30	X	130	58	7634	|	174	7C
31	Y	131	59	7635	}	175	7D
32	Z	132	5A	7636	~	176	7E
33	0	060	30	7637	DEL	177	7F
34	1	061	31	7640	NUL	000	00
35	2	062	32	7641	SOH	001	01
36	3	063	33	7642	STX	002	02
37	4	064	34	7643	ETX	003	03
40	5	065	35	7644	EOT	004	04
41	6	066	36	7645	ENO	005	05
42	7	067	37	7646	ACK	006	06
43	8	070	38	7647	BEL	007	07
44	9	071	39	7650	BS	010	08
45	+	053	2B	7651	HT	011	09
46	-	055	2D	7652	LF	012	0A
47	*	052	2A	7653	VT	013	0B
50	/	057	2F	7654	FF	014	0C
51	(	050	28	7655	CR	015	0D
52	)	051	29	7656	SO	016	0E
53	$	044	24	7657	SI	017	0F
54	=	075	3D	7660	DLE	020	10
55	(space)	040	20	7661	DC1	021	11
56	,	054	2C	7662	DC2	022	12
57	.	056	2E	7663	DC3	023	13
60	#	043	23	7664	DC4	024	14
61	[	133	5B	7665	NAK	025	15
62	]	135	5D	7666	SYN	026	16
63††	%	045	25	7667	ETB	027	17
64	"	042	22	7670	CAN	030	18
65	_†††	137	5F	7671	EM	031	19
66	!	041	21	7672	SUB	032	1A
67	&	046	26	7673	ESC	033	1B
70	'	047	27	7674	FS	034	1C
71	?	077	3F	7675	GS	035	1D
72	<	074	3C	7676	RS	036	1E
73	>	076	3E	7677	US	037	1F
74	@	100	40	7400	null	---	--
75	\	134	5C	7401	@	100	40
76	^	136	5E	7402	^	136	5E
77	;	073	3B	7403	null	---	--
7600	null	---	--	7404	:	072	3A
7601	a	141	61	7405	null	---	--
7602	b	142	62	7406	null	---	--
7603	c	143	63	7407	`	140	60

†In the 63-character set, this display code represents a null character. Also, use of the colon in program and data files may cause problems. This is particularly true when it is used in PRINT and FORMAT statements.

††In the 63-character set, this display code represents a colon (:), 7-bit ASCII code 072, 7-bit hexadecimal code 3A.

†††On TTY models having no underline, the backarrow (←) takes its place.

TABLE A-3. CODES (8-BIT) FOR NOS/BE

Bits	b_8 b_7 b_6 b_5	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
b_4 b_3 b_2 b_1	Hex Digits	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL 12-0-9-8-1 NUL 00	DLE 12-11-9-8-1 DLE 10	SP no-punch SP 40	0 0 0 F0	@ 8-4 @ 7C	P 11-7 P D7	` 8-1 ` 79	p 12-11-7 p 97
0 0 0 1	1	SOH 12-9-1 SOH 01	DC1 11-9-1 DC1 11	! 12-8-7 | 4F	1 1 1 F1	A 12-1 A C1	Q 11-8 Q D8	a 12-0-1 a 81	q 12-11-8 q 98
0 0 1 0	2	STX 12-9-2 STX 02	DC2 11-9-2 DC2 12	" 8-7 " 7F	2 2 2 F2	B 12-2 B C2	R 11-9 R D9	b 12-0-2 b 82	r 12-11-9 r 99
0 0 1 1	3	ETX 12-9-3 ETX 03	DC3 11-9-3 TM 13	# 8-3 # 7B	3 3 3 F3	C 12-3 C C3	S 0-2 S E2	c 12-0-3 c 83	s 11-0-2 s A2
0 1 0 0	4	EOT 9-7 EOT 37	DC4 9-8-4 DC4 3C	$ 11-8-3 $ 5B	4 4 4 F4	D 12-4 D C4	T 0-3 T E3	d 12-0-4 d 84	t 11-0-3 t A3
0 1 0 1	5	ENQ 0-9-8-5 ENQ 2D	NAK 9-8-5 NAK 3D	% 0-8-4 % 6C	5 5 5 F5	E 12-5 E C5	U 0-4 U E4	e 12-0-5 e 85	u 11-0-4 u A4
0 1 1 0	6	ACK 0-9-8-6 ACK 2E	SYN 9-2 SYN 32	& 12 & 50	6 6 6 F6	F 12-6 F C6	V 0-5 V E5	f 12-0-6 f 86	v 11-0-5 v A5
0 1 1 1	7	BEL 0-9-8-7 BEL 2F	ETB 0-9-6 ETB 26	' 8-5 ' 7D	7 7 7 F7	G 12-7 G C7	W 0-6 W E6	g 12-0-7 g 87	w 11-0-6 w A6
1 0 0 0	8	BS 11-9-6 BS 16	CAN 11-9-8 CAN 18	(12-8-5 (4D	8 8 8 F8	H 12-8 H C8	X 0-7 X E7	h 12-0-8 h 88	x 11-0-7 x A7
1 0 0 1	9	HT 12-9-5 HT 05	EM 11-9-8-1 EM 19	) 11-8-5) 5D	9 9 9 F9	I 12-9 I C9	Y 0-8 Y E8	i 12-0-9 i 89	y 11-0-8 y A8
1 0 1 0	10 (A)	LF 0-9-5 LF 25	SUB 9-8-7 SUB 3F	* 11-8-4 * 5C	: 8-2 : 7A	J 11-1 J D1	Z 0-9 Z E9	j 12-11-1 j 91	z 11-0-9 z A9
1 0 1 1	11 (B)	VT 12-9-8-3 VT 0B	ESC 0-9-7 ESC 27	+ 12-8-6 + 4E	; 11-8-6 ; 5E	K 11-2 K D2	[12-8-2 ¢ 4A	k 12-11-2 k 92	{ 12-0 { C0
1 1 0 0	12 (C)	FF 12-9-8-4 FF 0C	FS 11-9-8-4 IFS 1C	, 0-8-3 , 6B	< 12-8-4 < 4C	L 11-3 L D3	\ 0-8-2 \ E0	l 12-11-3 l 93	¦ 12-11 ¦ 6A
1 1 0 1	13 (D)	CR 12-9-8-5 CR 0D	GS 11-9-8-5 IGS 1D	– 11 – 60	= 8-6 = 7E	M 11-4 M D4	] 11-8-2 ! 5A	m 12-11-4 m 94	} 11-0 } D0
1 1 1 0	14 (E)	SO 12-9-8-6 SO 0E	RS 11-9-8-6 IRS 1E	. 12-8-3 . 4B	> 0-8-6 > 6E	N 11-5 N D5	^ 11-8-7 ¬ 5F	n 12-11-5 n 95	~ 11-0-1 ~ A1
1 1 1 1	15 (F)	SI 12-9-8-7 SI 0F	US 11-9-8-7 IUS 1F	/ 0-1 / 61	? 0-8-7 ? 6F	O 11-6 O D6	_ 0-8-5 _ 6D	o 12-11-6 o 96	DEL 12-9-7 DEL 07

64-character ASCII

95-character ASCII (does not include DEL)

128-character ASCII

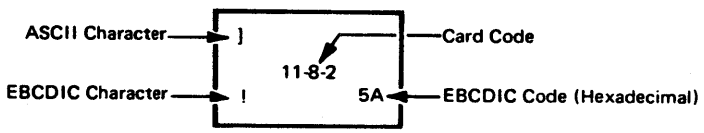


TABLE A-4. COLLATING WEIGHT TABLES

CDC Graphic	Octal Character Code	Decimal Weights			CDC Graphic	Octal Character Code	Decimal Weights		
		ASCII6	COBOL6	Display			ASCII6	COBOL6	Display
: (colon)	00†	26	53	0	5	40	21	59	32
A	01	33	25	1	6	41	22	60	33
B	02	34	26	2	7	42	23	61	34
C	03	35	27	3	8	43	24	62	35
D	04	36	28	4	9	44	25	63	36
E	05	37	29	5	+	45	11	15	37
F	06	38	30	6	-	46	13	18	38
G	07	39	31	7	*	47	10	17	39
H	10	40	32	8	/	50	15	19	40
I	11	41	33	9	(	51	8	21	41
J	12	42	35	10	)	52	9	13	42
K	13	43	36	11	$	53	4	16	43
L	14	44	37	12	=	54	29	22	44
M	15	45	38	13	blank	55	0	0	45
N	16	46	39	14	, (comma)	56	12	20	46
O	17	47	40	15	. (period)	57	14	12	47
P	20	48	41	16	≡	60	3	5	48
Q	21	49	42	17	⌊	61	59	3	49
R	22	50	43	18	⌋	62	61	44	50
S	23	51	45	19	%	63†	5	2	51
T	24	52	46	20	≠	64	2	23	52
U	25	53	47	21	⌐	65	63	4	53
V	26	54	48	22	∨	66	1	34	54
W	27	55	49	23	∧	67	6	6	55
X	30	56	50	24	↑	70	7	7	56
Y	31	57	51	25	↓	71	31	8	57
Z	32	58	52	26	<	72	28	24	58
0	33	16	54	27	>	73	30	9	29
1	34	17	55	28	≤	74	32	1	60
2	35	18	56	29	≥	75	60	10	61
3	36	19	57	30	¬	76	62	11	62
4	37	20	58	31	; (semicolon)	77	27	14	63

†In installations using the 63-graphic set, the octal character code 00 does not exist, and the weights in the ASCII6 and COBOL6 tables for the octal character code 63 assume the corresponding weights from character code 00.

Diagnostic messages are issued by FORTRAN 5 during both compilation and execution to inform the user of errors in the source program, input data, or intermediate results. This appendix explains the content and format of the FORTRAN 5 diagnostic messages.

COMPILE-TIME DIAGNOSTICS

When an error is detected during compilation of the source program, a diagnostic message is issued immediately after the erroneous source line. The format of the diagnostics is:

```
severity * message
```

The severity indicator tells the consequences the error will have on further processing of the program. One of the following severity indicators will accompany each error message:

FATAL	The program will not be executed.
WARNING	The error is severe, but the program will be executed. Although syntax is incorrect, the probable meaning of the source code is presumed.
TRIVIAL	A minor syntax error or omission was detected, or correct syntax was used but semantics were irregular.
ANSI	Usage does not conform to ANSI X3.9 - FORTRAN 77 specification. Listed only if the ANSI list option is specified on the FTN5 control statement.
MDEP	The line contains a use of a machine-dependent language feature. Listed only if the MD option is specified on the FTN5 control statement.

The compile-time diagnostics issued by FORTRAN 5 are summarized in alphabetical order in table B-1. Ellipses, denoted by ..., are replaced by items from the relevant source statement.

SPECIAL COMPILATION DIAGNOSTICS

When a compilation is aborted or prematurely terminated for internal reasons, one or more of the messages shown in table B-2 appear. This table also includes messages that appear only in the dayfile that are not caused by internal error.

COMPILER OUTPUT LISTING MESSAGES

Compiler output listing messages are printed in the source listing. They may appear before, during, or after the reference map and object code listings, depending on the error condition. The message format is different than that of the standard error summary; each message is usually left-justified on the output page, and may be preceded by several blank lines, or may be printed at the top of a page.

The compiler output listing messages are given in table B-3.

EXECUTION DIAGNOSTICS

Execution diagnostics are issued when an error occurs while a user program is running. The diagnostics are printed on the source listing in one of the following formats:

```
ERROR NUMBER x DETECTED BY routine
AT ADDRESS y
```

or

```
ERROR NUMBER x DETECTED BY routine
CALLED FROM routine AT ADDRESS z
```

or

```
ERROR NUMBER x DETECTED BY routine
CALLED FROM routine AT LINE d
```

where y and z are relative octal addresses, x is a decimal error number, and d is a decimal line number corresponding to a line number printed in the source listing.

Table B-4 summarizes the execution diagnostics by error number. In table B-4, the letters under Class mean:

F = Fatal

I = Informative, nonfatal

D = Debug (diagnostic can be issued only when in debug mode)

T = Trace (diagnostic can be issued only when in trace mode)

A = Always (diagnostic can always be issued)

TABLE B-1. COMPILE-TIME DIAGNOSTICS

Message		Significance	Action
ANSI	..... IS DEFINED TO BE INTRINSIC	The FORTRAN 5 defined intrinsic function is not supported in ANSI FORTRAN.	Supply the function for portability.
ANSI	..... IS NON-ANSI EDIT DESCRIPTOR	Nonstandard format specification.	Replace format specification.
ANSI	7 CHARACTER SYMBOL ..... IS NON-ANSI	ANSI allows only 6 characters.	Shorten symbol to 6 characters or less.
ANSI	CHARACTER ARRAY REQUIRED FOR FORMAT SPECIFIER	Format must be contained in character array.	Use an array of character type.
ANSI	COMMON BLOCK NAME ..... CANNOT BE	Common block name used as another symbol name in a nonANSI manner (for example, as an entry point name or as an intrinsic function name.)	Change the common block name or, when possible, the symbol name.
ANSI	COMMON CAN BE PRESET IN BLOCK DATA ONLY	ANSI allows COMMON to be preset in block data only.	Remove presetting of COMMON.
ANSI	COMPUTED GO TO INDEX MUST BE INTEGER	Index is of incorrect type.	Change GO TO index or declare it to be integer.
ANSI	DOUBLE PRECISION AND COMPLEX OPERANDS ARE MIXED	Cannot mix DOUBLE PRECISION and COMPLEX operands.	Apply REAL function to DOUBLE PRECISION operand.
ANSI	FILE DECLARATION LIST NON-ANSI	ANSI does not permit file declaration in the PROGRAM statement.	Remove file list from PROGRAM statement.
ANSI	FUNCTION REFERENCE IN CONSTANT EXPRESSION	ANSI does not allow function reference in constant expression.	Remove function reference.
ANSI	HOLLERITH CONSTANT NON-ANSI	ANSI uses character data type.	Switch usage to character.
ANSI	I/O KEYWORD BUFL IS NON-ANSI	ANSI does not permit I/O keyword BUFL.	Remove I/O keyword BUFL.
ANSI	MASK EXPRESSION NON-ANSI	ANSI does not permit mask expressions.	Remove mask expression.
ANSI	MULTIPLE ASSIGNMENT IS NON-ANSI	ANSI permits only one assignment per statement.	Break assignment statement into two or more statements.
ANSI	NAMELIST I/O IS NON-ANSI	ANSI does not permit NAMELIST I/O.	Remove NAMELIST I/O.
ANSI	OBJECT OF IF IS ILLEGAL DO TERMINATOR	A logical IF, used as the last statement in a DO loop, contains a nonstandard statement.	Change object of IF. Make the last statement in the loop a CONTINUE statement.
ANSI	OCTAL DATA TYPE NOT DEFINED IN ANSI	ANSI does not permit octal data type.	Write number as decimal.
ANSI	PAREN REPEAT LIST IS NOT PERMITTED	Repeated item list is not provided in standard FORTRAN.	Remove paren repeat list.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
ANSI	RETURN IN MAIN PROGRAM -- ACTS AS END	RETURN is considered the END statement in main program.	Change RETURN to END or STOP.
ANSI	STATEMENT FUNCTION ACTUAL ARGUMENT MUST AGREE IN TYPE WITH DUMMY ARGUMENT	ANSI requires that dummy and actual arguments to statement functions agree in type.	Change type declaration of dummy or actual argument.
ANSI	STATEMENT FUNCTION DUMMY ARGUMENT ... CANNOT BE AN ARRAY	Declaration of dummy argument is invalid.	Change declaration of dummy argument or name of dummy argument.
ANSI	STATEMENT IS NOT DEFINED IN ANSI	ANSI does not recognize statement.	Correct statement.
ANSI	SUBSCRIPT ..... OF ..... IS NOT TYPE INTEGER	ANSI requires integer subscripts.	Assign subscript expression to an integer variable and use the variable.
ANSI	SYMBOLIC CONSTANT IN COMPLEX CONSTANT NOT ANSI	Symbolic constant in complex constant is not allowed by ANSI.	Replace symbolic constant with constant.
ANSI	TRANSFER INTO RANGE OF DO	Cannot transfer into range of DO.	Rewrite loops to be closed.
FATAL	..... EDIT DESCRIPTOR REQUIRES COUNT	Program will not execute without count.	Supply a count for the edit descriptor.
FATAL	..... EXPRESSION NOT CONSTANT, OR NOT EVALUATABLE	Expression, which must be a constant, will not reduce.	Rewrite statement.
FATAL	..... -- ILLEGAL TRANSFER TO INSIDE A CLOSED DO LOOP OR IF BLOCK	To branch inside a DO loop, a branch must previously have been made out of the loop. Branching into an IF block is illegal.	Revise program flow to remove invalid branch.
FATAL	.... I/O CONTROL ALREADY SPECIFIED FOR THIS STATEMENT	Duplication of I/O specifier is invalid.	Remove duplicate I/O specifier.
FATAL	.... NOT I/O CONTROL KEYWORD	I/O control keyword not recognized.	Likely to be a misspelled keyword. Correct it.
FATAL	.... NOT LEGAL I/O CONTROL KEYWORD FOR THIS STATEMENT	Valid I/O keyword but not for this statement.	Remove I/O control keyword.
FATAL	..... BLOCK IF(S) NOT TERMINATED	Missing ENDIF statement.	Insert ENDIF statement.
FATAL	..... C$ IF(S) NOT TERMINATED	Missing C$ ENDIF statement.	Insert C$ ENDIF statement.
FATAL	..... CANNOT HAVE ASSUMED CHARACTER LENGTH	Only symbolic constants and dummy arguments may have (*) length.	Remove (*) length declaration.
FATAL	..... CAUSES CHARACTER DECLARATION CONFLICT IN EQUIVALENCE GROUP	Character declaration conflict encountered in EQUIVALENCE statement.	Check declarations of equivalenced character variables.
FATAL	..... ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN	The EQUIVALENCE statement has extended the common block origin backward.	Check all EQUIVALENCE statements containing the specified variable.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	..... ILLEGAL FIRST ELEMENT OF EXPRESSION	First element of expression found to be invalid.	Correct first element of expression.
FATAL	..... IN INPUT LIST IS ILLEGAL	Constants and expressions cannot appear in input lists.	Remove constant or expression.
FATAL	..... IS IN BLANK COMMON -- DATA IGNORED	Blank common variables must not be initialized.	Remove blank common variables from DATA statement.
FATAL	..... IS NOT DEFINED AS INTRINSIC	Name is not the name of a builtin intrinsic function.	Remove name or correct spelling.
FATAL	..... IS UNKNOWN C$ PARAMETER FOR	C$ parameter not recognized for this expression.	Correct C$ parameter or expression.
FATAL	..... MUST BE DO CONTROL VARIABLE	Expression must be a DO control variable.	Make expression a DO control variable.
FATAL	..... MUST BE A DUMMY-ARG	Expression must be a dummy-arg.	Make expression a dummy-arg.
FATAL	..... MUST BE INTEGER CONSTANT EXPRESSION	Expression must be an integer constant.	Make expression an integer constant.
FATAL	..... OPERAND CANNOT BE CONVERTED TO TYPE	The operand cannot be converted to the type attempted.	Check operand and type.
FATAL	PREVIOUSLY USED IN EXECUTABLE OR CONFLICTING DECLARATIVE	Dummy argument on ENTRY statement had previous use that prohibits use as a dummy argument.	Correct the previous usage or change the name of the dummy argument.
FATAL	REDEFINES A DO CONTROL INDEX	Variable redefines a current DO index.	Change variable usage. Check equivalence declarations.
FATAL	..... SUBSCRIPT OUTSIDE OF ARRAY BOUNDS	Subscript must be inside of array bounds.	Check subscript and dimension statement.
FATAL	3 BRANCH IF HAS ..... EXPRESSION	3 branch IF expression must be integer, real, or boolean.	Change type of expression to integer, real, or boolean.
FATAL	3 BRANCH IF MISSING LABEL	Label required for 3 branch IF.	Supply a label.
FATAL	ADJUSTABLE BOUND ..... MUST BE DUMMY-ARG OR IN COMMON	Variable used as a dimension bound must be a dummy-arg or in common.	Add variable to dummy-arg list or to common block.
FATAL	ADJUSTABLE DIMENSION BOUND ..... IS NOT INTEGER	Adjustable dimension bound must be integer.	Declare adjustable dimension bound to be integer.
FATAL	ALTERNATE RETURN IS ILLEGAL IN A FUNCTION	Legal only in a subroutine.	Remove alternate return.
FATAL	ARGUMENT COUNT ON ..... EXCEEDS 500	Too many arguments.	Reduce number of arguments.
FATAL	ARGUMENT COUNT ON ..... MUST BE MORE THAN ONE	Not enough arguments.	Increase number of arguments.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	ARGUMENT COUNT ON INTRINSIC ..... IS WRONG	Wrong number of arguments supplied for the intrinsic function.	Check syntax of the intrinsic function.
FATAL	ARGUMENT MODE ILLEGAL FOR GENERIC FUNCTION	Improper argument type.	Check definition of function to determine correct argument type.
FATAL	ARGUMENT MODE MUST AGREE WITH TYPE DEFINED FOR LIBRARY FUNCTION	Improper argument type.	Check definition of function to determine correct argument type.
FATAL	ARRAY .... DIMENSION .... -- DIMENSION BOUND EXPRESSION CONTAINS ILLEGAL OPERATION	Illegal expression in DIMENSION declaration.	Correct expression.
FATAL	ARRAY .... DIMENSION .... -- DIMENSION BOUND EXPRESSION CONTAINS NON-VARIABLE	Illegal expression in DIMENSION declaration.	Correct invalid variables in expression.
FATAL	ARRAY .... DIMENSION .... -- DIMENSION BOUND EXPRESSION CONTAINS ARRAY REFERENCE	Array references in expressions are illegal when the expression appears in a DIMENSION statement.	Remove all array references from expression.
FATAL	ARRAY ..... DIMENSION ..... -- LOWER BOUND EXCEEDS UPPER BOUND	Lower bound must be less than or equal to upper bound.	Correct dimension boundaries.
FATAL	ARRAY ..... DIMENSION ..... -- EXCEEDS $2^{**23}-1$	Dimension value too large.	Reduce dimension size.
FATAL	ARRAY ..... DIMENSION BOUND ..... NOT INTEGER	Dimension bounds must be integer.	Declare dimension bound to be integer.
FATAL	ARRAY ..... EXCEEDS 7 DIMENSIONS	Too many dimensions.	Reduce number of dimensions.
FATAL	ARRAY ..... MISSING SUBSCRIPT	Subscript required to reference an array element.	Supply subscript.
FATAL	ARRAY ..... SIZE EXCEEDS $2^{**23}-1$	Array too large.	Reduce size of array.
FATAL	ARRAY ..... SUBSCRIPT COUNT DOES NOT MATCH DIMENSION COUNT	Wrong number of subscripts supplied. The number of subscripts in an array reference must equal the number specified in the DIMENSION statement.	Check the number of subscripts on the DIMENSION statement.
FATAL	ARRAY DECLARATION FOR ..... MISSING RIGHT PAREN	Right parenthesis missing.	Supply right parenthesis.
FATAL	ASSUMED CHARACTER LENGTH ILLEGAL FOR IMPLICIT	Length must be declared in the IMPLICIT statement.	Declare character length.
FATAL	ASSUMED SIZE ARRAY ..... NOT ALLOWED IN I/O LIST	Assumed size array must have a subscript when appearing in an I/O list.	Specify a subscript.
FATAL	ASSUMED SIZE ARRAY NOT PERMITTED IN NAMELIST	Assumed size array must be subscripted when it appears in a namelist.	Specify a subscript.
FATAL	ASSUMED SIZE CAN ONLY BE ON LAST UPPER BOUND	Assumed size is not last upper bound.	Declare size when not last upper bound.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	ASSUMED SIZE OR ADJUSTABLE ARRAY ..... MUST BE DUMMY-ARG	Assumed size or adjustable array is not dummy-arg.	Make assumed size or adjustable array dummy-arg.
FATAL	BUFFER DIRECTION SPECIFIER MUST BE IN OR OUT	BUFFER statement incorrect; correct form is BUFFER IN or BUFFER OUT.	Make BUFFER statement: BUFFERIN or BUFFEROUT.
FATAL	BUFFER I/O ADDRESS CANNOT BE CHARACTER	Buffer I/O address must not be character.	Change Buffer I/O address.
FATAL	BUFFER I/O ADDRESS CANNOT BE	Buffer I/O address is not recognized.	Correct Buffer I/O address.
FATAL	BUFFER I/O LWA MUST BE GREATER THAN OR EQUAL TO FWA	Last-word-address must be greater than or equal to first-word-address.	Correct word-address boundaries.
FATAL	BUFFER I/O PARITY SPECIFIER MUST BE INTEGER CONSTANT OR VARIABLE	Buffer I/O parity specifier not recognized.	Correct Buffer I/O parity specifier.
FATAL	BUFFER I/O PARITY INDICATOR VALUE MUST BE ZERO OR 1	Buffer I/O parity indicator not zero or 1.	Make Buffer I/O parity indicator zero or 1.
FATAL	BUFFER LENGTH FOR FILE ..... EXCEEDS 360000B -- DEFINITION IGNORED	Buffer length too long.	Reduce buffer length.
FATAL	C$ IF EXPRESSION MUST BE LOGICAL	C$ IF expression is not type logical.	Make expression type logical.
FATAL	C$ ..... LABEL ..... DIFFERENT FROM C$ IF LABEL	Label on C$ IF does not match C$ ENDIF or C$ ELSE label.	Make labels identical.
FATAL	CALL STATEMENT MISSING ROUTINE NAME	The correct form is CALL routine-name (parameter list).	Insert routine name between CALL keyword and parameter list.
FATAL	CHARACTER AND OTHER TYPE OPERANDS MAY NOT BE MIXED	Character operands cannot be mixed with non-character operands.	Correct operands.
FATAL	CHARACTER DECLARATION CONFLICT EXISTS IN COMMON BLOCK	Common block contains character and non-character entities.	Make all common block members either type character or type non-character.
FATAL	CHARACTER LENGTH GREATER THAN $2^{**15}-1$	Character variable too long.	Shorten character variable.
FATAL	CHAR LENGTH NOT POSITIVE CONSTANT, (POSITIVE CONSTANT EXPRESSION) OR (*)	The length on a CHARACTER or type declaration was negative or zero.	Correct the length specification.
FATAL	CHARACTER LENGTH ZERO ILLEGAL	Length must be at least 1.	Correct character length value.
FATAL	CHARACTER LENGTHS OF ENTRY ..... AND FUNCTION CANNOT DISAGREE	Character lengths of entry and function disagree.	Correct disagreement.
FATAL	CHARACTER OPERAND ..... USED WITH OPERATOR	Operation illegal for character variable.	Correct conflict.
FATAL	COMMA BEFORE AN I/O LIST IS ALLOWED ONLY ON SHORT FORM READ OR PRINT STATEMENT	The comma before the I/O list is not allowed here.	Remove comma.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	COMMA OR E.O.S. MUST FOLLOW LEVEL LIST NAME	Comma or end of statement expected; statement contains extraneous information.	Correct statement.
FATAL	COMMON BLOCK ..... CANNOT BE DECLARED LEVEL 0	Wrong level declared for this block.	Declare correct level.
FATAL	COMMON BLOCK ..... EXCEEDS MAX BLOCK LENGTH 131071	Common block too large.	Break common block into two or more common blocks.
FATAL	COMMON BLOCK ..... EXCEEDS MAX LCM=G BLOCK LENGTH 1048568	Common block too large.	Break common block into two or more common blocks.
FATAL	COMMON ELEMENT ..... MAY NOT APPEAR IN SAVE	Names of entities in a common block may not appear in the SAVE statement.	Correct the SAVE statement.
FATAL	CONCATENATION OF ASSUMED LENGTH VARIABLE NOT ALLOWED HERE	Assumed length variable cannot be concatenated in this circumstance.	Do not concatenate variable here.
FATAL	CONFLICT IN EQUIVALENCE SPECIFICATION FOR	Indicated EQUIVALENCE is inconsistent with previous EQUIVALENCE.	Check all EQUIVALENCE statements containing the specified variable.
FATAL	CONSTANT CANNOT BE CONVERTED	Constant contains syntax error.	Correct syntax error in constant.
FATAL	CONSTANT DIVIDE BY ZERO -- RESULTS SET TO INFINITE	Division by zero is an undefined operation.	Correct division error.
FATAL	DATA INTO ..... IS ILLEGAL	DATA statement attempts to initialize something which cannot be initialized, such as a formal parameter.	Correct DATA statement.
FATAL	DATA VARIABLE LIST CONTAINS	DATA variable list contains a constant or an expression.	Correct DATA statement.
FATAL	DECIMAL POINT IS NOT SPECIFIED FOR THE EDIT DESCRIPTOR AT	Decimal point is invalid in this circumstance.	Remove decimal point.
FATAL	DECIMAL POINT REQUIRED IN EDIT DESCRIPTOR AT	Decimal point required.	Supply decimal point.
FATAL	DIMENSION ON ..... IGNORED -- PRIOR DIMENSION RETAINED	A dimension was specified more than once; first declaration is used.	Eliminate second declaration.
FATAL	DIRECT ACCESS I/O CANNOT BE FREE FORMAT	FORMAT specification needed.	Replace * with format specification.
FATAL	DIRECT ACCESS I/O CANNOT BE NAMELIST	FORMAT specification needed.	Replace namelist name with format specification.
FATAL	DIRECT ACCESS I/O CANNOT SPECIFY END	END option is illegal.	Remove END= specifier from I/O statement.
FATAL	DO ..... PARAMETER CANNOT BE	Type of the DO parameter is invalid.	Change the type of the parameter.
FATAL	DO-IMPLIED LOOPS IN DATA MUST BE INTEGER	DO-implied loops are required to be integer.	Make DO-implied loops integer.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	DO INDEX ..... MUST BE SIMPLE VARIABLE	DO index is required to be a simple variable.	Make DO index a simple variable.
FATAL	DO INDEX CANNOT BE	Type of DO index is invalid.	Change the type of the DO index.
FATAL	DO LOOP ..... CONTAINS UNCLOSED IF BLOCK	Entire IF block must be within the range of the DO loop.	Make IF block within range of DO loop.
FATAL	DO LOOP ..... MUST TERMINATE WITHIN IF BLOCK	Entire DO loop must be within the range of the IF block.	Make DO loop within range of IF block.
FATAL	DO LOOP ..... NOT TERMINATED BEFORE END OF PROGRAM	DO loop terminator missing.	Add DO terminator statement number where appropriate.
FATAL	DO LOOP ..... PREVIOUSLY DEFINED -- ILLEGAL NESTING	The label was previously used.	Choose a new statement number for the DO.
FATAL	DO LOOP INCREMENT MAY NOT BE ZERO	DO loop increment is required to be nonzero.	Provide nonzero increment.
FATAL	DUMMY-ARG FUNCTION ..... CANNOT HAVE ASSUMED CHARACTER LENGTH	Dummy-arg function has assumed character length.	Specify length of character dummy-arg.
FATAL	DUMMY ARGUMENT ..... CAN OCCUR ONLY ONCE IN ..... DEFINITION	Dummy argument previously defined in current statement function.	Remove excess dummy argument.
FATAL	DUMMY ARGUMENT ..... CANNOT BE EQUIVALENCED	Dummy argument must not appear in EQUIVALENCE statement.	Remove dummy argument from EQUIVALENCE statement.
FATAL	DUMMY ARGUMENT ..... MAY NOT APPEAR IN SAVE	Dummy argument must not appear in SAVE statement.	Remove dummy argument from SAVE statement.
FATAL	DUMMY ARGUMENT ..... MUST BEGIN WITH LETTER OR STAR	Dummy argument must begin with a letter or star.	Correct dummy argument.
FATAL	E.O.S. BEFORE END OF HOLLERITH COUNT	Premature end of statement encountered.	Check for incorrect hollerith count.
FATAL	EDIT DESCRIPTOR MISSING AT	Error in FORMAT statement.	Supply edit descriptor.
FATAL	ELSEIF EXPRESSION MUST BE LOGICAL	ELSEIF expression is not type logical.	Make ELSEIF expression type logical.
FATAL	ELSEIF REQUIRES THEN	THEN is missing from ELSEIF construct.	Add THEN where appropriate.
FATAL	EMPTY COMMON BLOCK	Common block contains no elements.	Remove COMMON statement or add variable list.
FATAL	END LINE ABSENT	END statement must be last statement in source deck.	Add END statement.
FATAL	END OR ERR REQUIRES STATEMENT LABEL	The END= or ERR= in a READ statement must be followed by the label number of an executable statement.	Provide statement label.
FATAL	ENTRY INSIDE DO LOOP OR IF BLOCK IS ILLEGAL	Illegal entry into range of DO loop or IF block.	Remove ENTRY or rewrite loop or block.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	EQUAL SIGN MUST BE FOLLOWED BY NAME, NUMBER OR SLASH	Equal sign required to be followed by a name, number, or a slash.	Correct expression after equal sign.
FATAL	EQUIVALENCED ARRAY ..... HAS SUBSCRIPT LESS THAN DIMENSION LOWER BOUND	Subscript must be greater than or equal to lower bound specified in the DIMENSION statement.	Change subscript or dimension.
FATAL	EQUIVALENCED ARRAY ..... HAS SUBSCRIPT WHICH EXCEEDS DIMENSION BOUND	Subscript must be less than or equal to upper bound specified in DIMENSION.	Change subscript or DIMENSION statement.
FATAL	EXCESS LEFT PAREN IN I/O LIST	Too many left parens.	Remove excess paren(s).
FATAL	EXCESS LEFT PAREN IN I/O LIST ITEM SUBSCRIPT	Too many left parens.	Remove excess paren(s).
FATAL	EXCESS RIGHT PAREN IN I/O LIST	Too many right parens.	Remove excess paren(s).
FATAL	EXCESS SUBSCRIPTS ON EQUIVALENCE VARIABLE	EQUIVALENCE variable has more subscripts than declared in DIMENSION.	Change subscripts or DIMENSION statement.
FATAL	EXECUTABLE STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM	Illegal executable statements in block data subprogram.	Remove executable statements.
FATAL	EXPECTED C$ DIRECTIVE LABEL -- FOUND	C$ directive label expected.	Check C$ directive keyword specification.
FATAL	EXPECTED C$ PARAMETER -- FOUND	C$ parameter expected.	Check C$ directive keyword specification.
FATAL	EXPECTED COMMA -- FOUND	Comma expected.	Check syntax of statement.
FATAL	EXPECTED COMMA AFTER COUNT -- FOUND	Comma after count expected.	Check syntax of statement.
FATAL	EXPECTED COMMA AFTER FORMAT SPECIFIER -- FOUND	Comma after format specifier expected.	Check syntax of statement.
FATAL	EXPECTED COMMA OR RIGHT PAREN -- FOUND	Comma or right paren expected.	Check syntax of statement.
FATAL	EXPECTED COMMA OR SLASH FOUND	Comma or slash expected.	Check syntax of statement.
FATAL	EXPECTED DO CONTROL INDEX -- FOUND	Syntax error in DO statement.	Check syntax of DO statement.
FATAL	EXPECTED E.O.S. -- FOUND	Extraneous information follows a legal statement.	Remove extra information.
FATAL	EXPECTED E.O.S. -- FOUND AND IGNORED	End of statement expected.	Check syntax of statement.
FATAL	EXPECTED EQUAL SIGN -- FOUND	Equal sign expected.	Check syntax of statement.
FATAL	EXPECTED FORMAT SPECIFIER -- FOUND	Format specifier expected.	Check format statement.
FATAL	EXPECTED FILE NAME, FOUND	File name expected.	Correct statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	EXPECTED INTRINSIC FUNCTION NAME -- FOUND	Intrinsic function name expected.	Check intrinsic statement.
FATAL	EXPECTED LEFT PAREN -- FOUND	Left parenthesis expected.	Check syntax of statement.
FATAL	EXPECTED LEFT PAREN BEFORE COUNT -- FOUND	Left parenthesis before count expected.	Check syntax of statement.
FATAL	EXPECTED LEFT PAREN FOR AN ARGUMENT LIST FOUND	Left parenthesis for an argument list expected.	Check syntax of statement.
FATAL	EXPECTED LEFT PAREN OR PERIOD -- FOUND	Left parenthesis or period expected.	Check syntax of statement.
FATAL	EXPECTED NAME -- FOUND	Name expected.	Correct statement.
FATAL	EXPECTED RANGE INDICATOR -- FOUND	Range indicator expected.	Correct statement.
FATAL	EXPECTED RIGHT PAREN -- FOUND	Right parenthesis expected.	Check syntax of statement.
FATAL	EXPECTED RIGHT PAREN AFTER STRING ADDRESS -- FOUND	Right parenthesis after string address expected.	Check syntax of statement.
FATAL	EXPECTED RIGHT PAREN OR COMMA -- FOUND	Right parenthesis or comma expected.	Check syntax of statement.
FATAL	EXPECTED SLASH -- FOUND	Slash expected.	Check syntax of statement.
FATAL	EXPECTED SYMBOL -- FOUND ..... STATEMENT SCAN STOPPED	Symbol expected; scan of statement stopped.	Check syntax of statement.
FATAL	EXPECTED VARIABLE OR COMMON BLOCK NAME -- FOUND	Common block name, enclosed in slashes, must follow COMMON keyword for named common blocks. A variable list must follow COMMON keyword for blank common.	Correct statement.
FATAL	EXPONENT FIELD ON EDIT DESCRIPTOR AT ..... IS ZERO OR NOT SPECIFIED	Exponent field is invalid.	Correct exponent field.
FATAL	EXPRESSION TOO COMPLICATED -- SCAN STOPPED AT	Expression too complicated; scan stopped.	Simplify expression using two or more statements.
FATAL	EXTERNAL UNIT SPECIFIER NOT INTEGER EXPRESSION	External unit specifier must be integer expression.	Make external unit specifier integer expression.
FATAL	EXTRA CHARACTERS .... AFTER UNIT SPECIFIER IGNORED	Extraneous information follows a legal unit specifier.	Remove extra characters.
FATAL	EXTRANEOUS NUMERIC FIELD IN EDIT DESCRIPTOR AT	Invalid numeric field in edit descriptor.	Remove extra numeric field.
FATAL	FIELD WIDTH NOT SPECIFIED FOR EDIT DESCRIPTOR AT	Field width required.	Supply field width.
FATAL	FIELD WIDTH OF EDIT DESCRIPTOR AT ..... IS ZERO OR NOT SPECIFIED	Field width is invalid.	Correct field width.
FATAL	FILE ..... PREVIOUSLY DEFINED -- IGNORED	File already defined.	Self-explanatory.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	FILE ..... NOT DEFINED -- DEFINITION IGNORED	File is not defined.	Define file.
FATAL	FORMAT SPECIFIER ..... IS NAMELIST NAME	Format specifier cannot be NAMELIST name.	Correct format specifier.
FATAL	FORMAT LABEL PREVIOUSLY REFERENCED AS CONTROL STATEMENT LABEL	Label being referenced or defined as a format label was previously referenced as a control statement label.	Check all references to the label in question for consistent usage.
FATAL	FORMAT LABEL PREVIOUSLY REFERENCED AS DO STATEMENT LABEL	Label being referenced or defined as a format label was previously referenced as a DO statement label.	Check all references to the label in question for consistent usage.
FATAL	FORMAT MUST HAVE STATEMENT LABEL	Format is required to have statement label.	Provide a unique statement label for each FORMAT statement.
FATAL	FUNCTION ENTRY ..... MAY NOT BE TYPE CHARACTER	Function entry must not be type character.	Make function entry noncharacter.
FATAL	FUNCTION ENTRY ..... MUST BE TYPE CHARACTER	All entries in a character function must be of type character.	Make function entry type character.
FATAL	FUNCTION NAME OR ENTRY OF TYPE ..... WAS NOT ASSIGNED A VALUE	The function name or entry must be assigned a value within the function.	Assign a value to the function name or entry within the function.
FATAL	FUNCTION REQUIRES EXPLICIT NULL ARGUMENT LIST	A null argument list is a left parenthesis followed immediately by a right parenthesis.	Provide null argument list after the function name in the function reference.
FATAL	GROUP NAME ..... PREVIOUSLY DEFINED	The group name appears twice in the same NAME-LIST statement or in a previous NAMELIST statement.	Check for duplicate name-list group names.
FATAL	HEADER CARD NOT FIRST STATEMENT -- IGNORED	PROGRAM, SUBROUTINE, BLOCK DATA, or FUNCTION must be the first statement of a program.	Correct first statement of program.
FATAL	I/O CONTROL KEYWORD .... MUST BE POSITIVE INTEGER EXPRESSION	I/O control keyword is required to be positive integer expression.	Make I/O keyword positive expression.
FATAL	I/O CONTROL KEYWORD PARAMETER .... CANNOT BE	I/O control keyword parameter is invalid.	Correct I/O control keyword parameter.
FATAL	I/O CONTROL KEYWORD PARAMETER .... MUST BE TYPE	I/O control keyword parameter is wrong type.	Correct I/O control keyword parameter type.
FATAL	ILL-FORMED COMPLEX CONSTANT	Complex constant invalid.	Correct complex constant.
FATAL	ILLEGAL BLOCK IF STRUCTURE	ELSEIF, ELSE, or ENDIF appears, but is not associated with a block IF.	Check IF block nesting.
FATAL	ILLEGAL BLOCK NAME IN COMMON STATEMENT	Block name in COMMON statement illegal.	Correct block name.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	ILLEGAL BUFFER LENGTH FOR FILE ..... -- DEFINITION IGNORED	Buffer length invalid.	Redefine buffer length.
FATAL	ILLEGAL CHARACTER COUNT	Character count for ENCODE or DECODE must be integer constant or simple integer variable.	Correct character count.
FATAL	ILLEGAL CONSTANT FOLLOWING + OR -	+ or - is followed by an illegal constant.	Correct illegal constant.
FATAL	ILLEGAL EXPLICIT LEVEL DECLARATION FOR COMMON MEMBER NAME	Explicit level declaration for a common member name is illegal.	Correct explicit level declaration.
FATAL	ILLEGAL FORM INVOLVING THE USE OF A COMMA	Parenthesized form with comma(s) in error. May be badly formed complex constant or I/O list with redundant parentheses.	Correct use of comma.
FATAL	ILLEGAL FORM OF EXPONENT	Exponent is invalid.	Correct form of exponent.
FATAL	ILLEGAL FORMAT SPECIFIER	Format specifier must be a legal statement label.	Correct format specifier.
FATAL	ILLEGAL IF BLOCK NESTING WITH DO LOOP	Range of the IF block must be within the range of the DO loop.	Make range of IF block within range of DO loop.
FATAL	ILLEGAL IF STATEMENT -- OBJECT MISSING	End of statement encountered before finding object of IF.	Correct the IF statement.
FATAL	ILLEGAL NESTING OF DO LOOPS	The range of an inner DO must be within the range of an outer DO.	Restructure DO loops.
FATAL	ILLEGAL OBJECT OF IF -- TROUBLE STARTED AT	Object of IF illegal.	Correct object of IF.
FATAL	ILLEGAL OBJECT OF LOGICAL IF	Improper statement type, used as true part of a logical IF. The object must be an executable statement. It cannot be a logical IF, DO, block IF, ELSEIF, ENDIF, ELSE, or END.	Correct object of logical IF.
FATAL	ILLEGAL RANGE -- ..... NOT LESS THAN ..... -- TRUNCATED	Range is illegal.	Correct range.
FATAL	ILLEGAL RECORD LENGTH FOR FILE ..... -- DEFINITION IGNORED	Record length invalid.	Redefine record length of file.
FATAL	ILLEGAL REFERENCE TO LABEL ..... DEFINED ON NON-EXECUTABLE STATEMENT	The label specifies a non-executable statement.	Correct reference to label.
FATAL	ILLEGAL REFERENCE TO STATEMENT LABEL ..... AS A FORMAT	The label referencing a FORMAT statement appears on an executable statement.	Correct reference to statement label.
FATAL	ILLEGAL REPEAT CONSTANT	Error in DATA statement.	Correct DATA statement.
FATAL	ILLEGAL SEPARATOR FOLLOWING DATA CONSTANT	The legal separators are), /, or . .	Replace with legal separator.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	ILLEGAL TRANSFER INTO RANGE OF DO	The indicated statement branches into a DO loop.	Check transfer into DO loop range.
FATAL	ILLEGAL TRANSFER TO ..... FORMAT	FORMAT statements cannot be the objects of transfers.	Correct illegal transfer.
FATAL	ILLEGAL USE OF ASSIGNMENT OPERATOR	Equal sign used improperly.	Correct use of equal sign.
FATAL	ILLEGAL USE OF ENTRY	Entry name used improperly.	Correct use of entry name.
FATAL	ILLEGAL USE OF NAMELIST GROUP NAME	Use of NAMELIST group name is invalid.	Correct use of NAMELIST group name.
FATAL	ILLEGAL USE OF OPERATOR/OPERAND --	Use of operator/operand is invalid.	Correct use of operator/operand.
FATAL	ILLEGAL USE OF PARAMETER	Use of parameter is invalid.	Use valid parameter.
FATAL	IMPLICIT MUST BE FOLLOWED BY A TYPE INDICATOR	Type information omitted.	Provide a type keyword, such as INTEGER or REAL.
FATAL	IMPLICIT STATEMENT MUST OCCUR BEFORE DECLARATIVE STATEMENTS	IMPLICIT must be the first statement after the PROGRAM statement.	Move the IMPLICIT statement.
FATAL	IMPLIED LOOP NOT TERMINATED	Implied loop must be terminated.	Check statement for syntax errors.
FATAL	IMPLIED I/O UNIT SPECIFIER NOT ALLOWED FOR THIS STATEMENT	Unit specifier must be explicit.	Explicitly specify I/O unit specifier.
FATAL	INITIAL LEFT PAREN MISSING	The initial left parenthesis is missing.	Provide left parenthesis.
FATAL	INQUIRE CANNOT SPECIFY BOTH UNIT AND FILE	Either a file name or a unit specifier must be specified in an INQUIRE statement.	Specify either unit or file.
FATAL	INQUIRE MUST SPECIFY UNIT OR FILE	INQUIRE statement is required to specify a file name or a unit specifier.	Specify either unit or file.
FATAL	INTEGER 0, 1, 2 OR 3 MUST FOLLOW LEVEL	0, 1, 2 or 3 are required to follow LEVEL in a LEVEL statement.	Correct LEVEL statement.
FATAL	INTERNAL FILE I/O CANNOT BE NAMELIST	Interval file I/O must not be NAMELIST.	Check NAMELIST.
FATAL	INTERNAL FILE REQUIRES A FORMAT	The internal file must have a format.	Provide format for internal file.
FATAL	INTERNAL FILE WITHOUT FORMAT OR MISSING COMMA BEFORE I/O LIST	Internal file must have format or comma missing before I/O list.	Provide format for internal file or place comma before I/O list.
FATAL	INTERNAL UNIT SPECIFIER CANNOT BE	Illegal use of internal unit specifier.	Correct illegal use.
FATAL	INTERNAL UNIT SPECIFIER CANNOT BE ASSUMED SIZE ARRAY	Internal unit specifier must not be assumed size array.	Specify size array.
FATAL	INTERNAL UNIT SPECIFIER NOT ALLOWED FOR THIS STATEMENT	Internal unit specifier invalid in this context.	Check statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	INTRINSIC FUNCTION ..... NOT ALLOWED AS ACTUAL ARGUMENT	INTRINSIC function is not allowed as actual argument.	Remove intrinsic function name from argument list.
FATAL	INTRINSIC LEN MUST NOT APPEAR IN PARAMETER CONSTANT EXPRESSION	LEN intrinsic appears in PARAMETER statement.	Rewrite statement.
FATAL	INVALID STATEMENT LABEL	The statement label is invalid.	Correct statement label.
FATAL	LEFT SIDE OF EQUAL SIGN IS ILLEGAL	Left side of equal sign illegal.	Correct left side of equal sign.
FATAL	LENGTH OF CHARACTER FORMAT SPECIFIER MUST BE GREATER THAN 1	The length of the character format specifier must be greater than 1.	Correct length of character format specifier.
FATAL	LEVEL 3 NAME ..... MAY NOT OCCUR IN THIS STATEMENT	Level 3 data cannot be used in expressions.	Correct use of level 3 data.
FATAL	LOCF ARGUMENT MUST NOT BE	LOCF argument must be a variable.	Make LOCF a variable.
FATAL	LOGICAL IF EXPRESSION MUST BE LOGICAL	Logical IF expression is required to be logical.	Make logical IF expression logical.
FATAL	LOGICAL IF MUST NOT BE OBJECT OF LOGICAL IF	Logical IF cannot be object of logical IF.	Correct object of logical IF.
FATAL	MAGNITUDE OF SUBSCRIPT ..... OF ..... EXCEEDS 2**23-1	Subscript too large or too small.	Correct subscript.
FATAL	MISSING COMMA AT	Comma is missing in statement.	Provide comma in proper place.
FATAL	MISSING LEFT PAREN AT	Left paren is missing in statement.	Provide left paren in proper place.
FATAL	MISSING NAME IN LEVEL LIST	Name missing in LEVEL list.	Insert missing name.
FATAL	MISSING SLASH ON GROUP NAME	Group name must be enclosed by slashes.	Provide slashes on group name.
FATAL	MISSING SUBSCRIPTS SET TO LOWER BOUND FOR EQUIVALENCE VARIABLE	EQUIVALENCE variable contains fewer subscripts than declared dimension.	Check declaration of the EQUIVALENCE variable.
FATAL	MORE THAN 7 SUBSCRIPTS	Too many subscripts.	Reduce number of subscripts.
FATAL	MULITPLE DECIMAL POINT IN EDIT DESCRIPTOR AT	Too many decimal points.	Remove extra decimal points.
FATAL	MULTIPLE DEFINITION OF CURRENT FORMAT LABEL	Format label previously defined.	Check FORMAT statements for duplicate labels.
FATAL	MULTIPLE OCCURRENCES OF DUMMY ARGUMENT	Dummy argument occurs more than once in dummy-arg list.	Remove multiple occurrences of dummy arguments.
FATAL	MULTIPLY DEFINED STATEMENT LABEL	The same statement label appears on more than one statement.	Change duplicate labels.
FATAL	NAME EXCEEDS 7 CHARACTERS -- TRUNCATED TO	Names must be unique within 7 characters.	Shorten name.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	NAME ..... IN DATA CONSTANT LIST MUST BE PARAMETER	Name must be parameter.	Remove name that is not a parameter.
FATAL	NESTING OF REPEAT COUNT IN DATA CONSTANT LIST IS ILLEGAL	Nesting of repeat count in data constant list is not allowed.	Remove nesting of repeat count.
FATAL	NO DIMENSION FOUND FOR EQUIVALENCE VARIABLE	Dimension of equivalence variable missing.	Supply dimension of equivalence variable.
FATAL	NO PREVIOUS C$ IF DIRECTIVE	C$ ELSE or ENDIF must be preceded by a C$ IF.	Provide C$ IF directive.
FATAL	NON-DUMMY ARGUMENT ..... CANNOT BE LEVELED	Leveled name must be a dummy-arg or in common.	Add name to argument list or to a common block.
FATAL	NON-NULL LABEL FIELD ON CONTINUATION LINE	Columns 1 through 5 are not on continuation line.	Remove extraneous label.
FATAL	OBJECT OF GO TO MISSING	The GO TO does not specify an existing statement label.	Provide statement label or change object of GO TO.
FATAL	OBJECT OF GO TO ..... DID NOT APPEAR IN ASSIGN STATEMENT	Object of GO TO must appear in ASSIGN statement.	Put object of GO TO in ASSIGN statement.
FATAL	ONLY ONE C$ ELSE ALLOWED IN C$ IF GROUP	More than one C$ ELSE in C$ IF group.	Remove excess C$ ELSE from C$ IF group.
FATAL	ONLY 9 PAREN LEVELS ALLOWED	Too many parenthesis levels in FORMAT statement.	Reduce number of parenthesis levels.
FATAL	ONLY 19 CONTINUATION LINES ARE PERMITTED	Too many continuation lines.	Reduce number of continuation lines.
FATAL	ONLY 500 DUMMY ARGUMENTS ARE PERMITTED -- EXCESS IGNORED	Total number of unique dummy arguments in the FUNCTION or SUBROUTINE statement and in all associated ENTRY statements exceed the allowed number.	Reduce number of dummy arguments.
FATAL	ONLY 500 COMMON BLOCKS ARE PERMITTED	Too many common blocks.	Reduce number of common blocks.
FATAL	ONLY LIST DIRECTED OUTPUT STATEMENTS MAY END WITH A COMMA	Extraneous comma found.	Remove comma.
FATAL	OPERAND HAS MODE NOT ALLOWED IN THIS CONTEXT	Wrong mode for this situation.	Correct mode.
FATAL	OPERAND OF // OPERATOR MUST BE TYPE CHARACTER	Operand is required to be type character.	Declare operand to be of type character.
FATAL	OPERAND OF ** OPERATOR MUST NOT BE TYPE CHARACTER	Exponentiation cannot be performed using character operands.	Correct operand type.
FATAL	OPERAND TO ** OPERATOR MUST NOT BE LOGICAL	Exponentiation cannot be performed using logical operands.	Correct operand type.
FATAL	OVCAP DIRECTIVE CAN APPEAR ONLY WITH SUBROUTINES HAVING NO ARGUMENTS	OVCAP directives can only appear with subroutines having no arguments.	Rewrite program.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	OVERLAY DIRECTIVE MUST BEGIN WITH LEFT PAREN	OVERLAY directives must begin with left parenthesis.	Add left parenthesis.
FATAL	PARAMETER REQUIRES INTEGER EXPONENTIATION	Integer exponentiation is required with this parameter.	Provide integer exponentiation for this parameter.
FATAL	PARAMETER ..... TYPE OR CHARACTER LENGTH CANNOT BE MODIFIED AFTER PARAMETER STATEMENT	Length of symbolic constant must not be changed by an IMPLICIT statement or other statements following a PARAMETER statement.	Correct usage of symbolic constant.
FATAL	PREMATURE E.O.S.	Premature end of statement.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. -- EXPECTED BLOCK NAME	End of statement encountered before a block name was found.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. -- EXPECTED SYMBOL OR SLASH	End of statement encountered before a symbol or slash was found.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN ENCODE OR DECODE	End of statement encountered; ENCODE or DECODE statement incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN I/O CONTROL LIST	End of statement encountered; I/O control list incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN I/O LIST ITEM SUBSCRIPT	End of statement encountered; I/O list item subscript incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. OR MISSING RIGHT PAREN	End of statement encountered or right parenthesis missing.	Check for incomplete statement.
FATAL	PREVIOUS REFERENCE TO DO LABEL ..... IS ILLEGAL	A DO label must not be referenced from outside the DO loop.	Check all previous references to the label.
FATAL	PREVIOUS REFERENCE TO FORMAT LABEL ..... IS ILLEGAL	The label was previously defined or referenced as a FORMAT label.	Check all previous references to the label.
FATAL	PREVIOUS REFERENCE TO LABEL ..... WAS ILLEGAL	Illegal reference to label.	Correct reference to the label.
FATAL	PROGRAM LENGTH EXCEEDS 2**17-1	Program too large.	Shorten program or break up into several routines.
FATAL	RECORD LENGTH EXCEEDS 131071 COLUMNS	Record too large. Error in FORMAT statement.	Check for incorrect repeat specification, hollerith count, and format specification.
FATAL	RECORD LENGTH FOR FILE ..... EXCEEDS 'MAX. RECL' B -- DEFINITION IGNORED	Record length too large.	Reduce record length.
FATAL	RECURSIVE DEFINITION OF STATEMENT FUNCTION	The function name appears on both sides of an equal sign.	Remove function name from the right side of the equal sign.
FATAL	REFERENCE TO EXTERNAL ..... REQUIRES AN ARGUMENT LIST	Function requires argument list.	Supply appropriate argument list.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	REFERENCE TO VARIABLE ..... AS A FUNCTION OR ARRAY	The variable has a subscript or argument list, but is not declared as an array or function.	Check for missing declaration.
FATAL	REPEAT COUNT IS NOT ALLOWED BEFORE THE EDIT DESCRIPTOR	A repeat count was used with a descriptor that does not allow one.	Remove repeat count.
FATAL	SCALAR FORMAT SPECIFIER MUST BE INTEGER	Scalar format is required to be integer.	Make scalar format integer.
FATAL	'SCM' COMMON BLOCKLENGTH EXCEEDS 131071	Common block too large.	Break common block into two or more common blocks.
FATAL	SEPARATOR MISSING AT	Error in FORMAT statement.	Correct FORMAT statement.
FATAL	SEQUENCE NUMBER OUT OF ORDER	Sequence number was specified out of order.	Correct statement sequence number.
FATAL	SIGNED COUNT ALLOWED ONLY BEFORE P EDIT DESCRIPTOR	Signed count used illegally.	Correct use of signed count.
FATAL	SIZE OF ARRAY ..... EXCEEDS 1048568	Array too large.	Reduce size of array.
FATAL	SIZE OF ARRAY ..... EXCEEDS 131071	Array too large.	Reduce size of array.
FATAL	SLASH MUST BE FOLLOWED BY AN OCTAL OR INTEGER CONSTANT	Octal or integer constant missing after slash.	Put octal or integer after slash.
FATAL	STAR DUMMY ARGUMENT ILLEGAL IN FUNCTION	Alternate returns illegal in functions.	Remove alternate returns.
FATAL	STATEMENT FUNCTION ..... -- MISPLACED EQUAL SIGN	Syntax error in statement function.	Correct syntax error in statement function.
FATAL	STATEMENT FUNCTION ..... INDIRECTLY REFERENCES ITSELF	Recursive statement functions are not allowed.	Check all appropriate statement functions for indirect recursion.
FATAL	STATEMENT FUNCTION DEFINITION MUST OCCUR BEFORE FIRST EXECUTABLE	Definition must precede first executable statement.	Move statement function definition, or check for undimensioned array.
FATAL	STATEMENT FUNCTION DUMMY ARGUMENT ..... CANNOT BE ASSUMED LENGTH	Dummy argument name appeared in a CHARACTER*(*) declaration.	Change type declaration or dummy argument name.
FATAL	STATEMENT FUNCTION DUMMY ARGUMENT ..... MUST BE USED AS SIMPLE VARIABLE	Dummy argument was followed by expression in parentheses.	Rewrite statement function expression.
FATAL	STATEMENT FUNCTION DUMMY PARAMETER ..... NOT SIMPLE VARIABLE	A constant or expression appears in the parameter list of a function definition.	Check parameter list of the function definition.
FATAL	STATEMENT FUNCTION INVALID IN DATA VARIABLE LIST	Attempt to use statement function that is in DATA statement.	Rewrite statement.
FATAL	STATEMENT LABEL ..... CONTAINS NON-DIGIT	Statement labels must consist of digits.	Correct statement labels.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	STATEMENT LABEL ..... EXCEEDS 5 DIGITS	Statement labels must be five digits or less.	Correct statement labels.
FATAL	STATEMENT LABEL ..... MUST BE NUMERIC	Statement labels must consist of digits.	Correct statement labels.
FATAL	STATEMENT LABEL ..... REFERENCED BUT NOT DEFINED	The indicated label does not appear as a statement label anywhere in the program.	Check for missing statement.
FATAL	STATEMENT LABEL EXPECTED BUT NOT FOUND	A statement label reference is missing.	Insert label.
FATAL	STATEMENT MISPLACED	Statement in the wrong place.	Put statement in proper place.
FATAL	STRING ADDRESS CANNOT BE	Invalid string address on encode or decode.	Check string address.
FATAL	STRING ADDRESS CANNOT BE CHARACTER	String address on encode or decode cannot be type character.	Change type of string address.
FATAL	SUBROUTINE ENTRY ..... MAY NOT APPEAR IN A DECLARATIVE STATEMENT	Subroutine entry cannot appear in a declarative statement.	Check declarative statement.
FATAL	SUBSCRIPT ..... OF ..... IS NOT A NUMERIC TYPE	Subscripts must be numeric.	Make subscripts numeric.
FATAL	SUBSCRIPTS IN DATA MUST BE INTEGER	Subscripts must be integer.	Make subscripts integer.
FATAL	SUBSTRING EXPRESSION NOT INTEGER	Substring expression must be integer.	Check substring expression.
FATAL	SUBSTRING ILLEGAL FOR OPERAND	Wrong substring for operand.	Check substring.
FATAL	SUBSTRING ILLEGAL FOR PARAMETER	Wrong substring for parameter.	Check substring.
FATAL	SUBSTRINGED VARIABLE ..... NOT TYPE CHARACTER	Variable must be character type.	Check substring variable.
FATAL	SYNTAX ERROR IN BLOCK NAME	Wrong syntax in block name.	Check block name for syntax error.
FATAL	SYNTAX ERROR IN DATA CONSTANT LIST	Wrong syntax in data constant list.	Check data constant list for syntax error.
FATAL	SYNTAX ERROR IN DATA STATEMENT	Wrong syntax in data statement.	Check data statement for syntax error.
FATAL	SYNTAX ERROR IN DIMENSION DECLARATION	Wrong syntax in dimension declaration.	Check dimension declaration for syntax error.
FATAL	SYNTAX ERROR IN EQUIVALENCE STATEMENT	Wrong syntax in EQUIVALENCE statement.	Check EQUIVALENCE statement for syntax error.
FATAL	SYNTAX ERROR IN GO TO STATEMENT	Wrong syntax in GO TO statement.	Check GO TO statement for syntax error.
FATAL	SYNTAX ERROR IN I/O CONTROL LIST AT	Wrong syntax in I/O control list.	Check I/O control list for syntax error.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	SYNTAX ERROR IN I/O IMPLIED DO	Wrong syntax in I/O implied DO.	Check I/O implied DO for syntax error.
FATAL	SYNTAX ERROR IN NAMELIST	Wrong syntax in NAMELIST.	Check NAMELIST for syntax error.
FATAL	SYNTAX ERROR IN PROGRAM UNIT NAME	Wrong syntax in program unit name.	Check program unit name for syntax error.
FATAL	SYNTAX ERROR IN SUBSTRING EXPRESSION FOR	Wrong syntax in substring expression.	Check substring expression for syntax error.
FATAL	SYNTAX OF DO MUST BE I=M1,M2,M3 OR M1,M2	DO statement syntax incorrect.	Use correct syntax.
FATAL	T EDIT DESCRIPTOR FOLLOWED BY ZERO OR NON-DIGIT	T code must be followed by nonzero column number.	Correct column number.
FATAL	TABLE OVERFLOW -- INCREASE FIELD LENGTH AND RERUN	Not enough field length for compilation.	Provide more field length for compilation.
FATAL	TERMINAL DELIMITER ..... MISSING	The terminal delimiter is missing.	Provide correct terminal delimiter.
FATAL	THE TERMINAL STATEMENT OF DO ..... PRECEDED THE DO DEFINITION	Terminal statement of DO must not precede the DO definition.	Provide terminal statement of DO in proper place.
FATAL	THIS IS NOT A FORTRAN STATEMENT	Unrecognizable statement.	Check syntax.
FATAL	THIS STATEMENT MAY NOT BE A DO TERMINAL	A DO loop cannot end with the specified statement.	Restructure DO loop.
FATAL	THIS STATEMENT MUST BE CONTAINED ON 1 CARD	Continuation lines not allowed for this statement.	Rewrite statement to fit on 1 line.
FATAL	TOO FEW LEFT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT	Left paren missing or there is a previous syntax error.	Check parenthesis matching or correct previous syntax.
FATAL	TOO FEW RIGHT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT	Right paren missing or there is a previous syntax error.	Check parenthesis matching or correct previous syntax.
FATAL	TRIP COUNT IS LESS THAN ONE	Trip count must be at least one if DO=0T is selected.	Make trip count at least one.
FATAL	TRIP COUNT OF ..... MUST BE POSITIVE	Trip count must be positive.	Make trip count positive.
FATAL	TRIP COUNT TOO HIGH -- SHORT LOOPS SELECTED	Trip count too high.	Lower trip count.
FATAL	UNBALANCED PARENS	Parentheses are unbalanced.	Balance parentheses.
FATAL	UNDECLARED INTRINSIC OR EXTERNAL FUNCTION ..... USED AS ACTUAL ARGUMENT	Cannot use undeclared function as actual argument.	Remove undeclared function.
FATAL	UNIT SPECIFIER FILE NAME GREATER THAN 7 CHARACTERS	Illegal file name.	Check character length of unit specifier.
FATAL	UNIT SPECIFIER NOT LEGAL FILE NAME	Illegal file name.	Check all uses of the file name.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
FATAL	UNIT SPECIFIER OUTSIDE RANGE 0 - 999	Illegal unit number.	Provide a unit number which is no more than 3 digits long.
FATAL	UNIT SPECIFIER MISSING	Unit specifier required.	Provide a unit number.
FATAL	UNKNOWN EDIT DESCRIPTOR	EDIT descriptor not recognized.	Check EDIT descriptor.
FATAL	UNMATCHED PARAMETER COUNT TO STATEMENT FUNCTION	The function reference and function definition contain different numbers of parameters.	Check for missing parameters.
FATAL	USAGE CONFLICT -- ..... CANNOT BE STATEMENT FUNCTION	The indicated statement function conflicts with a previous usage.	Check all other usages; the function name might be used as a variable or array name.
FATAL	USAGE CONFLICT -- ..... IS ..... AND CANNOT BE	Usage conflict.	Check uses of indicated name.
FATAL	USAGE CONFLICT ..... PREVIOUSLY USED AS	The label was previously used another way.	Check previous usage of label.
FATAL	USAGE CONFLICT -- ..... PREVOUSLY DEFINED AS DO TERMINAL	The label was previously defined as a DO terminal.	Check previous loops for use of the same label.
FATAL	USAGE CONFLICT -- ..... PREVIOUSLY DEFINED AS FORMAT	The label was previously defined as a FORMAT label.	Change label.
FATAL	USAGE CONFLICT -- ..... PREVIOUSLY USED AS A FORMAT LABEL	The label was previously used as a Format label.	Change label.
FATAL	ZERO IS SPECIFIED AS REPEAT COUNT AT	Repeat count must be greater than zero.	Make repeat count greater than zero.
FATAL	ZERO LENGTH CHARACTER OR HOLLERITH STRING	Character or hollerith string must have a positive nonzero length.	Make string positive nonzero length.
MDEP	BOOLEAN DATA TYPE IS MACHINE DEPENDENT	This data type is machine dependent.	Use the CHARACTER data type instead, for portability.
MDEP	BUFFER I/O IS MACHINE DEPENDENT	Buffer I/O is machine dependent.	Avoid using Buffer I/O if possible, especially usages that depend on the number of characters per word.
MDEP	ENCODE/DECODE ARE MACHINE DEPENDENT	ENCODE/DECODE is machine dependent.	Use internal files instead for portability.
MDEP	LIBRARY FUNCTIONS DATE, TIME AND CLOCK ARE MACHINE DEPENDENT	These functions are machine dependent.	Do not dismantle the output of these routines, print them out as a whole.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
MDEP	OVCAPS ARE MACHINE DEPENDENT	OVCAPS are machine dependent.	Do not let programs depend on certain properties of OVCAPS, such as reinitialization of variables when an OVCAP is reloaded.
MDEP	OVERLAYS ARE MACHINE DEPENDENT	OVERLAYS are machine dependent.	Do not let programs depend on certain properties of overlays, such as reinitialization of variables when an overlay is reloaded.
TRIVIAL	ARGUMENT ..... IS NOT USED IN STATEMENT FUNCTION	Specified argument not needed.	Remove argument.
TRIVIAL	CONSTANT ** CONSTANT CANNOT BE EVALUATED	Specified operation cannot be performed at compile time.	Change the expression.
TRIVIAL	CONSTANT TOO LONG, EXCESS DIGITS TRUNCATED	Constant truncated due to excess length.	Remove excess digits.
TRIVIAL	CONTINUE WITH NO STATEMENT LABEL -- IGNORED	CONTINUE without statement label is meaningless.	Insert label or eliminate CONTINUE.
TRIVIAL	IF RESULTS IN A SIMPLE TRANSFER	The IF can be replaced by a GO TO.	Make the substitution.
TRIVIAL	IF RESULTS IN A TRANSFER TO THE NEXT LINE	Control will always transfer to the next statement, regardless of the condition specified in the IF statement.	Reexamine the IF.
TRIVIAL	INTEGER ** NEGATIVE CONSTANT -- RESULTS ZERO	Integer raised to a negative power is zero.	Change the integer to real.
TRIVIAL	LAST IF RESULTS IN A NULL TRANSFER TO THIS STATEMENT	IF acts as a do-nothing statement.	Check syntax of IF.
TRIVIAL	MISSING PROGRAM STATEMENT -- PROGRAM START. ASSUMED	The PROGRAM statement must be the first statement of the main program.	Supply PROGRAM statement.
TRIVIAL	NO PATH TO THE ENTIRE RANGE OF DO	The statements in the loop cannot be reached.	Check for logic error, in current branch.
TRIVIAL	NO PATH TO THIS STATEMENT	Statement will never be executed.	Check program logic, particularly GO TO statements and IF statements.
TRIVIAL	NULL TRANSFER STATEMENT -- TRANSFER IGNORED	A GO TO statement transfers to the next statement.	GO TO can be eliminated.
TRIVIAL	RECORD LENGTH EXCEEDS 137 COLUMNS -- MAY EXCEED I/O DEVICE	Record length may be too large for peripheral device.	Reduce record length if necessary.
TRIVIAL	STATEMENT CAN TRANSFER TO ITSELF	Infinite loop possible.	Revise statement.
TRIVIAL	STATEMENT TRANSFERS TO ITSELF	Infinite loop results.	Change statement.
TRIVIAL	THIS DO LOOP WILL NOT EXECUTE	Condition always prohibits execution of DO loop.	Check logic of DO loop.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
TRIVIAL	TL EDIT DESCRIPTOR BACKSPACED BEYOND 1st COLUMN -- COLUMN POINTER RESET AT 1	Value of TL code is too large.	Check TL code.
TRIVIAL	VARIABLE ** ZERO -- RESULT ASSUMED ONE	Variable raised to zero power is equal to one.	Check expression.
TRIVIAL	ZERO ** ZERO -- RESULTS INDEFINITE	Zero raised to zero power is indefinite.	Check expression.
WARNING	*TO* ASSUMED FOR	Syntax error in ASSIGN statement.	Check ASSIGN statement for syntax error.
WARNING	..... PREVIOUSLY DECLARED INTRINSIC -- IGNORED	Function already declared.	Check declaration of functions.
WARNING	..... PREVIOUSLY DECLARED EXTERNAL -- IGNORED	Function already declared.	Check declaration of functions.
WARNING	..... PREVIOUSLY TYPED NON-CONFORMING -- PREVIOUS TYPE OVERRIDDEN	Most recent declaration used.	Check declarations.
WARNING	..... REDUNDANTLY DECLARED IN SAVE	The indicated name appears more than once in a SAVE statement.	Eliminate redundancy.
WARNING	ARGUMENT TO MASK MUST BE BETWEEN 0 AND 60	Argument to mask is not between 0 and 60.	Make argument to mask between 0 and 60.
WARNING	C$ PARAMETER VALUE FOR ..... ON ..... MUST BE 0 OR 1	C$ parameter must be zero or one.	Check C$ parameter.
WARNING	COMMA AFTER STATEMENT LABEL IGNORED	Comma is not needed.	Remove comma.
WARNING	COMMA MUST FOLLOW LEVEL NUMBER	Comma missing after level number.	Insert comma.
WARNING	CONFLICT IN RANGE INDICATOR -- FIRST RETAINED	Overlap of ranges in IMPLICIT statement.	Check for overlap of ranges in IMPLICIT statement.
WARNING	CONSTANT EXCEEDS 5 DIGITS -- TRUNCATED	Constant too long.	Reduce number of digits in constant to 5 or less.
WARNING	CONSTANT MISSING EXPONENT FIELD -- ZERO ASSUMED	Exponent field missing in constant; zero assumed.	Provide constant with an exponent field.
WARNING	DO CONCLUSION NOT COMPILED -- DO DEFINITION ERROR	Error in DO definition; DO conclusion not compiled.	Correct previous errors.
WARNING	ENTRY ..... MUST NOT BE DECLARED EXTERNAL -- IGNORED	The entry must not be declared external.	Correct declaration of entry.
WARNING	ENTRY STATEMENT IGNORED IN MAIN PROGRAM	An ENTRY statement in the main program has no purpose.	Remove ENTRY statement.
WARNING	EXCESS CONSTANTS IGNORED	Too many constants.	Reduce excess number of constants.
WARNING	EXPECTED COMMA AFTER I/O CONTROL -- FOUND	Comma should have followed I/O control statement.	Provide comma after I/O control statement.
WARNING	EXPECTED E.O.S. -- FOUND AND IGNORED	Extraneous information follows a legal statement.	Remove extra characters.
WARNING	EXPECTED LEFT PAREN -- FOUND	Left parenthesis not found.	Check syntax of statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message	Significance	Action
WARNING EXTRANEOUS COMMA IGNORED	Comma unrecognized and ignored.	Remove extraneous comma.
WARNING FIELD WIDTH IS LESS THAN MINIMUM REQUIRED ON EDIT DESCRIPTOR AT	Field width too small.	Increase field width.
WARNING FUNCTION ..... REFERENCED AS SUBROUTINE	Function names must not be the object of CALL statements.	Use function reference syntax.
WARNING FWA AND LWA NOT IN SAME ARRAY, EQUIVALENCE CLASS, OR COMMON BLOCK	First-word-address and last-word-address must be in the same common block, equivalence class, or array.	Check declarations for inconsistencies involving FWA and LWA.
WARNING GENERIC ONLY INTRINSIC .... TYPED--TYPING IGNORED	Name of intrinsic function, which is not specific function, appears in type.	Remove attempted typing.
WARNING HOLLERITH CONSTANT EXCEEDS 10 CHARACTERS	Self-explanatory.	Reduce constant to 10 characters or less.
WARNING I/O LIST IGNORED WHEN USING NAMELIST	Namelist I/O does not use an I/O list.	Eliminate I/O list.
WARNING ILLEGAL NAME -- ENTRY STATEMENT IGNORED	Name invalid.	Provide legal name.
WARNING INTRINSIC ..... TYPED NON-CONFORMING -- TYPE IGNORED	Declared type of intrinsic nonconforming.	Change type declaration.
WARNING LOCAL ..... IN BLOCK DATA -- IGNORED	Variable appears in BLOCK DATA subprogram, but not in a common statement.	Check common block for missing variables.
WARNING MISSING NAME -- ENTRY STATEMENT IGNORED	ENTRY statement needs a name.	Provide name for ENTRY statement.
WARNING MULTIPLY DEFINED LEVEL FOR NAME ..... -- IGNORED	Too many levels defined for name.	Check defined levels of name.
WARNING MULTIPLY DEFINED LEVEL FOR COMMON BLOCK NAME ..... -- IGNORED	Too many levels defined for common block name.	Check defined levels of common block name.
WARNING NAME ..... PREVIOUSLY DEFINED -- ENTRY STATEMENT IGNORED	Too many definitions of ENTRY name.	Check for another usage of the ENTRY name.
WARNING NON-OCTAL DIGIT IN OCTAL CONSTANT -- IGNORED	Digit must be less than or equal to 7.	Rewrite octal constant.
WARNING NUMBER OF ARGUMENTS IN REFERENCE TO ..... IS NOT CONSISTENT	Number of arguments in reference must be the same as the number of arguments in the FUNCTION or SUBROUTINE statement.	Check arguments.
WARNING OBJECT OF GO TO NOT INTEGER VARIABLE	Object of GO TO must be a simple integer variable.	Make object of GO TO integer variable.
WARNING ONLY 49 FILES ARE ALLOWED -- EXCESS IGNORED	Too many files were specified in the PROGRAM statement.	Reduce number of excess files.
WARNING PREMATURE E.O.S. -- EXPECTED VARIABLE AT	End of statement encountered; statement incomplete.	Check syntax.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

Message		Significance	Action
WARNING	PREMATURE E.O.S. OR EXTRA TRAILING SEPARATOR	End of statement encountered or an extraneous separator found.	Check statement or eliminate extra separator.
WARNING	PREVIOUS DEFINITION OF STATEMENT FUNCTION ..... IS OVERRIDDEN	The function was defined more than once; the most recent definition is used.	Change second definition.
WARNING	RANGE INDICATOR ..... NOT 1 LETTER -- TRUNCATED TO	Implicit statement range indicator not 1 letter.	Change the range indicator to 1 letter.
WARNING	REDUNDANT EQUIVALENCE SPECIFICATION FOR	EQUIVALENCE specification used before.	Check for occurrence of indicated symbol in previous EQUIVALENCE statement.
WARNING	SHIFT COUNT MUST BE BETWEEN -60 and 60	SHIFT count is not between -60 and 60.	Make SHIFT count between -60 and 60.
WARNING	STATEMENT FUNCTION ..... HAS NULL DEFINITION -- IGNORED	Statement function expansion reduces to a null code sequence.	Check for error in function definition statement.
WARNING	SUBROUTINE ..... REFERENCED AS FUNCTION	Subroutines are referenced with the CALL statement.	Use CALL statement.
WARNING	SUBSCRIPT ..... OF ..... VIOLATES LOWER DIMENSION BOUND	Subscript less than declared lower bound.	Correct subscript.
WARNING	SUBSCRIPT ..... OF ..... VIOLATES UPPER DIMENSION BOUND	Subscript greater than declared upper bound.	Correct subscript.
WARNING	TERMINAL CHARACTER ..... CONVERTED TO RIGHT PAREN	The indicated character appeared where a right parenthesis was expected.	Compiler assumes a right parenthesis.
WARNING	THIS STATEMENT HAS NO INITIAL LINE -- INITIAL ASSUMED	Initial line missing from statement.	Provide initial line.
WARNING	TOO FEW CONSTANTS -- VARIABLES FROM ..... NOT INITIALIZED	Not enough constants in data constant list.	Initialize the variables; uninitalized variables can cause run-time errors.
WARNING	TRIVIAL EQUIVALENCE GROUP WITH ONLY 1 MEMBER IS IGNORED	An EQUIVALENCE must contain at least 2 members.	Check EQUIVALENCE statement.
WARNING	TRIVIAL RANGE -- ..... SAME AS	Implicit range is trivial.	Check range.
WARNING	TYPING OF ..... IGNORED -- PRIOR TYPING RETAINED	The symbol appeared in more than one type statement; first type is used.	Eliminate second type declaration.
WARNING	UNIVERSAL SAVE DECLARED -- OTHER SAVE STATEMENTS ARE REDUNDANT	When universal SAVE declared other SAVE statements are not necessary.	Eliminate redundant SAVE statements.
WARNING	UNKNOWN FORM -- BLANK ASSUMED	Unrecognizable form of STOP or PAUSE statement.	Check STOP or PAUSE statement.
WARNING	VARIABLE ..... HAS NO DIMENSION BOUND -- IGNORED	Variable label must have dimension bound.	Provide dimension bound for variable label.
WARNING	VARIABLE ..... NOT INTEGER	Variable must be integer.	Make variable integer.

TABLE B-2. SPECIAL COMPILATION DIAGNOSTICS

Message	Significance	Action
COMPILING program LAST STATEMENT BEGAN AT LINE nnnnn ERROR AT aaaaa IN ddddddd LAST OVERLAY LOADED - (p,s)	Compiler, operating system, or hardware error has occurred while compiling program. program Name of source program unit. nnnnn Approximate compiler-assigned source line number where the difficulty arose. During transitions from one phase of compilation to another, the END line number might be displayed. ddddddd Name of compiler internal deck where abort occurred. Might be RA+0 if control was accidentally transferred to the control point job communications area. aaaaaa Address relative to origin of internal deck where abort occurred. p,s Primary and secondary level numbers of overlay last loaded before abort occurred: 0,0 - Control statement cracker; global communication and control 1,0 - (OPT=0) compilation overlay 2,0 - OPT>0 compilation batch controller 2,1 - (OPT>0) compilation normal pass 1 (lexical scan, parse, intermediate language generation) 2,2 - (OPT>0) compilation pass 2 (global and local optimization, object code generation) 2,3 - (OPT>0) compilation reference map generation and object code assembly phase	Follow site-defined procedures for reporting software errors or operational problems.
DEAD CODE IN Program	A section of code is unreachable and cannot be processed (can be issued only when OPT≥2).	Same as STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED.
EMPTY INPUT FILE. NO COMPILATION.	An end-of-partition or end-of-section was encountered on the first read of the input.	Check for extra EOP or EOS, or mispositioned input file.

TABLE B-3. COMPILER OUTPUT LISTING MESSAGES

Message	Significance	Action
STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED.	Executable statements in the source program can never be executed, due to program flow of control. No object code is compiled for dead statements. Accompanied by dayfile message DEAD CODE IN program. Detected only when OPT=2 has been selected.	Check flow control of program.

TABLE B-4. EXECUTION-TIME DIAGNOSTICS

No.	Class		Message	Significance	Action	Issued By
1	F	A	ERROR IN COMPUTED GO TO STATEMENT - INDEX VALUE INVALID	Value .LT. 1 or .GT. number of statement numbers. Occurs only if FORTRAN Extended 4 binary is used in a FORTRAN 5 job.	Recompile using FORTRAN 5 compiler.	GOTOER=
2	I	A	ARGUMENT ABS VALUE .GT. 1 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ACOSIN=(ACOS)
3	I	A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ALOG
4	I	A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ALOG10
5	I	A	ARGUMENT ABS VALUE .GT. 1 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ACOSIN=(ASIN)
6	I	A	ARGUMENT INDEFINITE	Note 1	Note 2	ATAN
7	I	A	ARGUMENT VECTOR ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ATAN2
8	I	A	ARGUMENT TOO LARGE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	CABS
9	I	T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ZTOI (Z**I)
10	I	T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CCOS
11	I	T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CEXP
12	I	T	ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	CLOG
13	I	A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SINCOS=(COS)
14	I	T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CSIN
15	I	T	INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	CSQRT

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
16	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE BASE IN EXPONENTIATION INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOX (D**X)
17	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DATAN
18	I A	ARGUMENT VECTOR 0,0 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DATAN2
19	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOD (D**D)
20	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOI (D**I)
21	I T	FLOATING OVERFLOW IN D** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)LOG(D) TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOZ (D**Z)
22	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DCOS
23	I A	ARGUMENT TOO LARGE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DEXP
24	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DLOG
25	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DLOG10
26	I T	DP INTEGER EXCEEDS 96 BITS 2ND ARGUMENT ZERO INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DMOD
28	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DSIN
29	I T	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DSQRT
30	I A	ARGUMENT TOO LARGE, FLOATING OVERFLOW ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	EXP

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
31	I T	INTEGER OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER	Note 1	Note 2	ITOJ (I**J)
33	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOD (X**D)
34	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOI (X**I)
35	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOY (X**Y)
36	I A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SINCOS=(SIN)
39	I A	ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SQRT
40	I T	ILLEGAL SENSE SWITCH NUMBER	Number not in range 1-6; return parameter set to 2.		SSWTCH
41	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	TAN
42	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	TANH
44	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ITOD (I**D)
45	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ITOX (I**X)
46	I T	FLOATING OVERFLOW IN I** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(I) TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ITOZ (I**Z)

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
47	I T	FLOATING OVERFLOW IN X** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(X) TOO LARGE INFINITE OR INDEF ARGUMENT	Note 1	Note 2	XTOZ(X**Z)
49	I A I A I A	COMMA MISSING AT END OF RECORD - COMMA ASSUMED NAMELIST DATA TERMINATED BY EOF NOT $ CONSTANTS MISSING AT END OF RECORD - NEXT RECORD READ	Error occurred during NAMELIST processing.	Check NAMELIST input data for errors.	NAMIN=
50	F A	FATAL ERROR IN LOADER.	Error occurred during load.	Inspect load map to determine cause of error.	OVERLA=
51	I A	Set by user via subroutine SYSTEM or SYSTEMC.	Defined by user.	Defined by user.	USER
52	F A	Set by user via subroutine SYSTEM or SYSTEMC. Error numbers larger than any listed in this table become error 52.	Defined by user.	Defined by user.	USER
53	F A	NOT ENOUGH FL FOR SORT/MERGE.	More memory required for Sort/Merge processing.	Extend program field length.	SMXXXX=
55	F A	END-OF-FILE ENCOUNTERED, FILENAME - - - - xxxxxxx.	Attempt to read past end-of-file.	Rewind before reading or correct program logic.	BUFIN=
56	F A	WRITE FOLLOWED BY READ, FILENAME - - - - xxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	BUFIN=
57	F A	AREA SPECIFICATION SPANS SCM/LCM.	In a buffered I/O statement the first and last word addresses must be in the same level of memory.	Check word addresses in buffered I/O statement.	BUFIO=
58	F A	BUFFER DESIGNATION BAD -- FWA.GT.LWA.	First-word address must be LE last word address.	Check buffer designation.	BUFIO=
59	F A	BUFFER SPECIFICATION BAD -- FWA.GT.LWA	First-word address must be LE last word address.	Check first and last word address.	BUFOUT=
62	F A	INVALID UNIT	Unit not recognized.	Check unit number.	GETFIT=
63	F A	END-OF-FILE ENCOUNTERED ON FILE xxxxxxx.	Attempt to read past end-of-file.	Rewind file or correct program logic.	INPB=
65	F A	END-OF-FILE ENCOUNTERED ON FILE xxxxxxx.	Attempt to read past end-of-file.	Rewind file or correct program logic.	INPC= NAMIN=

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
66	F A F A F A F A F A F A	NAMELIST NAME NOT FOUND-xxxxxxx. INCORRECT SUBSCRIPT. TOO MANY CONSTANTS. , ($ OR = EXPECTED, MISSING. VARIABLE NAME NOT FOUND-xxxxxxx. CONSTANT MISSING.	Error occurred during NAMELIST processing.	Check NAMELIST input data for errors.	NAMIN=
67	F A	DECODE RECORD LENGTH .LE. 0. DECODE LCM RECORD .GT. 150 CHARACTERS.	Bad first parameter to DECODE.	Check first parameter to DECODE.	DECODE=
68	F A F A	* ILL-PLACED NUMBER OR SIGN. * ILLEGAL FUNCTIONAL LETTER.	Illegal FORMAT.	Check format.	FMTAP=
69	F A	* IMPROPER PARENTHESIS NESTING.	Illegal FORMAT.	Check format.	FMTAP=
70	F A	* EXCEEDED RECORD SIZE.	The maximum record length specified on the PROGRAM, OPEN, ENCODE, DECODE or FILE control statement has been exceeded.	Change RL parameter on PROGRAM statement, MRL parameter on the FILE control statement, RECL parameter on the OPEN statement, or C parameter on the ENCODE or DECODE statement.	FMTAP=
71	F A	* SPECIFIED FIELD WIDTH ZERO.	w=0 in FORMAT.	Check field width in FORMAT.	FMTAP=
72	F A	* FIELD WIDTH .LE. DECIMAL WIDTH.	w LE d in FORMAT.	Check width in FORMAT.	FMTAP=
73	F A	*HOLLERITH FORMAT WITH LIST.	The FORMAT has no specifiers corresponding to the I/O statement.	Change one or the other.	INCOM=
78	F A	* ILLEGAL DATA IN FIELD * ' *	Usually a nondigit in a numeric input field.	Fix input data.	INCOM=
79	F A	* DATA OVERFLOW * ' *	Input value GT 1.26501E322.	Fix input data.	INCOM=
83	F A	OUTPUT FILE LINE LIMIT EXCEEDED.	The default or specified print limit to OUTPUT was exceeded.	Specify PL on FTN5 statement, PL on execution call, or change program to print less.	OUTC= NAMOUT= OUTF= SYSERR=
85	F A	ENCODE CHARACTER/RECORD .LE. 0 ENCODE LCM RECORD .GT. 150 CHARACTERS	Bad first parameter to ENCODE.	Check first parameter to ENCODE.	ENCODE=
88	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	INPB=

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
89	F A	LIST EXCEEDS DATA, READ ON FILE-xxxxxxx.	More words were specified in the I/O list than existed in the record of the file.	Check for missing data or incorrect input list.	INPB=
90	F A	PARITY ERROR ON FILE xxxxxxx DURING PREVIOUS READ.	Probable disk or tape error.	See systems analyst.	INPB=
91	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	INPC=
92	F A	PARITY ERROR READING (CODED) FILE-xxxxxxx.	Probable disk or tape error.	See systems analyst.	INPC= NAMIN=
93	F A	PARITY ERROR ON FILE-xxxxxxx DURING PREVIOUS READ.	Probable disk or tape error.	See systems analyst.	OUTB=
94	F A	PARITY ERROR ON LAST READ ON FILE-xxxxxxx.	Probable disk or tape error.	See systems analyst.	OUTC=
95	F A	PARITY ERROR ON FILE xxxxxxx DURING PREVIOUS WRITE	Probable disk or tape error.	See systems analyst.	ODAB
96	F A	PARITY ERROR ON FILE xxxxxxx DURING PREVIOUS READ	Probable disk or tape error	See systems analyst.	IDAB
97	F A	INDEX NUMBER ERROR.	Nonexistent index value specified or bad file.	Check index and file.	RANMS=
98	F A	FILE ORGANIZATION ERR OR FILE NOT OPEN.		Call OPENMS.	RANMS=
99	F A	WRONG INDEX TYPE.	Wrong type specified to OPENMS.	Check index type.	RANMS=
100	F A	INDEX IS FULL.	An index is full, and an attempt is being made to add a new record to it.	Increase index size.	RANMS=
101	F A	DEFECTIVE INDEX CONTROL WORD.	Bad file.	File must be recreated.	RANMS=
102	F A	RECORD LENGTH EXCEEDS SPACE ALLOCATED.	Record length too long.	Increase space allocation.	RANMS= BUFIO=
103	F A	RECORD MANAGER ERROR xxx ON FILE xxxxxxx, RECORD xxxxxxx.	Record Manager error.	See Record Manager reference manual.	RANMS=
104	F A	INDEX KEY UNKNOWN.	Invalid index key.	Correct index key.	RANMS=
105	F A	RECORD LENGTH NEGATIVE.	Record length must not be negative.	Fix call.	RANMS=
107	F A	ILLEGAL PARAMETER VALUE.	Argument to Sort/ Merge routine has bad value.	Check parameter value of Sort/ Merge routine.	SMXXXX=
108	F A	TOO FEW OR TOO MANY PARAMETERS.	Valid number of parameters not provided.	Provide proper number of parameters.	SMXXXX=

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class		Message	Significance	Action	Issued By
109	F	A	KEYWORD (xxxxxxx) INVALID.	Keyword not recognized.	Provide legal keyword.	SMXXXX=
110	F	A	A ROUTINE CALLED OUT OF SEQUENCE.	Sequence (SMSORT, SMSORTB, SMSORTP, or SMMERGE), (other Sort/Merge calls), (SMEND or SMABT) not followed.	Check sequence of routine call.	SMXXXX=
111	F	A	LCM BLOCK COPY ERROR.	Parity error.	See systems analyst.	COMIO=, DECODE=, ENCODE=, INPB=, OUTB=, READEC, WRITEC
114	F	A	CONNEC CHARACTER CODE CONVERSION IS OUT OF RANGE	Bad second argument in CALL CONNEC.	Change to specify correct character set.	CONDIS=
115	I	A	ARGUMENT INFINITE ARGUMENT TOO SMALL	Note 1	Note 2	EXP
116	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	HYP=(COSH)
117	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	HYP=(SINH)
118	I	A	ARGUMENT TOO SMALL	Note 1	Note 2	DEXP
119	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DHYP=(DCOSH)
120	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DHYP=(DSINH)
121	I	A	ARGUMENT INDEFINITE	Note 1	Note 2	DTANH
122	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DTAN
123	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GT. 1.0.	Note 1	Note 2	DASNCS(DASIN)
124	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GT. 1.0	Note 1	Note 2	DASNCS(DACOS)
125	I	A	ARGUMENT INDEFINITE	Note 1	Note 2	ERF(ERF)
126	I	A	ARGUMENT INDEFINITE	Note 1	Note 2	ERF(ERFC)
127	I	A	ARGUMENT TOO LARGE	Note 1	Note 2	ERF(ERFC)
128	I	A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GE. 1.0.	Note 1	Note 2	ATANH

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
129	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	SIND
130	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	COSD
131	F A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE ARGUMENT ODD MULTIPLE OF 90			TAND
132	F A	DUPLICATE CHARACTER IN CSOWN CALL	Entry in collating sequence is defined twice.	Check CSOWN call.	CSOWN=
133	F A	IRRECONCILABLE STATUS OPTION	Status option irreconcilable.	Check status option.	OPECAP=
134	F A	STATUS OPTION INCOMPATIBLE WITH OLD FILE	Status option inconsistent with specified file.	Check status option and/or file name.	OPECAP=
135	F A	FORM CHANGE ON OPENED FILE	File was previously opened for a different processing type. The valid types are formatted, unformatted, buffer I/O, and random I/O.	Check processing type.	OPECAP=
136	F A	BAD RECL VALUE		Correct record length value.	OPECAP=
137	F A	BLANK OPTION ON UNFORMATTED FILE	Blank option applies to formatted I/O only.	Remove blank option.	OPECAP=
138	F A	BAD BUFL VALUE	Negative value, or, for open files, value not equal to the file buffer length.	For open files, verify that the BUFL value = the buffer length.	OPECAP=
139	F A	BAD OPEN OPTION	Option not allowed.	Check OPEN option.	OPECAP=
140	F A	ERROR DURING FILE CLOSING	Conflicting file attributes. CRM cannot perform close.	Check file attributes.	FORSYS
141	F A	BAD ARGUMENT TO ICHAR	Argument is not of type character, or does not have a character length of 1.	Check argument to ICHAR.	ICHAR=
142	F A	BAD CLOSE PARAMETER	Keyword specified for STATUS option was not 'KEEP' or 'DELETE'.	Correct keyword.	CLOSE

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class		Message	Significance	Action	Issued By
143	F	A	ACCESS CHANGE ON OPEN FILE	ACCESS option cannot be changed on open file.	Remove or change ACCESS option of OPEN statement.	OPECAP=
144	I	D	xxxx SUBSCRIPT OF ARRAY nnn = yyy, DECLARED LOWER WAS llll, UPPER WAS uuuu.	The xxxx subscript of array nnn has value yyy. This value is outside the range llll to uuuu.	Correct substring limits.	CDL=
145	I	D	STARTING CHARACTER POSITION OF xxxx SHOULD BE .GT. ZERO.	Substring reference outside of character string.	Correct substring limits.	CDL=
146	I	D	CHARACTER LENGTH OF xxxx SHOULD BE .GT. ZERO.	A negative character length is invalid.	Correct substring limits.	CDL=
147	I	D	NEW CHARACTER LENGTH OF xxxx EXCEEDS OLD LENGTH OF xxxx	New character length must be LE old character length.	Correct substring limits.	CDL=
148	F	A	INTERNAL FILE RECORD LENGTH .LE. ZERO	Record length must be positive.	Correct substring limits.	IIFC= OIFC=
149	F	A	INTERNAL FILE LCM RECORD EXCEEDS 150 CHARACTERS	Internal file LCM record cannot exceed 150 characters.	Reduce LCM record length or move record to SCM.	IIFC= OIFC=
150	F	A	INTERNAL FILE I/O LIST EXCEEDS FILE SIZE.	Internal file I/O list cannot exceed file size.	Check I/O list.	IIFC= OIFC=
151	F	A	DIRECT ACCESS OPEN HAS NO RECL PARAMETER	Record length parameter missing.	Insert the parameter.	OPECAP=
152	I	A	REWIND PROHIBITED ON DIRECT FILE -- IGNORED	REWIND used only for sequential files.	Remove REWIND.	REWIND=
153	F	A	ARGUMENT TO CSOWN NOT TYPE CHARACTER	Noncharacter argument passed to CSOWN.	Supply a collating sequence string argument.	CSOWN=
154	F	A	UNALLOCATED RECORD LENGTH GREATER THAN 150	Explicit open call attempted to make record length greater than 150 for an unallocated record in static mode.	Declare proper record length on PROGRAM statement.	OPECAP=
155	F	A	SEQUENTIAL I/O ATTEMPTED ON DIRECT FILE	Sequential I/O commands used on direct access file.	Use the direct access I/O commands.	OUTB= OUTC= OUTF= INPB= INPC= INPF=
156	F	A	CODED I/O attempted on xxxx FILE xxxx	Formatted READ or WRITE attempted on a file which was opened for unformatted, buffer, or random I/O.	Self-evident.	IDAB= ODAB=

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
157	F A	INVALID KEYWORD FOR COLSEQ	Attempt to specify an invalid collating sequence. The valid keywords are ASCII6, DISPLAY, INTBCD, and COBOL6.	Supply a valid keyword.	COLSEQ=
158	F A	OVER 1499 CHARACTERS IN REPEATED CHARACTER STRING	Character string with more than 1499 characters has a repeat factor for list directed input. Probably caused by missing apostrophe.	Add the terminating apostrophe or break the input into substrings.	LDIN=
159	F A	SCRATCH FILE xxxx CANNOT BE CLOSED WITH STATUS=KEEP	It is illegal to use status='KEEP' on a CLOSE when status='SCRATCH' was specified on the OPEN.	Correct the CLOSE or OPEN statement.	FORSYS=
160	F A	ILLEGAL USE OF ASTERISK AS STRING DELIMITER IN FORMAT	Asterisk is an invalid format string delimiter in FORTRAN 5.	Use apostrophe.	KODER= KRAKER=
161	F A	NON EXISTENT OVCAP	An attempt to load an OVCAP that does not exist.	Check that the name specified is the name of the first subroutine after an OVCAP statement.	LOVCAP or XOVCAP
162	F A	OVCAP .... IS ALREADY LOADED	LOVCAP has been called twice for the same OVCAP name.	Check program logic and eliminate redundant call.	LOVCAP or XOVCAP
163	F A	OVCAP .... WAS NEVER LOADED	A call to UOVCAP has been made specifying an OVCAP that has not been loaded.	Check program logic.	UOVCAP
164	F A	FDL ERROR XX DURING LOAD OR UNLOAD of OVCAP	A Fast Dynamic Loader error has been raised for reasons beyond user control.	Check error number in Loader reference manual. Follow site-defined procedure.	LOVCAP, XOVCAP, or UOVCAP
165	F A	INVALID SEQUENCE	SMKEY call specified a col-seg parameter without specifying a coding identifier of DISPLAY.	Ensure coding identifier is set to DISPLAY.	SMKEY
166	F A	RESERVED COL-SEQ	SMSEQ/SMEQU call specified a sequence name equivalent to one of the standard collating sequence names (ASCII6/COBOL6 /DISPLAY/INTBCD) in an attempt to redefine it.	Select another name for the user-supplied collating sequence.	SMSEQ/ SMEQU

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
167	F A	H, ', ", ILLEGAL INPUT FORMATS	Single quote, double quote, or H format are illegal in FORTRAN 5 input.	Correct format.	KRAKER=
168	F A	* DECIMAL POINT MISSING	Decimal point required.	Supply decimal point.	FMTAP=
169	F A	FORMAT VARIABLE DOES NOT CONTAIN ASSIGNED FORMAT	Assignment of a format to a variable used for I/O is not allowed.	Use ASSIGN statement to assign statement label to variable.	INPC= OUTC= IDAC= ODAC=
170	F A	ZERO LENGTH HOLLERITH STRING	Hollerith string must have a positive nonzero length.	Make string positive nonzero length.	KODER=
171	F A	BAD FILENAME GIVEN	Illegal character was used or character length was greater than 7.	Supply a valid file name.	OPECAP=
172-199 Reserved.					
200	F A	WRITE ATTEMPTED ON UNOPENED FILE xxxx	Direct access file must be opened before I/O is allowed.	Use OPEN command.	ODAB= ODAC=
201	F A	DIRECT WRITE ATTEMPTED ON SEQUENTIAL FILE xxxx	Direct I/O command used on sequential file.	Use sequential I/O command.	ODAB=
202	F A	BINARY WRITE ATTEMPTED ON xxxx FILE xxxx	Unformatted WRITE attempted on a file opened for formatted, buffer, or random I/O.	Self-evident.	ODAB=
203	F A	ATTEMPT TO WRITE NON-POSITIVE RECORD NUMBER	Record number must be positive.	Check record number.	ODAB= ODAC=
204	F A	READ ATTEMPTED ON UNOPENED FILE xxxx	Direct access file must be opened before I/O is allowed.	Use OPEN command.	IDAB= ODAC=
205	F A	DIRECT READ ATTEMPTED ON SEQUENTIAL FILE xxxx	Direct I/O command used on sequential file.	Use sequential I/O command.	IDAB= IDAC=
206	F A	BINARY READ ATTEMPTED ON xxxx FILE xxxx	Unformatted READ attempted from a file opened for formatted, buffer, or random I/O.	Self-evident.	IDAB=
207	F A	ATTEMPT TO READ NON-POSITIVE RECORD NUMBER	Record number must be positive.	Check record number.	IDAB= IDAC=
208	F A	LIST EXCEEDS RECORD LENGTH FOR FILE xxxx	List too long or record length too short.	Check record list and record length.	ODAB=

TABLE B-4. EXECUTION-TIME DIAGNOSTICS (Cont)

No.	Class	Message	Significance	Action	Issued By
209	F A	CMM ERROR IN CODED DIRECT-ACCESS OUTPUT	Common Memory Manager must be available to handle a record length greater than 1500 for formatted direct access I/O.	Define proper record length on the PROGRAM statement.	ODAC=
210	F A	DIRECT ACCESS ON SEQUENTIAL FILE	Direct I/O command used on sequential file.	Use sequential I/O command.	IDAC= IDAB= ODAC= ODAB=
211	F A	FORMATTED WRITE ON UNFORMATTED FILE	Unformatted I/O commands must be used with unformatted files.	Used unformatted command or change files.	ODAC= IDAC=
212	F A	CMM ERROR IN CODED DIRECT-ACCESS INPUT	Common Memory Manager must be available to handle a record length greater than 1500 for formatted direct access I/O.	See Common Memory Manager reference manual.	IDAC=
213	F A	DIRECT ACCESS ON SEQUENTIAL FILE	Direct I/O command used on sequential file.	Use sequential I/O command.	ODAB= ODAC= IDAB= IDAC=
214	F A	FORMATTED READ ON UNFORMATTED FILE	Unformatted I/O must be used with unformatted files.	Use unformatted command or change files.	IDAC=
215	F A	UNDEFINED WEIGHT PASSED TO CHAR	The character argument (weight) passed is not defined in the collating table.	Check character argument or current collating sequence.	CHAR= CHARF=
216	F A	SUBSTRING ERROR ON NAMELIST ITEM xxxx IN GROUP yyyy	Format of the substring is not correct.	Correct format of the substring.	NAMIN=
217	F A	NAMELIST ITEM xxxx IN GROUP yyyy, ITEM LENGTH	The substring has an upper bound greater than the length of the named character string.	Correct substring limits.	NAMIN=

Note 1 Infinites can be generated by dividing a nonzero number by zero, or by an addition, substraction, multiplication, or division whose result was greater than 10^{322} in absolute value. Indefinites are usually generated by dividing zero by zero.

Note 2 Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

GLOSSARY C

This glossary does not include terms defined in the ANSI standard for FORTRAN, X3.9-1978.

Advanced Access Methods (AAM) -
A file manager that processes indexed sequential, direct access, and actual key file organizations, and supports the Multiple-Index Processor. See CYBER Record Manager.

Basic Access Methods (BAM) -
A file manager that processes sequential and word addressable file organizations. See CYBER Record Manager.

Beginning-of-Information (BOI) -
CYBER Record Manager defines beginning-of-information as the start of the first user record in a file. System-supplied information, such as an index block, control word, or tape label, exists prior to beginning-of-information.

Blank Common Block -
An unlabeled common block. No data can be stored into a blank common block at load time. The size of the block is determined by the largest declaration for it. Contrast with Labeled Common Block.

Block -
In the context of input/output, a physical grouping of data on a file that provides faster data transfer. CYBER Record Manager defines four block types on sequential files: I, C, K, and E. Other kinds of blocks are defined for indexed sequential, direct access, and actual key files. Also refers to a common block.

Buffer -
An intermediate storage area used to compensate for a difference in rates of data flow, or times of event occurrence, when transmitting data between central memory and an external device during input/output operations.

Buffer Statement -
One of the input/output statements BUFFER IN or BUFFER OUT.

Common Block -
An area of memory that can be declared in a COMMON statement by more than one relocatable program and used for storage of shared data. See Blank Common Block and Labeled Common Block.

CYBER Record Manager (CRM) -
A generic term relating to the common products AAM and BAM that run under the NOS and NOS/BE operating systems, and which allow a variety of record types, blocking types, and file organizations to be created and accessed. The execution time input/output of COBOL 5, FORTRAN 5, Sort/Merge 4, ALGOL 4, and the DMS-170 products is implemented through CRM. Neither the input/output of the NOS and NOS/BE operating systems themselves, nor any of the system utilities such as COPY or SKIPF, is implemented through CRM. All CRM file processing requests ultimately pass through the operating system input/output routines.

Default Type -
The data type assumed by a variable in the absence of any type declarations for the variable. Variables whose names begin with one of the letters A through H or O through Z have a default type of real. Variables whose names begin with one of the letters I through N have a default type of integer.

Direct Access Input/Output -
A method of input/output in which records can be read or written in any order. Direct access input/output is performed by direct access READ and WRITE statements.

End-of-File (EOF) -
A particular kind of boundary on a sequential file, recognized by the END= parameter, the functions EOF and UNIT, and written by the ENDFILE statement. Any of the following conditions is recognized as end-of-file:

End-of-section (for INPUT file only)

End-of-partition

End-of-information (EOI)

W type record with flag bit set and delete bit not set

Tape mark

Trailer label

Embedded zero length level 17 block

End-Of-Information (EOI) -
The end of the last programmer record in a file. Trailer labels are considered to be past end-of-information. End-of-information is undefined for unlabeled S or L tapes.

Entry Point -
A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Equivalence Class -
A group of variables and arrays whose position relative to each other is defined as a result of an EQUIVALENCE statement.

Extended Memory -
Extended memory for the CDC CYBER 170 Models 171, 172, 173, 174, 175, 720, 730, 750, and 760 is extended core storage (ECS). Extended memory for the CDC CYBER 170 Model 176 is large central memory (LCM) or large central memory extended (LCME). ECS, LCM, and LCME are functionally equivalent, except as follows:

- LCM and LCME cannot link mainframes and do not have a distributive data path (DDP) capability.

- LCM and LCME transfer errors initiate an error exit, not a half exit. Refer to the COMPASS reference manual for complete information and a definition of half exit.

- The CYBER 170 Model 176 supports direct LCM and LCME transfer COMPASS instructions (octal machine language instruction codes 014 and 015). Refer to the COMPASS Reference Manual for complete information.

The storage level can be selected; refer to the LEVEL statement in section 2.

External File -
A file residing on an external storage device. An external file starts at beginning-of-information and ends at end-of-information. See File.

External Reference -
A reference in one program unit to an entry point in another program unit.

Field Length -
The area (number of words) in central memory assigned to a job.

File -
A logically related set of information; the largest collection of information that can be addressed by a file name. FORTRAN 5 recognizes two types of files, internal files and external files.

File Control Statement -
A control statement that contains parameters used to build the file information table for processing. Basic file characteristics such as organization, record type, and description can be specified on this statement.

File Information Table (FIT) -
A table through which a user program communicates with CYBER Record Manager. All file processing executes on the basis of fields in the table. Some fields can be set by the FORTRAN user in the FILE control statement.

Generic Function Name -
The name of an intrinsic function that can have arguments of any data type. Except for data type conversion generic functions, the data type of the result is the same as the data type of the arguments.

Implicit Type -
The type of a variable as declared in an IMPLICIT statement.

Internal File -
A character variable, array, or substring on which input/output operations are performed by formatted READ and WRITE statements. Internal files provide a method of transferring and converting data from one area of memory to another.

Labeled Common Block -
A common block into which data can be stored at load time. The first program unit declaring a labeled common block determines the amount of memory allocated. Contrast with Blank Common Block.

Logical File Name -
The name by which a file is identified; consists of one through seven letters or digits, the first a letter. Files used in standard FORTRAN 5 input/output statements can have a maximum of six letters or digits.

Main Overlay -
An overlay that must remain in memory throughout execution of an overlayed program.

Mass Storage Input/Output -
The type of input/output used for random access to files; it involves the subroutines OPENMS, READMS, WRITMS, CLOSMS, and STINDX.

Object Code -
Executable code produced by the compiler.

Object Listing -
A compiler-generated listing of the object code produced for a program, represented as COMPASS code.

Offset -
The starting position of the array in the first word of its storage (0 to 9).

Optimizing Mode -
One of the compilation modes in the FORTRAN 5 compiler, indicated by the control statement options OPT=0, 1, 2 or 3.

Overlay -
One or more relocatable programs that were relocated and linked together into a single absolute program. It can be a main, primary, or secondary overlay.

Partition -
CYBER Record Manager defines a partition as a division within a file with sequential organization. Generally, a partition contains several records or sections. Implementation of a partition boundary is affected by file structure and residence. Partition boundaries are shown in table C-1.

Notice that in a file with W type records a short PRU of level 0 terminates both a section and a partition.

Pass by Name -
A method of referencing a subprogram in which the addresses of the actual arguments are passed.

Pass by Value -
A method of referencing a subprogram in which only the values of the actual arguments are passed.

Primary Overlay -
A second level overlay that is subordinate to the main overlay. A primary overlay can call its associated secondary overlays and can reference entry points and common blocks in the main overlay.

Procedure -
A FORTRAN function subprogram, subroutine, or statement function.

Program Unit -
A sequence of FORTRAN statements terminated by an END statement. The FORTRAN program units are main programs, subroutines, functions, and block data subprograms.

PRU -
Under NOS and NOS/BE, the amount of information transmitted by a single physical operation of a specified device. The size of a PRU depends on the device: a PRU which is not full of user data is called a short PRU; a PRU that has a level terminator, but not user data, is called a zero-length PRU. PRU sizes are shown in table C-2.

TABLE C-1. PARTITION BOUNDARIES

Device	Record Type (RT)	Block Type (BT)	Physical Boundary
PRU device†	W	I	A short PRU of level 0 containing a one-word deleted record pointing back to the last I block boundary, followed by a control word with a flag indicating a partition boundary.
	W	C	A short PRU of level 0 containing a control word with a flag indicating a partition boundary.
	D,F,R, T,U,Z	C	A short PRU of level 0 followed by a zero-length PRU of level 17_8.
	S	-	A zero-length PRU of level number 17_8.
S or L format tape	W	I	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with a flag indicating a partition boundary.
	W	C	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with a flag indicating a partition boundary.
	D,F,T, R,U,Z	C,K,E	A tapemark.
	S	-	A tapemark.
Any other tape format	-	-	Undefined.

†NOS and NOS/BE only.

TABLE C-2. PRU SIZES

Device	Size in Number of 60-Bit Words
Mass storage (NOS and NOS/BE only).	64
Tape in SI format with coded data (NOS/BE only).	128
Tape in SI format with binary data.	512
Tape in I format (NOS only).	512
Tape in any other format.	Undefined.

PRU Device -
A mass storage device or a tape in SI (NOS and NOS/BE), I (NOS and NOS/BE), or X (NOS/BE only) format, so called because records on these devices are written in PRUs.

Record -
CYBER Record Manager defines a record as a group of related characters. A record or a portion thereof is the smallest collection of information passed between CYBER Record Manager and a user program in a single read or write operation. Eight different record types exist, as defined by the RT field of the file information table.

Other parts of the operating systems and their products might have additional or different definition of records.

Record Length -
The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Record Type -
The term record type can have one of several meanings, depending on the context. CYBER Record Manager defines eight record types established by an RT field in the file information table.

Reference Listing -
A part of listing produced by a FORTRAN compilation, which displays some or all of the entities used by the program, and provides other information such as attributes and location of these entities.

Relocation -
Placement of object code into central memory in locations that are not predetermined, and adjusting the addresses accordingly.

SCOPE 2 Record Manager -
The record manager used under the SCOPE 2 operating system. It processes all files read and written as a result of user requests at execution time, as well as all files read and written at compile time by the compiler. The SCOPE 2 Record Manager processes all input/output files.

Secondary Overlay -
The third level of overlays. A secondary overlay is called into memory by its associated primary overlay. A secondary overlay can reference entry points and common blocks in both its associated primary overlay and the main overlay.

Section -
CYBER Record Manager defines a section as a division within a file with sequential organization. Generally, a section contains more than one record and is a division within a partition of a file. A section terminates with a physical representation of a section boundary. Section boundaries are described in table C-3.

The NOS and NOS/BE operating systems equate a section with a system-logical-record of level 0 through 16_8.

Sequential -
A file organization in which the location of each record is defined only as occurring immediately after the preceding record. A file position is defined at all times, which specifies the next record to be read or written.

Sequential Access Input/Output -
A method of input/output in which records are processed in the order in which they occur on a storage device.

Source Code -
Code written by the programmer in a language such as FORTRAN, and input to a compiler.

Source Listing -
A compiler-produced listing, in a particular format, of the user's original source program.

Specific Function Name -
The name of an intrinsic function that accepts arguments of a particular data type, and returns a result of a particular data type.

System-Logical-Record -
Under NOS/BE, a data grouping that consists of one or more PRUs terminated by a short PRU or zero-length PRU. These records can be transferred between devices without loss of structure.

Unit Specifier -
An integer constant, or an integer variable with a value of either 0 to 999, or an L format logical file name. In input/output statements, it indicates on which unit the operation is to be performed. It is linked with the actual file name by the PROGRAM statement or OPEN statement.

Word Addressable -
A file organization in which the location of each record is defined by the ordinal of the first word in the record, relative to the beginning of the file.

Working Storage Area -
An area within the user's field length, intended for receipt of data from a file or transmission of data to a file. Transmission to or from a buffer intervenes, except for buffer statements.

Zero-Byte Terminator -
12 bits of zero in the low order position of a word that marks the end of the line to be displayed at a

TABLE C-3. SECTION BOUNDARIES

Device	Record Type (RT)	Block Type (BT)	Physical Representation
PRU device	W	I	A deleted one-word record pointing back to the last I block boundary followed by a control word with flags indicating a section boundary. At least the control word is in a short PRU of level 0.
	W	C	A control word with flags indicating a section boundary. The control word is in a short PRU of level 0.
	D,F,R, T,U,Z	C	A short PRU with a level less than 17_8.
	S	-	Undefined.
S or L format tape	W	I	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with flags indicating a section boundary.
	W	C	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with flags indicating a section boundary.
	D,F,R, T,U,Z	C,K,E	Undefined.
	S	-	Undefined.
Any other tape format	-	-	Undefined.

LANGUAGE SUMMARY D

The following symbols are used in the descriptions of the FORTRAN 5 statements:

v	variable name, array name, or array element
sl	statement label
iv	integer variable
name	symbolic name
u	input/output unit specifier, which can be an integer expression with a value of 0 through 999, or a Boolean expression containing a display code file name in L format
fs	format specifier
iolist	input/output list
ios	input/output status specifier
recn	record number

Other symbols are defined individually in the statement descriptions.

ac — Is a single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

EXTERNAL DECLARATION

EXTERNAL name [,name]... 2-9

INTRINSIC DECLARATION

INTRINSIC name [,name]... 2-10

STORAGE ALLOCATION

type array(d) [,array(d)]... 2-4

DIMENSION array(d) [,array(d)]... 2-4

type Is INTEGER, CHARACTER, BOOLEAN, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

d Is one through seven array bound expressions separated by commas, as described in section 2.

COMMON [/[name]/]nlist [[,]/[name]/nlist]... 2-6

nlist Is a list of variables or arrays, separated by commas, to be included in the common block.

DATA nlist/clist/ [[,]nlist/clist/]... 2-11

nlist Is a list of names to be initially defined. Each name in the list can take the form:

variable
array
element
substring
implied DO list

clist Is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in section 2.

EQUIVALENCE (nlist) [,(nlist)]... 2-7

nlist Is a list of variable names, array names, array element names, or character substring names. The names are separated by commas.

LEVEL n,name [,name]... 2-8

n Is an unsigned integer constant, or symbolic constant, with the value 0, 1, 2, or 3 indicating the storage level.

PARAMETER (name=exp [,name=exp] ...) 2-5

exp Is a constant or constant expression.

SAVE [name [,name]...] 2-9

FLOW CONTROL

GO TO sl 4-1

GO TO (sl[,sl]...)[,] expression 4-1

GO TO iv [[,](sl [,sl]...)] 4-2

ASSIGN sl TO iv 4-2

IF (arithmetic or Boolean expression) sl_1,sl_2,sl_3 4-2

IF (logical expression) statement 4-3

IF (logical expression) THEN 4-3

ELSE IF (logical expression) THEN 4-3

ELSE 4-3

END IF 4-4

DO sl [,] $v=e_{1},e_{2}[,e_{3}]$ 4-5

e_{1},e_{2},e_{3} Are indexing parameters. They can be integer, real, double precision, or Boolean constants, symbolic constants, variables, or expressions.

PAUSE [n] 4-9

STOP [n] 4-10

n Is a string of 1 through 5 digits, or a character constant.

END 4-10

MAIN PROGRAM

PROGRAM name (fpar[,fpar]...) 6-1

fpar Is a file declaration in one of the following forms:

filename

filename=buffer length

filename=/record length

filename=buffer length/record length

alternate name=filename

SUBPROGRAM

SUBROUTINE name [(argument [,argument]...)] 6-3

[type] FUNCTION name([argument [,argument]]...) 6-4

type Is BOOLEAN, CHARACTER, INTEGER, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

BLOCK DATA [name] 6-3

STATEMENT FUNCTION

name ([argument [,argument]]...)=expression 6-5

SUBROUTINE CALL

CALL name [(argument[,argument]...)] 6-9

FUNCTION REFERENCE

name ([argument, [argument]]...) 6-10

ENTRY POINT

ENTRY name [(argument [,argument]...)] 6-6

RETURN

RETURN [expression] 6-10

FORMATTED INPUT/OUTPUT

READ ([UNIT=]u, [FMT=]fs [,IOSTAT=ios] [,ERR=sl] [,END=sl]) [iolist] 5-4

READ fs [,iolist] 5-4

WRITE ([UNIT=]u, [FMT=]fs [,IOSTAT=ios] [,ERR=sl]) [iolist] 5-5

PRINT fs [,iolist] 5-5

PUNCH fs [,iolist] 5-5

UNFORMATTED INPUT/OUTPUT

READ ([UNIT=]u [,IOSTAT=ios] [,ERR=sl] [,END=sl]) [iolist] 5-22

WRITE ([UNIT=]u [,IOSTAT=ios] [,ERR=sl]) [iolist] 5-22

LIST DIRECTED INPUT/OUTPUT

READ ([UNIT=] u,[FMT=] * [,IOSTAT=ios] [,ERR=sl] [,END=sl]) [iolist] 5-22

READ *[,iolist] 5-22

WRITE ([UNIT=] u,[FMT=] * [,IOSTAT=ios] [,ERR=sl]) [iolist] 5-24

PRINT * [,iolist] 5-24

PUNCH * [,iolist] 5-24

DIRECT ACCESS INPUT/OUTPUT

READ ([UNIT=] u,[FMT=] fs [,IOSTAT=ios] [,ERR=sl] [,REC=recn]) [iolist] 5-29

WRITE ([UNIT=] u,[FMT=] fs [,IOSTAT=ios] [,ERR=sl] [,REC=recn]) [iolist] 5-30

NAMELIST INPUT/OUTPUT

NAMELIST /name/v [,v]...[/name/v [,v]...]... 5-25

READ ([UNIT=] u,[FMT=] name [,IOSTAT=ios] [,ERR=sl] [,END=sl]) 5-26

READ name 5-26

WRITE ([UNIT=] u,[FMT=] name [,IOSTAT=ios] [,ERR=sl]) 5-27

PRINT name 5-27

PUNCH name 5-27

name Is a NAMELIST group name.

BUFFER INPUT/OUTPUT

BUFFER IN (u,p) (a,b) 5-28

BUFFER OUT (u,p) (a,b) 5-29

p Is an integer constant or variable:

zero = even parity

nonzero = odd parity

a Is the first word of the data block to be transferred.

b Is the last word of the data block to be transferred.

INTERNAL DATA TRANSFER

ENCODE (c,fs,v)iolist — 5-36

DECODE (c,fs,v)iolist — 5-36

v	Is the starting location of the record to be transferred.
c	Specifies the number of characters to be transferred to or from each record.

FORMAT SPECIFICATION

sl FORMAT (flist) — 5-5

flist Is a list of items, separated by commas, having the following forms:

[r]ed
ned
[r](flist)

ed	Is a repeatable edit descriptor.
ned	Is a nonrepeatable edit descriptor.
r	Is a nonzero unsigned integer constant repeat specification.

EDIT DESCRIPTORS

srEw.d	Single precision floating-point with exponent.	5-8
srEw.dEe	Single precision floating-point with specified exponent length.	5-8
srFw.d	Single precision floating-point without exponent.	5-10
srDw.d	Double precision floating-point with exponent.	5-11
srGw.d	Single precision floating-point with or without exponent.	5-10
srGw.dEe	Single precision floating-point with or without specified exponent length.	5-10
rIw	Decimal integer.	5-8
rIw.m	Decimal integer with specified minimum number of digits.	5-8
rLw	Logical.	5-14
rA	Character with variable length.	5-13
rAw	Character with specified length.	5-13
rRw	Rightmost characters with binary zero fill.	5-14
rOw	Octal.	5-15
rOw.m	Octal with minimum digits and leading zeros.	5-15
rZw	Hexadecimal.	5-15
rZw.m	Hexadecimal with minimum digits and leading zeros.	5-15

s	Is an optional scale factor of the form kP.
r	Is an optional repetition factor.
w	Is an integer constant indicating field width.
d	Is an integer constant indicating digits to right of decimal point.
e	Is an integer constant indicating digits in exponent field.
m	is an integer constant indicating minimum number of digits in field.
n	is a positive nonzero decimal digit.

BN Blanks ignored on numeric input. — 5-13

BZ	Blanks treated as zeros on numeric input.	5-13
SP	+ characters produced on output.	5-13
SS	+ characters suppressed on output.	5-13
S	+ characters suppressed on output.	5-13
nX	Skip n spaces.	5-16
Tn	Tabulate to nth column.	5-17
TRn	Tabulate forward.	5-17
TLn	Tabulate backward.	5-17
nH	Hollerith or character string output.	5-16
"..."	Hollerith or character string output.	5-16
'...'	Hollerith or character string output.	5-16
:	Format control.	5-20
/	End of FORTRAN record.	5-17

FILE POSITIONING

BACKSPACE ([UNIT=] u[,IOSTAT=ios] [,ERR=sl]) 5-38

BACKSPACE u 5-38

REWIND ([UNIT=] u [,IOSTAT=ios] [,ERR=sl]) 5-37

REWIND u 5-37

ENDFILE ([UNIT=] u [,IOSTAT=ios] [,ERR=sl]) 5-38

ENDFILE u 5-38

FILE STATUS

OPEN ([UNIT=] u [,IOSTAT=ios] [,ERR=sl] [,FILE=fin] [,STATUS=sta]
[,ACCESS=acc] [,FORM=fm] [,RECL=rl] [,BLANK=blnk] [BUFL=bl]) 5-30

INQUIRE ({[UNIT=]u / FILE=fin} [,IOSTAT=ios] [,ERR=sl] 5-33
[,EXIST=ex] [,OPENED=od] [,NUMBER=num] [,NAMED=nmd]
[,NAME=fn] [,ACCESS=acc] [,SEQUENTIAL=seq] [,DIRECT=dir]
[,FORM=fm] [,FORMATTED=FMT] [,UNFORMATTED=unf] [,RECL=fcl]
[,NEXTREC=nr] [,BLANK=blnk])

CLOSE ([UNIT=] u [,IOSTAT=ios] [,ERR=sl] [,STATUS=sta]) 5-32

OVERLAYS

OVERLAY ([fname,] i,j [,orig] [,OV=m]) 9-3

fname	Is the name of the file on which the overlay is to be written.
i,j	Are the overlay level numbers.
orig	Specifies the origin of the overlay.
m	Optional specification of number of higher level overlays.

CALL OVERLAY (fname,i,j [,recall] [,k]) 9-3

recall	Is the recall parameter.
k	Indicates location of fname.

Also see CYBER Loader reference manual

A C$ directive is a special form of comment line that controls compiler processing. A particular C$ directive affects an aspect of the compiler's interpretation of those lines following the directive and preceding either a subsequent directive modifying the same aspect, if such a directive appears, or the end of the program unit. The aspects of interpretation that can be controlled are:

- Listing of the program and associated compiler-produced information, called listing control
- Specification of program lines to be processed or ignored, called conditional compilation
- Character data comparison collation table, called collation control
- Minimum trip count and long trip count for DO loops, called DO loop control

The general form of a C$ directive is shown in figure E-1.

C$ keyword[(p[=c] [,p[=c]] . . .)] [[,] lab]

keyword	Is one of LIST, IF, ELSE, ENDIF, COLLATE, or DO. The keyword can begin in any column starting with column 7. In sequenced mode the keyword can begin in any column following the character $.
p	Is a parameter. Depending upon the keyword that appears, one or more parameters can be specified.
c	Is an integer constant, or symbolic name of an integer constant, with a value of zero or one. Depending upon the parameter p, the constant either is optional or must not appear.
lab	Is a label. Depending upon the keyword that appears, a label may be specified. If a label appears and no parameters are present, a comma must separate the keyword and the label.

Figure E-1. C$ Directive

The letter C in column 1 together with the character $ in column 2 identify a line as a C$ directive line. Such a line will be interpreted as a comment only if the directive suppression (DS) option is specified on the FTN5 control statement. The entire directive must appear on a single line. A C$ directive interrupts statement continuation.

In sequenced mode the letter C in the column immediately to the right of the sequence number together with the character $ immediately to the right of the C identify a C$ directive line. A line with no sequence number in sequence mode cannot be a C$ directive.

LISTING CONTROL

A listing control directive has the keyword LIST. It must have the form shown in figure E-2.

C$ LIST(p[=c] [,p[=c]] . . .)

p	Is S, O, R, A, M or ALL.
c	Is a constant or the symbolic name of a constant.

Figure E-2. Listing Control Directive

The constant is optional for all parameters; its absence is equivalent to the appearance of a constant with the value 1.

The listing control directive modifies the state of any initially enabled list option switches. A list option switch is initially enabled when the corresponding list option is requested on the FTN5 control statement. Any attempt to modify a list option switch that was not initially enabled is ignored: p=0 disables switch p; p=1 enables switch p.

ALL=c is equivalent to S=c, O=c, R=c, A=c, M=c.

A listing control directive found by the compiler to be in error results in a warning diagnostic.

The list option switches offer the following control:

S Source lines are listed when enabled.

O Generated object code is listed for statements processed when enabled.

R Symbol references are accumulated for the cross-reference list when enabled. Symbols with no accumulated references will not appear in that list; no accumulation for an entire program unit suppresses cross-reference list.

A The symbol attribute list is generated if this switch is enabled when the END statement is processed.

M The symbol map list is generated if this switch is enabled when the END statement is processed.

An example of listing control directives is shown in figure E-3. The complete output listing is shown. All source statements appearing between C$ LIST (S=0) and C$ LIST (S=1) are suppressed in the output listing. (Source statement lines with errors are listed on the ERROR file along with diagnostics.) The C$ LIST (ALL=0) directive, active when the END statement is encountered, suppresses the reference map.

Program containing listing control directive:

FTN5,LO=S/A/R/M.
7/8/9 in column 1

```
      PROGRAM P
C  PROGRAM TO TEST LISTING CONTROL DIRECTIVES
C$    LIST(S=0)
      DIMENSION A(10)
C  THE FOLLOWING CARD CONTAINS AN ERROR
      INTEGER B/C
C$    LIST(S=1)
      DO 100 I=1,10
100   A(I) = 0.0
C$    LIST(ALL=0)
      END
```

6/7/8/9 in column 1

Compiler output listing:

```
                1            PROGRAM P
                2          C  PROGRAM TO TEST LISTING CONTROL DIRECTIVES
                3          C$    LIST(S=0)
                6                INTEGER B/C
FATAL      *    6          EXPECTED COMMA -- FOUND  /
                7          C$    LIST(S=1)
                8                DO 100 I=1,10
                9          100   A(I) = 0.0
               10          C$    LIST(ALL=0)

                1 FATAL     ERROR IN  P
```

Figure E-3. Listing Control Directive Example

CONDITIONAL COMPILATION

A conditional compilation directive has a keyword which is one of IF, ELSE, or ENDIF. Such a directive controls whether the lines immediately following the directive are to be processed or ignored by the compiler.

The conditional compilation directives are divided into three categories:

- An IF directive with the keyword IF
- An ELSE directive with keyword ELSE
- An ENDIF directive with keyword ENDIF

The IF directive, ELSE directive, and ENDIF directive are shown in figures E-4, E-5, and E-6, respectively.

For each IF directive there must appear exactly one ENDIF directive later in the same program unit, and for each ENDIF directive there must appear exactly one IF directive earlier in the same program unit. Between an IF directive and its corresponding ENDIF directive will appear zero or more lines called a conditional sequence. A conditional sequence can optionally contain one ELSE directive corresponding to the IF directive and ENDIF directive delimiting the conditional sequence. An ELSE directive can appear only within a conditional sequence. A conditional sequence can not contain more than one ELSE directive unless it contains another conditional sequence. If an ELSE directive is contained within more than one conditional sequence, the ELSE directive corresponds to that IF-ENDIF pair which delimits the smallest, that is, innermost, conditional sequence containing the ELSE directive.

C$ IF(e)[[,] lab]

e Is a logical constant expression. If a symbolic constant appears, it must have been previously defined in a PARAMETER statement in the program containing the IF directive.

lab Is an optional label. It must be a symbolic name. Its use as a label in a conditional compilation directive does not affect, and is not affected by, its use for any other purpose in the program unit.

Figure E-4. IF Directive

C$ ELSE[,lab]

lab Is as for an IF directive.

Figure E-5. ENDIF Directive

C$ ENDIF[,lab]

lab Is as for an IF directive.

Figure E-6. ELSE Directive

If two (or three) corresponding conditional directives have a label, it must be the same label. No other restriction applies to labels on conditional directives. There is no requirement that any conditional directive have a label. The same label can be used on more than one sequence of corresponding conditional directives in a single program unit, including the case of conditional directives whose conditional sequence contains other conditional directives with the same label.

A conditional sequence can contain any number of properly corresponding conditional directives, and therefore other conditional sequences. If two conditional sequences contain the same line, one conditional sequence must lie wholly within the other conditional sequence.

A conditional compilation directive found by the compiler to be in error results in a diagnostic message.

If an IF directive is processed by the compiler and the logical expression is true, following lines are processed as if the IF directive had not appeared, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are ignored by the compiler. If an IF directive is processed by the compiler and the logical expression is false, the following lines are ignored until the corresponding ENDIF directive is encountered, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are processed.

An example of conditional compilation directives is shown in figure E-7. The sample program contains two DO loops. Conditional compilation directives are included to test the value of the symbolic constant M. If M is 0, the first loop is ignored and the second loop is compiled. If M is 1, the first loop is compiled and the second loop is ignored. The program is compiled and executed two times; once with the statement PARAMETER (M=0) and once with the statement PARAMETER (M=1).

COLLATION CONTROL

A collation control directive has the form shown in figure E-8.

The collation control directive specifies whether collation of character relational expressions is directed by the fixed or user-specified weight table.

A collation control directive directs the interpretation of character relational expressions in the lines following the directive and preceding either another collation control directive or the END statement of the program unit. In the case of a character relational expression in a statement function statement, the collation that applies is that in effect for the line or lines containing a reference to the statement function. Consider the example:

```
      PROGRAM P
      LOGICAL LSF
      CHARACTER*5, X, Y, S, T
C$    COLLATE(USER)
      LSF(X,Y) = X.LT.Y
      .
      .
      .
C$    COLLATE(FIXED)
      IF (LSF(S,T)) A=1.0
      .
      .
      .
      END
```

Example 1:

```
      PROGRAM B
      PARAMETER (M=1)
      DIMENSION A(10)
      DATA A/10*0.0/
C$    IF(M .EQ. 0)
      DO 8 I=1,10
8     A(I) = A(I) + 1.0
C$    ELSE
      DO 12 I=1,10
12    A(I) = A(I) - 1.0
C$    ENDIF
      PRINT*, ' A= ', A
      STOP
      END
```

Output:

```
A= -1. -1. -1. -1. -1. -1. -1. -1.
   -1. -1.
```

Example 2:

```
      PROGRAM B
      PARAMETER (M=0)
      DIMENSION A(10)
      DATA A/10*0.0/
C$    IF(M .EQ. 0)
      DO 8 I=1,10
8     A(I) = A(I) + 1.0
C$    ELSE
      DO 12 I=1,10
12    A(I) = A(I) - 1.0
C$    ENDIF
      PRINT*, ' A= ', A
      STOP
      END
```

Output:

```
A= 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
```

Figure E-7. Conditional Compilation Example

C$ COLLATE(p)

p Is FIXED or USER.

Figure E-8. Collation Control Directive

The reference LSF(S,T) results in an evaluation of the character relational expression S.LT.T with the collation that of the fixed weight table.

A collation control directive found by the compiler to be in error results in a diagnostic.

DO LOOP CONTROL

A DO loop control directive has the form shown in figure E-9.

```
C$    DO(p[=c][,p[=c]])
```

p	Is either OT or LONG.
c	Is a constant or the symbolic name of a constant.

Figure E-9. DO Loop Control Directive

The constant is optional for both parameters; its absence is equivalent to the appearance of a constant with the value 1.

The DO loop control directive modifies the state of one or both DO loop switches. Each switch is initially set (or reset) when the corresponding DO option (appendix E) is (or is not) requested at compile time. A DO loop control directive switch selection overrides the corresponding DO option request.

The OT parameter controls the minimum trip count. If OT is set (OT=1), the minimum trip count for DO loops is one. If OT is not set (OT=0), the minimum trip count for DO loops is zero.

The LONG parameter controls the maximum trip count. If LONG is set (LONG=1) the trip count can exceed 131 071. If LONG is not set (LONG=0), the trip count cannot exceed 131 071.

A DO loop control directive affects the interpretation of only those DO loops whose DO statements follow the directive in the same program unit.

A DO loop control directive found by the compiler to be in error results in a diagnostic.

INPUT/OUTPUT IMPLEMENTATION F

This section describes the structure of files read and written by FORTRAN 5. All files read and written as a result of user requests at execution time are processed through Record Manager. The files read and written at compile time by the compiler itself (including source input, coded output, and binary output) are processed by operating system routines when compilation is under NOS or NOS/BE, and by SCOPE 2 Record Manager when compilation is under SCOPE 2.

EXECUTION-TIME INPUT/OUTPUT

All input and output between a file referenced in a program and the file storage device is under control of Record Manager. The version of Record Manager used depends on the operating system.

NOS and NOS/BE use CYBER Record Manager Basic Access Methods (BAM), encompassing sequential and word addressable file organizations, for standard input/output statements, and CYBER Record Manager Advanced Access Methods (AAM) for indexed sequential, direct access, and actual key file organizations, and multiple-index capability, through the CYBER Record Manager interface routines.

SCOPE 2 uses the SCOPE 2 Record Manager for all input/output.

CYBER Record Manager can be called directly, as described in section 8, to use the extended file structure and processing available. SCOPE 2 Record Manager cannot be called directly from the FORTRAN 5 compiler. This appendix deals only with Record Manager processing that results from standard language use.

File processing is governed by values compiled into the file information table (FIT) for each file. If a file or its FIT is changed by other than standard FORTRAN input/output statements, subsequent FORTRAN input/output to that file may not function correctly. Thus, it is recommended that the user not try to use both standard FORTRAN and nonstandard input/output on the same file within a program.

FILE AND RECORD DEFINITIONS

A file is a collection of records referenced by its logical file name. It begins at beginning-of-information and ends with end-of-information. A record is data created or processed by:

- One execution of an unformatted READ or WRITE
- One card image or a print line defined within a formatted, list directed, or namelist READ or WRITE
- One call to READMS or WRITMS
- One execution of BUFFER IN or BUFFER OUT

On storage, a file can have records in one of eight formats (record types) defined to Record Manager. Only four of these are part of standard processing:

Z Record is terminated by a 12-bit zero byte in the low order byte position of a 60-bit word.

W Record length is contained in a control word prefixed to the record by Record Manager.

U Record length is defined by the user.

S System logical record.

The remaining types can be formatted within a program under user control and written to a device using a WRITE statement if the FILE control statement is used to specify another record type. Similarly, these types can be read by a READ statement.

The user is responsible for supplying record length information appropriate to each type before a write and for determining record end for a read. For example, a D type record requires a field within the record to specify record length, and F type records require that the user READ/WRITE exactly FL characters in each record.

Unformatted READ and WRITE are implemented through the GETP and PUTP macros of Record Manager; consequently, record operations must conform to macro restrictions. Specifically, RT=R and RT=Z cannot be specified for unformatted operations.

Direct access I/O must be done with RT=U. RT=U is the default.

STRUCTURE OF INPUT/OUTPUT FILES

FORTRAN 5 sets certain values in the file information table depending on the nature of the input/output operation and its associated file structure. Table F-1 lists these values for their respective FIT fields; all except those marked with an asterisk (*) can be overridden at execution time by a FILE control statement. (Numbers in parentheses refer to notes listed following the table.)

Sequential Files

The following information is valid, unless the FIT field is overriden by a FILE control statement.

With READ and WRITE statements, the record type (RT) depends on whether the access is formatted or unformatted (applies only to NOS and NOS/BE). A formatted WRITE produces RT=Z records, with each record terminated by a system-supplied zero byte in the low order bits of the last word in the record. An unformatted WRITE produces RT=W records, in which each record is prefixed by a system-supplied control word. Blocking is type C for formatted and I for unformatted records. The files named INPUT, OUTPUT, and PUNCH always have record type Z and block type C. These files should only be processed by formatted, list directed, and namelist input/output statements.

On SCOPE 2 only with READ and WRITE statements, the record type is W for all file types; blocking is I for tape files, and unblocked for all other files.

PRINT and PUNCH statements produce Z type records with C type blocks or on SCOPE 2 only, W type records unblocked for processing on unit record equipment.

BUFFER IN and BUFFER OUT assume S type records or, on SCOPE 2 only, W type records. Formatting is determined by the parity designator in each BUFFER statement. An unformatted operation does not convert character codes during tape reading or writing (CM=NO), while a formatted operation does.

The ENDFILE statement writes a boundary condition known as an end-of-partition. When this boundary is encountered during a read, the EOF function returns end-of-file status. An end-of-partition may not necessarily coincide with end-of-information, however, and reading can continue on the same file until end-of-information on the file has been encountered.

End-of-partition is written as the file is closed during program termination. A third boundary for sequential files, a section, is not recognized during reading except for the special case of the file INPUT.

Mass Storage Input/Output

Files created by the random mass storage routines OPENMS, WRITMS, STINDX, and CLOSMS described in section 7 are word addressable files. The master index, which is the last record in the file, is created and maintained by FORTRAN routines rather than Record Manager routines.

One WRITMS call creates one U type record; one READMS call reads one U type record. If the length specified for a READMS is longer than the actual record, the excess locations in the user area are not changed by the read. If the record is longer than the length specified for a READMS, the excess words in the record are skipped.

Direct Access Input/Output

Files created by direct access READ and WRITE statements are word addressable files. There is no index. Except where the format specifies multiple records, one direct access WRITE creates one U type record and one direct access READ reads one U type record.

FILE CONTROL STATEMENT

The FILE control statement provides a means to override FIT field values compiled into a program and consequently a means to change processing normally supplied for standard input/output. In particular, it can be used to read or create a file with a structure that does not conform to the assumptions of default processing.

A FILE control statement can also be used to supplement standard processing. For example, setting DFC can change the type of Record Manager information listed in the dayfile.

At execution time, FILE control statement values are placed in the FIT when the referenced file is opened. These values have no effect if the execution routines do not use the fields referenced. Furthermore, FORTRAN routines may, in some cases, reset FIT fields after the FILE control statement is processed. These fields are noted in table F-1.

The format of the FILE control statement is shown in figure F-1.

FILE(lfn,field=value[,field=value]...)

lfn	**Is the file name as it appears on the execution control statement; if file name does not appear there, then lfn is file name as it appears in the PROGRAM or OPEN statement.**
field	**Is a FIT field mnemonic.**
value	**Is a symbolic or integer value.**

Figure F-1. FILE Control Statement

The FILE control statement can appear anywhere in the control statements prior to program execution, but it must not interrupt a load sequence.

This deck shown in figure F-2 illustrates the use of the FILE control statement to override default values supplied by the FORTRAN compiler. Assuming the source program is using formatted writes and 100-character records are always written, the file is written on magnetic tape with even parity, at 800 bpi. No labels are recorded, and no information is written except that supplied by the user. The following values are used:

Block type = character count

Record type = fixed length

Record length = 100 characters

Conversion mode = YES

SEQUENTIAL FILE OPERATIONS

The sequential file operations are BACKSPACE/REWIND and ENDFILE.

Backspace/Rewind

Backspacing on FORTRAN files repositions them so that the previous record becomes the next record.

BACKSPACE is permitted only for files with F, S, or W record type or tape files with one record per block.

The user should remember that formatted input/output operations can read/write more than one record; unformatted input/output and BUFFER IN/OUT read/write only one record.

The REWIND operation positions a magnetic tape file so that the next FORTRAN input/output operation references the first record. A mass storage file is positioned to the beginning-of-information.

Table F-2 details the actions performed prior to positioning.

TABLE F-1. DEFAULTS FOR FIT FIELDS

FIT Fields		Formatted, NAMELIST, and List Directed Sequential READ/WRITE	Unformatted Sequential READ/WRITE	BUFFER IN/ BUFFER OUT	Mass Storage Input/Output	Direct Access I/O Formatted and Unformatted
Meaning	Mnemonic					
CIO buffer size (words)	(1) BFS†	(1)	(1)	(1)	(1)	(1)
Buffer Below Highest Address	BBH	0	0	n/a	0	0
Block type	BT	C† /(9)††	I† /(9)††	C† /(9)††	n/a	C*
Close flag (positioning of file after close)	CF	N*	N*	N*	N*† /R*††	N*
Length in characters of record trailer count field (T type records only)	CL	0	0	0	n/a	n/a
Conversion mode	CM	YES† /NO	NO	(2)	n/a	n/a
Beginning character position of trailer count field, numbered from zero (T type records only)	CP	0	0	0	n/a	n/a
Length field (D type records) or trailer count field (T type records) is binary	C1†	NO	NO	NO	n/a	n/a
Type of information to be listed in dayfile	DFC†	3	3	3	3	3
Type of information to be listed on error file	EFC†	0	0	0	0	0
Error options	EO	AD	AD	AD	AD	AD
Trivial error limit	ERL	0	0	0	0	0
Fatal Flush	FF†	0	0	n/a	0	0
Length in characters of an F or Z type record (same as MRL)	FL†	150(5)*	n/a	n/a	n/a	n/a
File organization	FO	SQ *	SQ *	SQ *	WA *	WA *
Character length of fixed header for T type records	HL	0	0	0	n/a	n/a
Length of user's label area (number of characters)	(7) LBL	0 *	0 *	0 *	n/a	n/a
Logical file name	LFN	(3)	(3)	(3)	(3)	(3)
Length in characters of record length field (D type records)	LL	0	0	0	n/a	n/a

TABLE F-1. DEFAULTS FOR FIT FIELDS (Contd)

FIT Fields		Formatted, NAMELIST, and List Directed Sequential READ/WRITE	Unformatted Sequential READ/WRITE	BUFFER IN/ BUFFER OUT	Mass Storage Input/Output	Direct Access I/O Formatted and Unformatted
Meaning	Mnemonic					
Beginning character position of record length, numbered from zero (D type records)	LP	0	0	0	n/a	n/a
Label type	(7) LT	ANY	ANY	ANY	n/a	n/a
Maximum block length in characters	MBL	0	0	0	n/a	n/a
Minimum block length in characters	MNB†	0	0	0	n/a	n/a
Minimum record length in characters	MNR†	0	0	0	n/a	n/a
Maximum record length in characters	(5) MRL	n/a	$2^{23}-1$	(8) *	n/a	n/a
Multiple of characters per K, E type block	MUL†	2	2	2	n/a	n/a
Open flag (positioning of file after open)	(7) OF	N*	N*	N*	N*†/R†† *	N*
Padding character for K, E type blocks	PC†	76B	76B	76B	n/a	n/a
Processing direction	PD	IO	IO	IO	IO	IO
Number of records per K type block	RB	1	1	1	n/a	n/a
Record mark character (R records)	RMK	62B	n/a	62B	n/a	n/a
Record type	RT	Z†/W††(10)	W(6)	S†/W††	U	U *
Length field (D type records) or trailer count field (T type records) has sign overpunch	SB†	NO	NO	NO	n/a	n/a
Suppress buffering	SBF†	NO*	NO*	YES(11)	NO*	NO*
Suppress read ahead	SPR	NO	NO	NO	n/a	n/a
Character length of trailer portion of T type records	TL	0	0	0	n/a	n/a
User label processing	(7) ULP	NO	NO	NO	NO	n/a
End of volume flag (positioning of file at volume CLOSEM time)	VF	U	U	U	U	U

TABLE F-1. DEFAULTS FOR FIT FIELDS (Contd)

Notes:

n/a	FIT field not applicable to this input/output mode.
*	Default cannot be overridden by a FILE control statement.
(1)	Buffer size can be declared on the PROGRAM statement, OPEN statement, or FILE control statement. Otherwise, CRM chooses the buffer size according to device type. Buffer is allocated on the first I/O operation and deallocated when the file is closed.
(2)	Set by parity designator in BUFFER IN or BUFFER OUT statement.
(3)	Set by PROGRAM statement, OPEN statement, or execution control statement.
(4)	Set by CYBER Record Manager.
(5)	Default can be changed on PROGRAM or OPEN statement. For formatted, NAMELIST, and list directed READ/WRITE statements, a FILE control statement can decrease but not increase the maximum record length declared on the PROGRAM statement.
(6)	Default can be overridden by a FILE control statement only if RT≠R and RT≠Z. For RT=F, FL must be a multiple of 10.
(7)	The LABEL subroutine (section 7) sets LBL=80, LT=ST, OF=R, and ULP=F.
(8)	Maximum record length equal to length of record specified in BUFFER IN or BUFFER OUT statement.
(9)	Unblocked if mass storage file; I if tape file.
(10)	Default can be overridden by FILE control statement only if RT≠U.
(11)	On a CYBER 170 Model 176, SBF must be set to NO on a FILE control statement if a level 2 or 3(LCM) variable is used in a buffer statement under NOS/BE.

†Applies to NOS and NOS/BE only.
††Applies to SCOPE 2 only.

End File

Tables F-3 and F-4 indicate the action taken when an ENDFILE statement is executed. The action depends on the record and block type, as well as the device on which the file resides.

INPUT/OUTPUT RESTRICTIONS

Meaningful results are not guaranteed in the following circumstances:

- Mixed formatted and unformatted read or write statements and buffer input/output statements on the same file (without an intervening REWIND, ENDFILE, or without encountering an end-of-file as determined by the EOF Function).
- Requesting a LENGTH function or LENGTHX call on a buffer unit before requesting a UNIT function.
- Two consecutive buffer input/output statements on the same file without the intervening execution of a UNIT function call.
- Writing formatted records on a 7-track S or L tape without specifying CM=NO on a file control statement.
- Using items in an input list after encountering end-of-file in a read.
- Attempting to write a noise record on an S or L tape. This can occur with block types K and E (and C for SCOPE 2) using record types F,D,R,T, or U with MNB <noise size.
- Sequential I/O operations REWIND, BACKSPACE, and ENDFILE on a direct access file.

COMPILE TIME INPUT/OUTPUT

The compiler expects source input files to have certain characteristics and it produces coded and binary files which must be structured in specific ways according to the operating system under which it runs. A program compiled under SCOPE 2 must be executed under control of SCOPE 2; a program compiled under other operating systems cannot be executed under SCOPE 2. Programs compiled under NOS or NOS/BE can be executed under either of these operating systems.

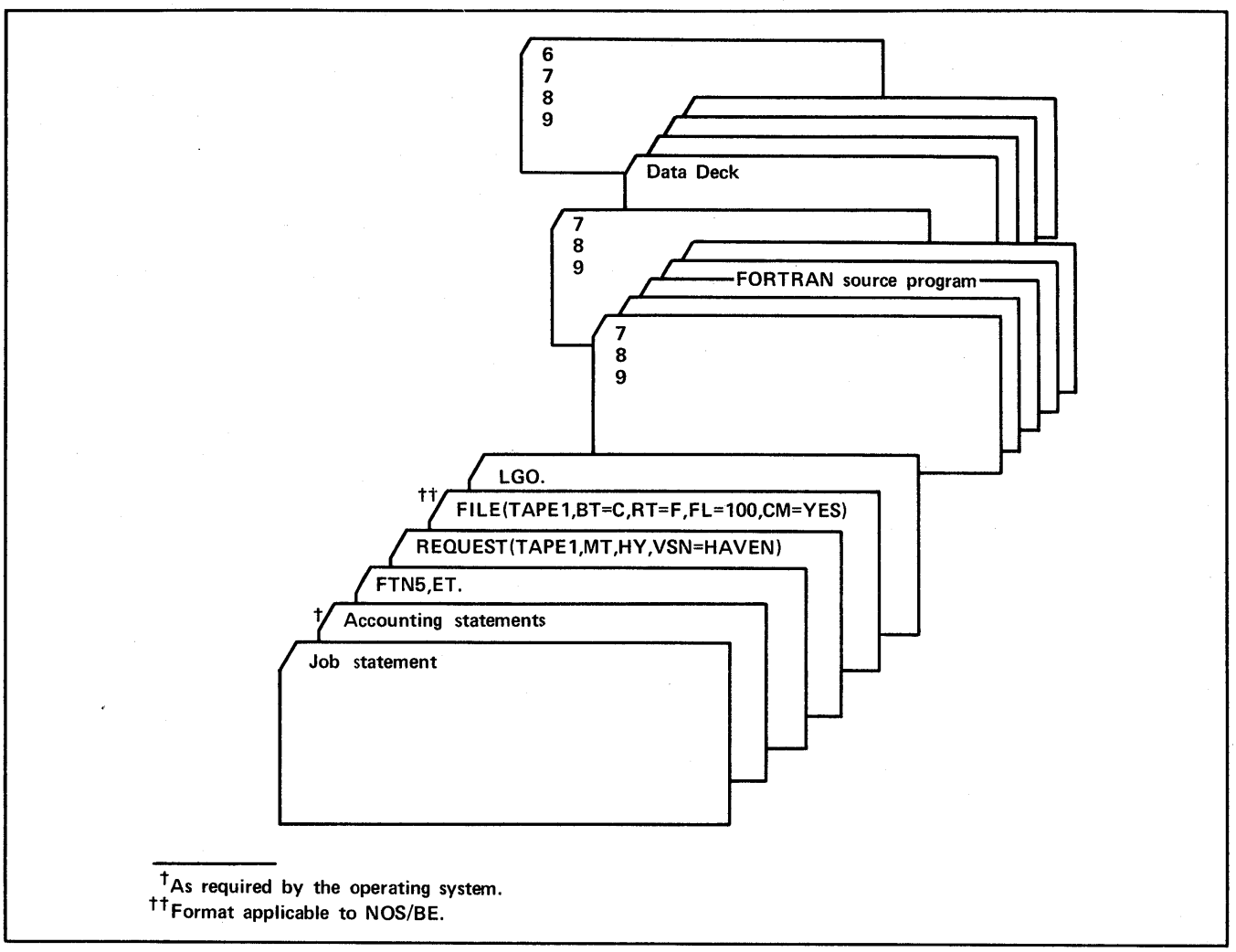


Figure F-2. FILE Control Statement Example

Under SCOPE 2, the compiler uses SCOPE 2 Record Manager for all input/output operations. However, a FILE control statement should not be used since the compiler overrides file information table settings after this control statement is processed. Under NOS and NOS/BE, the compiler makes direct calls to the operating system for input/output; CRM is not used.

SOURCE INPUT FILE STRUCTURE

A source input file must have a certain structure. Only the first 90 characters of each record are processed or reproduced in the listing output file. The characteristics are described in table F-5.

CODED OUTPUT FILE STRUCTURE

Two coded output files can be produced: the listing file and the errors file. The characteristics are described in table F-6.

BINARY OUTPUT FILE STRUCTURE

The content of the executable object code file differs, depending on the loader supported by the operating system. The characteristics are described in table F-7.

TABLE F-2. ACTION BEFORE POSITIONING FOR BACKSPACE/REWIND

Condition	Device Type	Action
Last operation was WRITE or BUFFER OUT	Mass Storage	Any unwritten blocks for the file are written. An end-of-partition is written. If record format is W, a deleted zero length record is written.
	Unlabeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written. Two file marks are written.
	Labeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted record is written. A file mark is written. A single EOF label is written. Two file marks are written.
Last operation was WRITE or BUFFER OUT††	Mass Storage	ENDFILE is issued. Any unwritten blocks for the file are written. End-of-information is written.
	Unlabeled Magnetic S or L Tape	ENDFILE is issued. Any unwritten blocks for the file are written. Two file marks are written.
	Labeled Magnetic Tape or Unlabeled System Magnetic Tape	ENDFILE is issued. Any unwritten blocks for the file are written. A tape mark is written. A single EOF label is written. Two tape marks are written.
Last operation was READ, BUFFER IN or BACKSPACE	Mass Storage	None.
	Unlabeled Magnetic Tape	None.
	Labeled Magnetic Tape	None.
No previous operation	All Devices†	REWIND request causes the file to be rewound when first referenced.
	Mass Storage††	
	Magnetic Tape††	If the file is assigned to on-line magnetic tape, a REWIND request is executed. For SCOPE 2, if the file is staged, the REWIND request has no effect. The file is staged and rewound when it is first referenced.
Previous operation was REWIND		Current REWIND is ignored.

†Applies to NOS and NOS/BE only.
††Applied to SCOPE 2 only.

TABLE F-3. ENDFILE ACTION (NOS AND NOS/BE)

Record Type	Device Type	
	S or L Tape	Other Device
W	An end-of-partition flag is written.	An end-of-partition flag is written.
	The block is terminated.	The block is terminated with a short PRU of level 0.
Other	The block is terminated.	The block is terminated with a short PRU of level 0.
	A tape mark is written.	A zero length PRU of level 17 is written.

TABLE F-4. ENDFILE ACTION (SCOPE 2)

Record Type	Blocking	
	Blocked	Unblocked
W	An end-of-partition flag is written. The block is terminated.	An end-of-partition flag is written.
Z	If C type blocking, the block is terminated. Otherwise, the block is terminated and a tape mark recovery control word is written.	A level 17 PRU is written.
S	If C type blocking, the block is terminated with a zero length PRU of level 17. Otherwise, the block is terminated and a tape mark recovery control word is written.	Not applicable.
Others on Mass Storage	The block is terminated. A tape mark recovery control word is written.	Ignored.
Others on Magnetic Tape	The block is terminated. A tape mark is written.	Not applicable.

TABLE F-5. SOURCE INPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a short or zero length PRU	Sequential (FO=SQ) unblocked
Record type	Zero-byte terminated	Control word (RT=W)
Maximum record length	158 characters	158 characters (MRL=158)
Conversion mode	Not applicable	No (CM=NO)
Label type of tape	Under operating system control	Unlabeled (LT=UL)

TABLE F-6. CODED OUTPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a short PRU	Sequential (FO=SQ) unblocked
Maximum block length	Not applicable	None
Record type	Zero-byte terminated (equivalent to Record Manager Z type)	Control word (RT=W)
Maximum record length	137 characters	137 characters
Conversion mode	Not applicable	No (CM=NO)
Tape label type	Under operating system control	Unlabeled (LT=UL)

TABLE F-7. BINARY OUTPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a zero length PRU which is then back-spaced over	Sequential (FO=SQ) unblocked
Record type	Operating system logical record (equivalent to Record Manager S type)	Control word (RT=W)
Maximum record length	None	1,310,710 characters
Conversion mode	Not applicable	No (CM=NO)
Tape label type	Under operating system control	Unlabeled (LT=U)

FUTURE SYSTEM MIGRATION GUIDELINES G

This appendix contains programming practices recommended by CDC for users of the software described in this manual. When possible, application programs based on this software should be designed and coded in conformance with these recommendations.

Two forms of guidelines are given. The general guidelines minimize application program dependence on the specific characteristics of a hardware system. The feature use guidelines ensure the easiest migration of an application program to future hardware or software systems.

GENERAL GUIDELINES

Programmers should observe the following practices to avoid hardware dependency:

- Avoid programming hardcoded constants. Manipulation of data should never depend on the occurrence of a type of data in a fixed multiple such as 6, 10, or 60.
- Do not manipulate data based on the binary representation of that data. Characters should be manipulated as characters, rather than as octal display-coded values or as 6-bit binary digits. Numbers should be manipulated as numeric data of a known type, rather than as binary patterns within a central memory word.
- Do not identify or classify information based on the location of a specific value within a specific set of central memory word bits.
- Avoid using COMPASS in application programs. COMPASS and other machine-dependent languages can complicate migration to future hardware or software systems. Migration is restricted by continued use of COMPASS for stand-alone programs, by COMPASS subroutines embedded in programs using higher-level languages, and by COMPASS owncode routines in CDC standard products. COMPASS should only be used to create part or all of an application program when the function cannot be performed in a higher-level language or when execution efficiency is more important than any other consideration.

FEATURE USE GUIDELINES

The recommendations in the remainder of this appendix ensure the easiest migration of an application program for use on future hardware or software systems. These recommendations are based on known or anticipated changes in the hardware or software system, or comply with proposed new industry standards or proposed changes to existing industry standards.

ADVANCED ACCESS METHODS

The Advanced Access Methods (AAM) offer several features within which choices must be made. The following paragraphs indicate preferred usage.

Access Methods

The recommended access methods are indexed sequential (IS), direct access (DA), and multiple index processor (MIP).

Record Types

The recommended record types are either F for fixed length records, or W for variable length records. Record length for W records is indicated in the control word; the length must be supplied by the user in the RL FIT field on a put operation and is returned to the user in RL on a get operation.

FORTRAN Usage

The following machine-independent coding practices are encouraged for a FORTRAN programmer using AAM:

- Initialize the FIT by FILExx calls or by the FILE control statement.
- Modify the FIT with STOREF calls.
- Use the FORTRAN 5 CHARACTER data type when working with character fields rather than octal values of display code characters; specify lengths of fields, records, and so forth, in characters rather than words.

BASIC ACCESS METHODS

The Basic Access Methods (BAM) offer several features within which choices must be made. The following paragraphs indicate preferred usage.

File Organizations

The recommended file organization is sequential (SQ). For files with word-addressable (WA) organization, use an accessing technique that can easily be modified to byte addresses.

Block Types

The recommended block type is C.

Record Types

The recommended record types are F for fixed length records and W for variable length records. For purely coded files that are to be listed, Z type records can be used.

Block Size

Set the Maximum Block Length (MBL) to 640 characters for mass storage files and 5120 characters for tape files.

Host Language Input/Output

Use of host language input/output statements (for example, a FORTRAN READ statement) to process BAM files is always a safe procedure. Host language statements provide appropriate default values for record type, block type, and block size. Do not use the CYBER Record Manager FORTRAN interface routines to process sequential files.

Collating Sequence

The default collating sequence or the ASCII collating sequence should be used.

FORTRAN 5

FORTRAN 5 offers users several capabilities that are processor-dependent. The use of such capabilities restrict FORTRAN 5 program migration. The following paragraphs indicate preferred usages.

Processor-dependent Values

Coding should not depend on the internal representation of data (floating-point layout, number of characters per word, and so forth). Where coding must depend on these representations, use parameter variables for processor-dependent characteristics such as the number of characters per word.

Boolean Data Types

Do not use Boolean data types and operations (SHIFT, MASK, and so forth) because they can be processor-dependent. Use type CHARACTER instead, if working with character data.

LOCF Function

Do not use the intrinsic function LOCF. For most applications, this function should not be necessary.

ENCODE and DECODE Statements

Do not use ENCODE and DECODE; use the ANSI standard internal files feature instead. ENCODE and DECODE are generally dependent on the number of characters per word.

DATE, TIME, and CLOCK Functions

Do not dismantle values returned by the DATE, TIME, and CLOCK functions; use these functions only for printing out values as a whole.

BUFFER IN and BUFFER OUT Statements

Do not use BUFFER IN and BUFFER OUT, especially when use depends on the number of characters per word.

CYBER Record Manager Interface Routines

Do not use the CYBER Record Manager interface routines for sequential files. Instead, use FORTRAN input/output statements such as READ or WRITE.

Overlays

If possible, use segmented loading instead. If overlays must be used, do not depend on such properties as reinitialization of variables when an overlay is reloaded.

LABEL Subroutine

Avoid use of the LABEL subroutine. Changes to the ANSI standard for tape labels might require changes to the interface used by this subroutine.

STATIC Memory Management and Capsule Loading

Do not use this capability unless absolutely necessary. Use of Common Memory Manager and OVCAPs is preferred.

The user must be thoroughly aware of the capsules needed to perform the types of I/O operations required. It is the user's responsibility to ensure that the capsules are loaded by explicitly specifying the appropriate STLxxx subroutine call. Only default block and record types are supported by the STLxxx subroutines. To force load nondefault block type/record type handling of capsules, the user must use the following control statement sequence:

```
FILE,LFN,...,RT=...,BT=...,...USE=...
LDSET(STAT=ifn)
```

SORT/MERGE VERSIONS 4 AND 1

Sort/Merge offers several features among which choices must be made. The following paragraphs indicate preferred usage.

Key Alignment

Ensure that SORT keys are aligned on character or word boundaries. Do not place SORT keys in arbitrary bit positions within words.

SORT and MERGE Statements

Always perform logically separated SORT and MERGE operations with separate control statements.

INDEX

COMMENT SHEET

MANUAL TITLE: FORTRAN Version 5 Reference Manual

PUBLICATION NO.: 60481300 **REVISION:** E

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ Please reply ☐ No reply necessary

CUT ALONG LINE

PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

APE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

215 Moffett Park Drive
Sunnyvale, California 94086

CUT ALONG LINE

FOLD

FOLD

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION