60497800

CONTROL DATA CORPORATION

# FORTRAN EXTENDED VERSION 4 REFERENCE MANUAL

CDC® OPERATING SYSTEMS:

NOS 1

NOS 2

NOS/BE 1

SCOPE 2

# REVISION RECORD

| Revision | Description |
|---|---|
| A (11/01/75) | Original release. |
| B (03/05/76) | This revision documents Version 4.6 of FORTRAN Extended. Features documented include CP155, Compiler Enhancements, and CP079, Math Library Upgrade. |
| C (04/15/77) | Revised to include feature F7540, CYBER 170 Model 176 Support, as well as miscellaneous technical corrections, at PSR level 446. |
| D (03/31/78) | This revision documents Version 4.7 of FORTRAN Extended. Features documented include CP091 and CP162, CRM products BAM and AAM, 191, Math Library Upgrade, CP184, Fast Overlay Loading, and 66, CYBER Interactive Debug interface. Also documented is the implementation of STATIC mode memory management, as well as miscellaneous technical changes and corrections. |
| E (07/20/79) | This revision documents Version 4.8 of FORTRAN Extended. The Post Mortem Dump facility is documented with this release, as well as numerous technical changes. |
| F (08/22/80) | This revision documents changes to Post Mortem Dump, adds the FORTRAN Interface to Common Memory Manager, and adds the STATIC option to FORTRAN Extended. Numerous technical changes are included. PSR level 524. |
| G (01/15/81) | This revision documents release of Post Mortem Dump and STATIC option under SCOPE 2. Numerous technical changes are included. PSR level 533. |
| H (08/13/82) | This revision documents numerous technical changes and corrections. PSR level 552. |
| J (06/10/83) | This revision documents numerous technical and editorial corrections. PSR level 577. |

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



Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
P. O. BOX 3492
SUNNYVALE, CALIFORNIA 94088-3492

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 |
|---|---|
| Front Cover | - |
| Inside Front Cover | E |
| Title Page | - |
| ii | J |
| iii | J |
| iv | J |
| v | J |
| vi | H |
| vii/viii | H |
| ix | H |
| x | J |
| xi | H |
| xii | H |
| 1-1 thru 1-4 | A |
| 1-5 | B |
| 1-6 thru 1-9 | A |
| 1-10 | E |
| 1-11 thru 1-17 | A |
| 2-1 | A |
| 2-2 | A |
| 2-3 | E |
| 2-4 | A |
| 2-5 | E |
| 2-6 | D |
| 2-7 thru 2-20 | A |
| 3-1 thru 3-9 | A |
| 3-10 | D |
| 3-11 | A |
| 3-12 | C |
| 3-13 | J |
| 3-14 | H |
| 3-15 | H |
| 3-16 | A |
| 3-17 | A |
| 3-18 | E |
| 3-19 | E |
| 3-20 | A |
| 3-21 | A |
| 4-1 | A |
| 4-2 | A |
| 4-3 | E |
| 4-4 | F |
| 4-5 thru 4-9 | A |
| 4-10 | E |
| 4-11 | A |
| 4-12 | A |
| 4-13 | F |
| 4-14 thru 4-16 | A |
| 5-1 | B |
| 5-2 | C |
| 5-3 | F |
| 5-4 thru 5-6 | C |
| 5-7 | A |
| 5-8 | D |
| 5-9 | E |
| 5-10 | E |
| 5-10.1 | E |
| 5-10.2 | E |

| Page | Revision |
|---|---|
| 5-11 thru 5-19 | C |
| 5-20 | J |
| 5-21 | J |
| 5-22 | F |
| 5-23 | E |
| 5-24 | D |
| 5-25 thru 5-28 | E |
| 6-1 | A |
| 6-2 | D |
| 6-3 | A |
| 6-4 | A |
| 6-5 | E |
| 6-6 thru 6-8 | A |
| 6-9 | E |
| 6-10 | C |
| 6-11 thru 6-17 | A |
| 6-18 | E |
| 6-18.1/6-18.2 | E |
| 6-19 thru 6-21 | E |
| 6-22 | A |
| 6-23 | C |
| 6-24 | D |
| 6-25 thru 6-27 | A |
| 6-28 | F |
| 6-29 | B |
| 6-30 | A |
| 6-31 | F |
| 6-32 | A |
| 6-33 | A |
| 6-34 | H |
| 6-35 thru 6-37 | A |
| 7-1 | A |
| 7-2 | D |
| 7-3 | B |
| 7-4 | J |
| 7-5 thru 7-9 | A |
| 7-10 | E |
| 7-11 | A |
| 7-12 | C |
| 7-13 thru 7-15 | A |
| 7-16 | C |
| 7-17 | A |
| 7-18 | A |
| 7-19 | E |
| 7-20 | E |
| 7-21 | G |
| 7-22 | H |
| 7-23 | D |
| 7-24 thru 7-28 | A |
| 8-1 | D |
| 8-2 | J |
| 8-3 | F |
| 8-4 | C |
| 8-5 | A |
| 8-6 | F |
| 8-6.1/8-6.2 | F |

| Page | Revision |
|---|---|
| 8-7 | G |
| 8-8 | E |
| 8-9 | D |
| 8-10 | E |
| 8-11 thru 8-16 | J |
| 8-16.1/8-16.2 | J |
| 8-17 | H |
| 8-18 | C |
| 8-19 | C |
| 8-20 | G |
| 8-21 | E |
| 8-22 | D |
| 8-23 | J |
| 8-24 | J |
| 8-25 | H |
| 8-26 | H |
| 8-26.1/8-26.2 | J |
| 8-27 | H |
| 8-28 | H |
| 8-29 | D |
| 8-30 | D |
| 8-31 | E |
| 8-32 | D |
| 8-32.1 /8-32.2 | C |
| 8-33 thru 8-35 | A |
| 8-36 | E |
| 8-37 | A |
| 8-38 | A |
| 8-39 | F |
| 8-40 | F |
| 8-41 | E |
| 8-42 | F |
| 8-43 | E |
| 8-44 | H |
| 8-45 thru 8-47 | J |
| 8-48 thru 8-51 | F |
| 8-52 | H |
| 8-53 | G |
| 8-54 | G |
| 8-55 | H |
| 8-56 | H |
| 8-56.1/8-56.2 | H |
| 8-57 thru 8-61 | F |
| 9-1 | E |
| 9-2 thru 9-4 | C |
| 9-5 | A |
| 9-6 | H |
| 9-7 | C |
| 9-8 | A |
| 9-9 | A |
| 9-10 | C |
| 9-11 thru 9-29 | A |
| 10-1 | H |
| 10-2 | G |
| 10-3 | D |
| 10-4 | E |

| Page | Revision |
|---|---|
| 10-5 thru 10-7 | J |
| 10-8 | D |
| 10-9 | G |
| 10-10 | G |
| 11-1 | C |
| 11-2 | A |
| 11-3 | D |
| 11-4 | A |
| 11-5 | F |
| 11-6 | C |
| 11-7 | B |
| 12-1 | B |
| 12-2 | B |
| 12-3 | A |
| 12-4 | B |
| 12-5 | B |
| 13-1 | A |
| 13-2 | A |
| 13-3 | F |
| 13-4 | F |
| 13-5 thru 13-8 | A |
| 13-9 | C |
| 13-10 | A |
| 13-11 | E |
| 13-12 | F |
| 13-13 | E |
| 13-14 | A |
| 13-15 | A |
| 13-16 thru 13-20 | B |
| 13-21 | C |
| 14-1 | A |
| 14-2 | A |
| 14-3 | D |
| 15-1 thru 15-3 | G |
| 16-1 | D |
| 16-2 | D |
| 16-3 | G |
| 16-4 | G |
| 16-5 | J |
| 16-6 | D |
| 16-7 | D |
| 16-8 | C |
| 16-9 | A |
| 16-10 | A |
| 16-11 | H |
| 16-12 | C |
| 16-13 | D |
| 17-1 | J |
| 17-2 | C |
| 17-3 | A |
| 17-4 | A |
| 17-5 | D |
| 17-6 | D |
| 18-1 | D |
| 18-2 thru 18-6 | A |
| 18-7 | D |
| 18-8 | D |
| 18-9 thru 18-11 | A |
| 19-1 | D |
| 19-2 | A |
| 19-3 | D |
| 19-4 thru 19-7 | A |
| 19-8 | F |
| 19-9 thru 19-11 | A |
| 19-12 | C |
| 19-13 thru 19-18 | A |
| 19-19 | E |
| 19-20 thru 19-27 | A |
| 19-28 | C |
| 19-29 thru 19-32 | A |
| 19-33 | C |
| 19-34 | A |

| Page | Revision |
|---|---|
| 20-1 | F |
| A-1 | A |
| A-2 | F |
| A-3 | A |
| B-1 thru B-4 | D |
| B-5 | E |
| B-6 | G |
| B-7 | D |
| B-8 | D |
| B-9 | F |
| B-10 | F |
| B-11 thru B-14 | D |
| B-15 | F |
| B-16 | D |
| B-17 | D |
| B-18 | F |
| B-19 | D |
| B-20 | F |
| B-21 | D |
| B-22 | F |
| B-23 thru B-26 | D |
| B-27 | F |
| B-28 | F |
| B-29 | D |
| B-30 | E |
| B-31 thru B-33 | D |
| B-34 | G |
| B-35 | D |
| B-36 | D |
| B-37 | F |
| B-38 thru B-41 | D |
| B-42 | E |
| B-43 | G |
| B-44 thru B-49 | D |
| B-50 | F |
| B-51 thru B-64 | D |
| B-65 | F |
| B-66 | D |
| B-67 | D |
| B-68 | G |
| B-69 thru B-76 | D |
| B-77 | F |
| B-78 | H |
| B-79 | D |
| B-80 | F |
| B-81 | E |
| B-82 | D |
| B-83 | F |
| B-84 | F |
| B-85 | D |
| B-86 | H |
| B-87 | F |
| B-88 | H |
| B-89 | F |
| B-90 | D |
| B-91 | D |
| B-92 | H |
| B-93 | E |
| B-94 | D |
| B-95 | F |
| B-96 | E |
| B-97 | E |
| C-1 thru C-8 | A |
| C-9 thru C-11 | C |
| D-1 | E |
| D-2 | A |
| D-3 thru D-5 | C |
| D-6 | A |
| D-7 | D |
| D-8 | A |
| E-1 thru E-6 | D |
| Index-1 | H |

| Page | Revision |
|---|---|
| Index-2 | J |
| Index-3 | H |
| Index-4 | H |
| Index-5 | J |
| Index-6 | J |
| Comment Sheet | J |
| Mailer | - |
| Back Cover | - |

# PREFACE

This manual describes the FORTRAN Extended 4.8 language. FORTRAN Extended is designed to comply with American National Standards Institute FORTRAN language, as described in X3.9-1966. It is assumed the reader has knowledge of an existing FORTRAN language and is familiar with the computer system on which the language is used.

The FORTRAN Extended compiler operates in conjunction with the COMPASS 3 assembly language processor under control of the following operating systems:

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

NOS 2 for the CDC® CYBER 170 Series, CYBER 70 Models 71, 72, 73, 74, and 6000 Series Computer Systems

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

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

All references in this manual to NOS 1 refer to both NOS 1 and NOS 2.

Due to capsule loading, relocatable binaries compiled by versions of FORTRAN Extended prior to version 4.7 cannot be run with CRM BAM 1.5 or AAM 2; they must be recompiled.

Control Data extensions to the FORTRAN language are indicated by shading. Example programs or parts of programs are shaded in their entirety if they contain lines using extensions to the ANSI standard (unless the only such extension is the PROGRAM statement). Shading is used only in sections 1 through 8, which contain the specification of the FORTRAN Extended language; later sections describe the implementation of these specifications and shading is not used.

Extended memory for the CYBER 170 Model 176 is large central memory (LCM) or large central memory extended (LCME). Extended memory for the CYBER 170 800 Series Computer Systems is unified extended memory (UEM). Extended memory for all other NOS or NOS/BE computer systems is extended core storage (ECS) or extended semiconductor 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. These publications are listed alphabetically and grouped according to their importance to the FORTRAN user. The NOS 1, NOS 2, and NOS/BE 1 manual abstracts are pocket-sized manuals containing brief descriptions of the contents and intended audience of all operating system and product set manuals. The abstracts manuals can be useful in determining which manuals are of greatest interest to a particular user.

The Software Publications Release History is a guide for determining which revision level of software documentation corresponds to the Programming System Report (PSR) level of installed site software.

The following publications are of primary interest:

| Publication | Publication Number | NOS 1 | NOS 2 | NOS/BE 1 | SCOPE 2 |
|---|---|---|---|---|---|
| FORTRAN Common Library Mathematical Routines Reference Manual | 60498200 | X | X | X | X |
| FORTRAN Extended Version 4 DEBUG User's Guide | 60498000 | X | X | X | X |
| FORTRAN Extended Version 4 User's Guide | 60499700 | X | | X | |
| NOS Version 1 Reference Manual Volume 1 of 2 | 60435400 | X | | | |
| NOS Version 2 Reference Set Volume 3, System Commands | 60459680 | | X | | |
| NOS/BE Version 1 Reference Manual | 60493800 | | | X | |
| SCOPE Version 2 Reference Manual | 60342600 | | | | X |

The following publications are of secondary interest:

| Publication | Publication Number | NOS 1 | NOS 2 | NOS/BE 1 | SCOPE 2 |
|---|---|---|---|---|---|
| Common Memory Manager Version 1 Reference Manual | 60499200 | X | X | X | X |
| COMPASS Version 3 Reference Manual | 60492600 | X | X | X | X |
| CYBER Interactive Debug Version 1 Reference Manual | 60481400 | X | X | X | |
| CYBER Interactive Debug Version 1 Guide for Users of FORTRAN Extended Version 4 | 60482700 | X | X | X | |
| CYBER Loader Version 1 Reference Manual | 60429800 | X | X | X | |
| CYBER Record Manager Advanced Access Methods Version 2 Reference Manual | 60499300 | X | X | X | |
| CYBER Record Manager Advanced Access Methods Version 2 User's Guide | 60499400 | X | X | X | |
| CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual | 60495700 | X | X | X | |
| CYBER Record Manager Basic Access Methods Version 1.5 User's Guide | 60495800 | X | X | X | |

| Publication | Publication Number | NOS 1 | NOS 2 | NOS/BE 1 | SCOPE 2 |
|---|---|---|---|---|---|
| DMS-170 DDL Version 3 Reference Manual Volume 1: Schema Definition for Use With: COBOL FORTRAN Query Update | 60481900 | X | X | X | |
| FORTRAN Data Base Facility Version 1 Reference Manual | 60482200 | X | X | X | |
| INTERCOM Interactive Guide for Users of FORTRAN Extended | 60495000 | | | X | |
| INTERCOM Version 5 Reference Manual | 60455010 | | | X | |
| Loader Version 1 User's Guide | 60482300 | | X | | |
| Network Products Interactive Facility Version 1 Reference Manual | 60455250 | X | | | |
| NOS Version 1 Diagnostic Index | 60455720 | X | | | |
| NOS Version 1 Manual Abstracts | 84000420 | X | | | |
| NOS Version 2 Diagnostic Index | 60459390 | | X | | |
| NOS Version 2 Manual Abstracts | 60485500 | | X | | |
| NOS Version 2 Reference Set Volume 1, Introduction to Interactive Usage | 60459660 | | X | | |
| NOS/BE Version 1 Diagnostic Index | 60456490 | | | X | |
| NOS/BE Version 1 Manual Abstracts | 84000470 | | | X | |
| SCOPE Version 2 Loader Reference Manual | 60454780 | | | | X |
| SCOPE Version 2 Record Manager Reference Manual | 60495700 | | | | X |
| Software Publications Release History | 60481000 | X | X | X | X |
| Sort/Merge Versions 4 and 1 Reference Manual | 60497500 | X | X | X | X |

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

# FORTRAN LANGUAGE ELEMENTS 1

A FORTRAN program contains executable and non-executable statements. Executable statements specify actions the program is to take, and non-executable statements describe characteristics of operands, statement functions, arrangement of data, and format of data.

## CODING FORTRAN STATEMENTS

The FORTRAN source program is written on the coding form illustrated in figure 1-1. Each line on the coding form represents an 80-column source line (terminal line or card image). The FORTRAN character set is used to code statements.

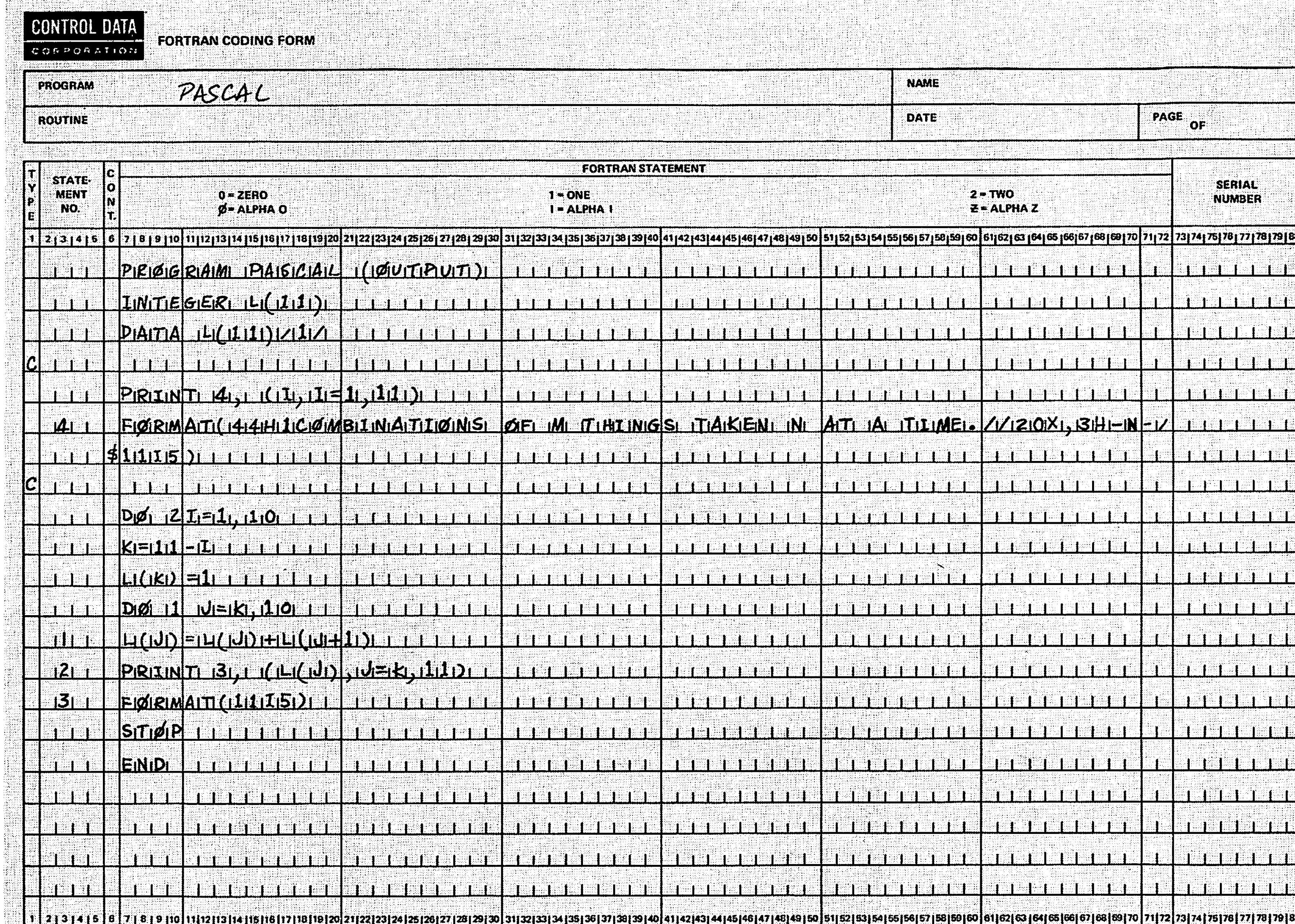
### FORTRAN CHARACTER SET

| | | |
|---|---|---|
| Alphabetic | A to Z | |
| Numeric | 0 to 9 | |
| Special | = equal | ) right parenthesis |
| | + plus | , comma |
| | - minus | . decimal point |
| | * asterisk | $ dollar sign |
| | / slash | blank |
| | ( left parenthesis | ≠ or " quote |

In addition, any character (Appendix A) may be used in Hollerith constants and in comments. Blanks are not significant except in Hollerith fields.

### COLUMN USAGE

| | | |
|---|---|---|
| Column 1 | C or $ or * indicates comment line | |
| Columns 1-2 | C$ indicates a debug directive if in DEBUG mode. | |
| Columns 1-2 | C/ indicates a list directive. | |
| Columns 1-5 | Statement label. | |
| Column 6 | Any character other than blank or zero denotes continuation; does not apply to comment lines or list directives. A debug continuation line must contain C$ in columns 1-2. | |
| Columns 7-72 | Statement. | |
| Columns 73-80 | Identification field, not processed by compiler. | Can contain information for debug AREA directive. |



CONTROL DATA CORPORATION FORTRAN CODING FORM

PROGRAM PASCAL

ROUTINE

NAME

DATE

PAGE OF

```
      PROGRAM PASCAL (OUTPUT)
      INTEGER L(11)
      DATA L(11)/1/
C
      PRINT 4, (I, I=1,11)
    4 FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME./120X,3HN-/
     $11I5)
C
      DO 2 I=1,10
      K=11-I
      L(K)=1
      DO 1 J=K,10
    1 L(J)=L(J)+L(J+1)
    2 PRINT 3, (L(J), J=K,11)
    3 FORMAT(11I5)
      STOP
      END
```

Figure 1-1. Program PASCAL

## COMMENTS

In column 1 a C, *, or $ indicates a comment line. Comments do not affect the program; they can be written in column 2 to 80 and can be placed anywhere within the program. If a comment occupies more than one line, each line must begin with C, *, or $ in column 1. In a comment line a character in column 6 is not recognized as a continuation character. Comments can appear between continuation lines; they do not interrupt the statement continuation.

Comment lines following an END line are listed at the beginning of the next program unit unless the END line is continued.

## STATEMENT LABELS

A statement label (any 1- to 5-digit integer) uniquely identifies a statement so it can be referenced by another statement. Statements that will not be referenced do not need labels. Blanks and leading zeros are not significant. Labels need not occur in numerical order; however, a given label must not be used more than once in the same program unit. A label is known only in the program unit containing it; it 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. A label on a continuation line is ignored.

## CONTINUATION

Statements are coded in columns 7-72. If a statement is longer than 66 columns, 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. Column 1 can contain any character other than C, *, or $; columns 2, 3, 4, and 5 can contain any character. Any statement except a comment or a list directive can be continued, including the END statement.

## COLUMNS 73-80

Any information can appear in columns 73-80 because they are not part of the statement. Entries in these columns are copied to the source program listing. They are generally used to order the lines in a deck, but can contain information for DEBUG AREA processing.

# STATEMENT SEPARATOR

Several statements can be written on one line if they are separated by the special character $. Each statement following a $ sign is treated as a separate statement. For example:

```
ACUM=24.$I=0 $ IDIFF=1970-1626
```

is the same as

```
ACUM = 24.
I = 0
IDIFF = 1970-1626
```

$ can be used following any statement except FORMAT statements or list and debug directives. The statement following $ cannot be labeled; the information following $ is treated exactly as if it were in column 7 on the next line.

## BLANK LINES

Blank lines can be used freely between statements to produce blank lines on the source listing. Unlike a comment line, a blank line interrupts statement continuation, and the line following the blank line is the beginning of a new statement. This line can optionally have the form of a continuation line.

## DATA

No restrictions are imposed on the format of data read by the source program. Data input on cards is limited to 80 characters per card, but a record can span more than one card. The maximum length in characters for formatted, list directed, and NAMELIST records must agree with the length, r, specified in the PROGRAM statement. If r is not specified, a default value of 150 is used.

# ORDERING OF STATEMENTS

The following table shows the general form of a FORTRAN program unit. Statements within a group can appear in any order, but groups must be ordered as shown. Comment lines can appear anywhere within the program.

STATEMENTS

<table>
<tr><td>1</td><td colspan="4">OVERLAY</td></tr>
<tr><td>2</td><td colspan="4">PROGRAM*<br>FUNCTION*<br>SUBROUTINE*<br>BLOCK DATA</td></tr>
<tr><td>3</td><td colspan="4">IMPLICIT</td></tr>
<tr><td>4</td><td colspan="3">type<br>COMMON<br>DIMENSION<br>EQUIVALENCE<br>EXTERNAL*<br>LEVEL</td><td rowspan="3">*FORMAT</td></tr>
<tr><td>5</td><td>Statement function* definitions</td><td rowspan="2">N†A*MELIST</td><td rowspan="2">DATA</td></tr>
<tr><td>6</td><td>ENTRY*<br>Executable statements*</td></tr>
<tr><td>7</td><td colspan="4">END</td></tr>
</table>

*Not allowed in BLOCK DATA Subprograms
†Namelist group name must be defined before it is used

# CONSTANTS

A constant is a fixed quantity. The seven types of constants are: integer, real, double precision, complex, octal, Hollerith, and logical.

## INTEGER CONSTANT

| $n_1 n_2 \cdots n_m$ |
|---|

n is a decimal digit (0-9)

$1 \leqslant m \leqslant 18$

Examples:

237 -74 +136772 0 -0024

An integer constant is a string of 1-18 decimal digits written without a decimal point. It may be positive, negative or zero. If the integer is positive, the plus sign may 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).

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, or as an index in a DO statement or implied DO, the maximum value is $2^{17}-1$ ($2^{17}-1$ = 131 071), and the minimum is 1.

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 also should be in this range. If an integer constant exceeding this range is used, a fatal diagnostic is issued. Any other cases are not diagnosed, and the results are unpredictable. For integer addition and subtraction (where both operands are integers), the full 60-bit word is used.

When values are converted from real to integer or from integer to real (in an expression or assignment statement), 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 CONSTANT

| n.n | .n | n. | n.nE±s | .nE±s | n.E±s | nE±s |
|---|---|---|---|---|---|---|

| | |
|---|---|
| n | Coefficient $\leqslant$ 15 decimal digits |
| E±s | Exponent (base 10) |

A real constant consists of a string of decimal digits written with a decimal point or an exponent, or both. Commas are not allowed. If the exponet is positive, the plus sign is optional.

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.    23798.14    .5    - .72    42.E1    700.E-2
```

Examples of invalid real constants:

| | |
|---|---|
| `3,50.` | (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 sign may be omitted if the exponent is positive, but it must be present if the exponent is negative.

Examples:

| | |
|---|---|
| `42.E1` | ($42. \times 10^1 = 420.$) |
| `.00028E+5` | ($.00028 \times 10^5 = 28.$) |
| `6.205E12` | ($6.205 \times 10^{12} = 6205000000000.$) |
| `8.0E+6` | ($8. \times 10^6 = 8000000.$) |
| `700.E-2` | ($700. \times 10^{-2} = 7.$) |
| `7E20` | ($7. \times 10^{20} = 70\ 000\ 000\ 000\ 000\ 000\ 0000.$) |

Example of invalid real constants:

| | |
|---|---|
| `7.2E3.4` | exponent not an integer |

## DOUBLE PRECISION CONSTANT

| n.nD±s | .nD±s | n.D±s | nD±s |
|---|---|---|---|

| | |
|---|---|
| n | Coefficient |
| D±s | Exponent (base 10) |

Double precision constants are written in the same way as real constants except the exponent is specified by the letter D instead of E. Double precision values are represented internally by two computer words, giving extra precision. A double precision constant is accurate to approximately 29 decimal digits. If the exponent is positive, the plus sign is optional.

Examples:

| | |
|---|---|
| `5.834D2` | ($5.834 \times 10^2 = 583.4$) |
| `14.D-5` | ($14. \times 10^{-5} = .00014$) |
| `9.2D03` | ($9.2 \times 10^3 = 9200.$) |
| `-7.D2` | ($-7. \times 10^2 = -700.$) |
| `3120D4` | ($3120. \times 10^4 = 31200000.$) |

Examples of invalid double precision constants:

| | |
|---|---|
| `7.2D` | exponent missing |
| `D5` | exponent alone not allowed |
| `2,1.3D2` | comma illegal |
| `3.141592653589793238462643383279` | D and exponent missing |

## COMPLEX CONSTANT

$\boxed{\text{(r1, r2)}}$

| | |
|---|---|
| r1 | Real part |
| r2 | Imaginary part |

Complex constants are written as a pair of real constants separated by a comma and enclosed in parentheses.

| FORTRAN Coding | Complex Number | |
|---|---|---|
| `(1., 7.54)` | 1. + 7.54i | $i = \sqrt{-1}$ |
| `(-2.1E1, 3.24)` | -21. + 3.24i | |
| `(4.0, 5.0)` | 4.0 + 5.0i | |
| `(0., -1.)` | 0.0 - 1.0i | |

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 may be preceded by a plus or minus sign. Complex values are represented internally by two consecutive computer words.

Both parts of complex constants must be real; they may not be integer.

Examples of invalid complex constants:

| | |
|---|---|
| `(275, 3.24)` | 275 is an integer |
| `(12.7D-4 16.1)` | comma missing and double precision not allowed |
| `4.7E+2,1.942` | parentheses missing |
| `(0,0)` | 0 is an integer |

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 (see Appendix D) even when this range is not exceeded.

## OCTAL CONSTANT

$\boxed{n_1 \ldots n_m B}$

n is an octal digit, 0 through 7. $1 \leq m \leq 20$ octal digits

An octal constant consists of 1 to 20 octal digits suffixed with the letter B.

Examples:

```
777777B

52525252B

500127345B
```

Invalid octal constants:

| | |
|---|---|
| `892777B` | 8 and 9 are non-octal digits |
| `7700000000777775252525 2B` | exceeds 20 digits |
| `O7766` | O not allowed |

An octal constant must not exceed 20 digits nor contain a non-octal digit. If it does, a fatal compiler diagnostic is printed. When fewer than 20 octal digits are specified, the digits are right justified and zero filled. Octal constants can be used anywhere integer constants can be used, except that they cannot be used as statement labels or statement label references, in a FORMAT statement, or as the character count when a Hollerith constant is specified.

They can be used in DO statements, expressions, and DATA statements, and as dimension specifications.

Examples:

| | |
|---|---|
| `BAT = (I*5252B) .OR. JAY` | masking expression |
| `J = MAXO (I,1000B,J,K+40B)` | octal constant used as parameter in function |
| `NAME = I .AND. 77700000B` | masking expression |
| `J = (5252B + N)/K` | arithmetic expression |
| `DIMENSION BUF(1000B)` | dimension specification |

When an octal constant is used in an expression, it assumes the type of the dominant operand of the expression (Table 2-1, section 2).

## HOLLERITH CONSTANT

```
nHf     nLf
nRf     ≠f≠
```

| | |
|---|---|
| n | Unsigned decimal integer representing number of characters in string including blanks; must be greater than zero. |
| f | String of characters; must contain at least one character |
| ≠ | String delimiter |
| H | Left justified with blank fill |
| L | Left justified with binary zero fill |
| R | Right justified with binary zero fill |

A Hollerith constant has two forms: one is an unsigned decimal integer followed by the letter H, L, or R and a string of characters; the other is a ≠ delimited string.

Hollerith constants can be used in DATA statements, as arguments in subroutine calls or function references, and in expressions. In an expression, they are limited to 10 characters; and in a DATA statement they should be limited to 10 characters (see section 3). If a Hollerith constant is used as an operand of an arithmetic operation, an informative diagnostic is given. If a Hollerith constant is used as an argument in a subprogram call, it is followed by a zero word.

The Hollerith specification in a FORMAT statement (see section 6) is not the same as a Hollerith constant.

Example:

```
      PROGRAM HOLL (OUTPUT)
      I = 6HABCDEF
      J = 6LABCDEF
      K = 6RABCDEF
      L = ≠ABCDEF≠
      PRINT 1, I,I,J,J,K,K,L,L
    1 FORMAT (O24,A15)
      STOP
      END
```

| Stored Internally: | Display Code: | |
|---|---|---|
| 01020304050655555555 | ABCDEF | H format |
| 01020304050600000000 | ABCDEF:::: | L format |
| 00000000010203040506 | ::::ABCDEF | R format |
| 01020304050655555555 | ABCDEF | ≠ format |

**nHf and ≠f≠**

These two forms produce left-justified display code constants with 10 characters per word. If the string length is not a multiple of 10, the final word is blank filled.

nHf Examples:

```
18HTHIS IS A CONSTANT

7HTHE END

19HRESULT NUMBER THREE
```

≠f≠ Examples:

```
      IF(V.EQ.≠YES≠) Y=Y+1.

      PRINT 1, ≠ SQRT = ≠, SQRT(4.)
1     FORMAT (A10,F10.2)

      PRINT 2, ≠ TEST PASSED ≠
2     FORMAT (2A10)

      INTEGER LINE(7), N1THRU9
      LOGICAL NEWPAGE
      IF (NEWPAGE) LINE(7) = ≠PAGE 0≠+ N1 THRU 9
```

#### nRf and nLf

These two forms produce display code constants with 10 characters per word. If the string length is not a multiple of 10, the final word is zero-filled, and justified; nRf indicates right justification and nLf indicates left justification.

nRf Example:

```
IVAL (I) = 1RA
```

nLf Example:

```
INDEX (I) = 3LLGO
```

### LOGICAL CONSTANT

A logical constant takes the forms:

.TRUE. or .T. representing the value true

.FALSE. or .F. representing the value false

The decimal points are part of the constant and must appear.

Examples:

```
LOGICAL X1, X2
.
.
.
X1 = .TRUE.
X2 = .FALSE.
```

## VARIABLES

A variable represents a quantity whose value can be varied; this value can be changed repeatedly during program execution. Variables are identified by a symbolic name of one to 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 can have its type specified in a type statement (see section 3) as integer, real, double precision, complex, or logical. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer and any other letter implies type real, unless an IMPLICIT statement (see section 3) is used to change this normal implicit type.

Example:

```
IMPLICIT DOUBLE PRECISION (A)
COMPLEX ALPHA
•
•
•
APPLE=ORANGES+PEARS
```

An explicit declaration overrides an IMPLICIT declaration. Therefore, ALPHA is type complex; APPLE is type double precision.

Default typing of variables:

| | |
|---|---|
| A-H,O-Z | Real |
| I-N | Integer |

## INTEGER VARIABLES

An integer variable is a variable that is typed explicitly or implicitly as described under Variables.

The value range is - $(2^{59}-1)$ to $2^{59}-1$. When an integer variable is used as a subscript, the maximum value is $2^{17}-1$. The resulting absolute value of conversion from integer to real, or real to integer must be less than $2^{48}$. The operands, as well as the result, of an integer multiplication or division must be less than $2^{48}$ in absolute value. If any of these restrictions are violated, the results are unpredictable. For integer addition and subtraction, the full 60-bit word is used; the resulting absolute value must be less than $2^{59}$.

See section 4 for restrictions or integers used in DO statements.

An integer variable occupies one word of memory.

Examples:

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

## REAL VARIABLES

A real variable is a variable that is typed explicitly or implicitly as described under Variables.

The value range is $10^{-293}$ to $10^{+322}$ with approximately 14 significant digits of precision. A real variable occupies one word of storage.

Examples:

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

### DOUBLE PRECISION VARIABLES

Double precision variables must be typed by a type declaration. The value of a double precision variable can range from $10^{-293}$ to $10^{+322}$ with approximately 29 significant digits of precision.

Double precision variables occupy two consecutive words of memory. 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

Complex variables must be typed by a type declaration. A complex variable occupies two words of memory; 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 ZERA,MU,LAMBDA
```

### LOGICAL VARIABLES

Logical variables must be typed by a type declaration. A logical variable has the value true or false and occupies one word of memory.

Example:

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

## ARRAYS

A FORTRAN array is a set of elements identified by a single name composed of one to seven letters and digits beginning 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 (see Variables in this section). The array name and its dimensions must be declared in a DIMENSION or COMMON statement or a type declaration. Arrays can have one, two, or three dimensions.

The number of dimensions in the array is indicated by the number of subscripts in the declaration.

| | |
|---|---|
| DIMENSION STOR(6) | declares a one-dimensional array of six elements |

| | |
|---|---|
| REAL STOR(3,7) | declares a two-dimensional array of three rows and seven columns |
| LOGICAL STOR(6,6,3) | declares a three-dimensional array of six rows, six columns and three planes |

The entire array may be referenced by the unsubscripted array name when it is used as an item in an input/output list, as an actual parameter, or in a DATA statement. In any other context, only the first element of the array is implied by the unsubscripted array name.

Example 1:

The array N consists of six values in the order: 10, 55, 11, 72, 91, 7

| | |
|---|---|
| N(1) | value 10 |
| N(2) | value 55 |
| N(3) | value 11 |
| N(4) | value 72 |
| N(5) | value 91 |
| N(6) | value 7 |

Example 2:

The two-dimensional array TABLE (4,3) has four rows and three columns.

| | Column 1 | Column 2 | Column 3 |
|---|---|---|---|
| Row 1 | 44 | 10 | 105 |
| Row 2 | 72 | 20 | 200 |
| Row 3 | 3 | 11 | 30 |
| Row 4 | 91 | 76 | 714 |

To refer to the number in row two, column three write TABLE(2,3).

TABLE(3,3) = 30　　TABLE(1,1) = 44　　TABLE(4,1) = 91

TABLE(4,4) would be outside the bounds of the array and results are unpredictable.

Example 3:

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA (X,12)
      CALL IOTA (Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (* ARRAY X = *,12E9.1,5X,*ARRAY Y = *6E9.1)
      STOP
      END
```

The program declares and references two arrays; X is a two-dimensional array of 12 elements and Y is a one-dimensional array of six elements.

## SUBSCRIPTS

A subscript indicates the position of a particular element in an array. A subscript consists of a pair of parentheses enclosing from one to three subscript expressions which are separated by commas. The subscript follows the array name. A subscript expression can be any valid arithmetic expression. If the value of the expression is not integer, it is truncated to integer.

If the number of subscript expressions is less than the number of declared dimensions, the compiler assumes the omitted subscripts have a value of one. The number of subscript expressions in a reference must not exceed the number of declared dimensions.

The value of a subscript must never be zero or negative. It should be less than or equal to the product of the declared dimensions, or the reference will be outside the array. If the reference is outside the bounds of the array, results are unpredictable.

The amount of storage allocated to arrays is discussed under DIMENSION declarations in section 3.

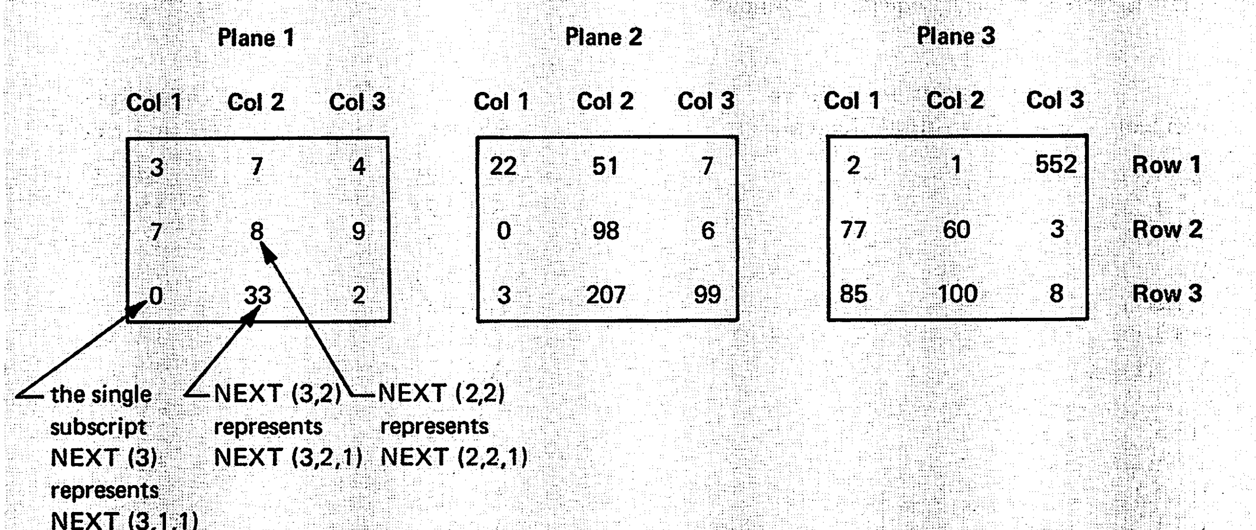
Valid subscript forms:

```
A(I,K)
B(I+2,J-3,6*K+2)
LAST(6)
ARAYD(1,3,2)
STRING(3*K*ITEM+3)
```

Invalid subscript forms:

| | |
|---|---|
| ATLAS(0) | zero subscript causes a reference outside of the array |
| D(1 .GE. K) | relational or logical expression illegal |
| A(,I) or A(I,,K) | commas can only be used to separate adjacent subscript expressions |

Example:



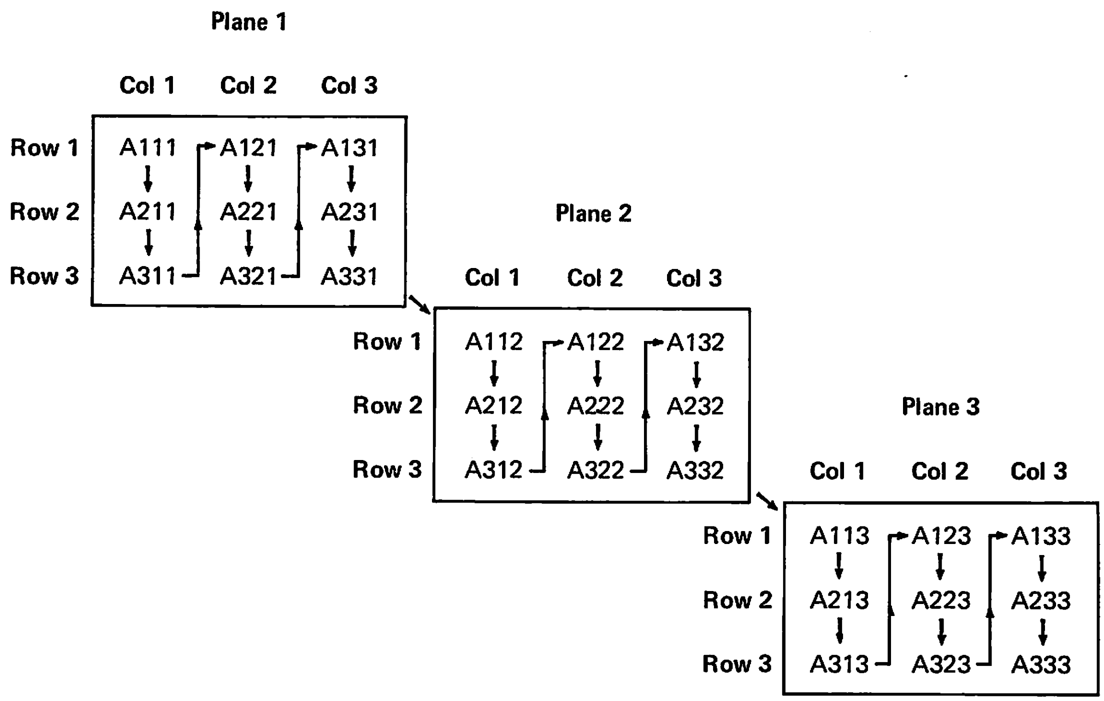

In the three-dimensional array NEXT when only one or two subscripts are shown, the remaining subscripts are assumed to be one.

## ARRAY STRUCTURE

Arrays are stored in ascending locations: the value of the first subscript increases most rapidly, and the value of the last increases least rapidly.

Example:

In an array declared as A(3,3,3), the elements of the array are stored by columns in ascending locations.




The array is stored in linear sequence as follows:

| Element | Location Relative to first Element | Element | Location Relative to first Element |
|---|---|---|---|
| A(1,1,1) | 0 | A(3,2,2) | 14 |
| A(2,1,1) | 1 | A(1,3,2) | 15 |
| A(3,1,1) | 2 | A(2,3,2) | 16 |
| A(1,2,1) | 3 | A(3,3,2) | 17 |
| A(2,2,1) | 4 | A(1,1,3) | 18 |
| A(3,2,1) | 5 | A(2,1,3) | 19 |
| A(1,3,1) | 6 | A(3,1,3) | 20 |
| A(2,3,1) | 7 | A(1,2,3) | 21 |
| A(3,3,1) | 8 | A(2,2,3) | 22 |
| A(1,1,2) | 9 | A(3,2,3) | 23 |
| A(2,1,2) | 10 | A(1,3,3) | 24 |
| A(3,1,2) | 11 | A(2,3,3) | 25 |
| A(1,2,2) | 12 | A(3,3,3) | 26 |
| A(2,2,2) | 13 | | |

To find the location of an element in the linear sequence of storage locations the following method can be used:

| Number of Dimensions | Array Dimension | Subscript | Location of Element Relative to Starting Location |
|---|---|---|---|
| 1 | ALPHA(K) | ALPHA(k) | (k-1)XE |
| 2 | ALPHA(K,M) | ALPHA(k,m) | (k-1+KX(m-1) )XE |
| 3 | ALPHA(K,M,N) | ALPHA(k,m,n) | (k-1+KX(m-1+MX(n-1)))XE |

K, M, and N are dimensions of the array.

k,m, and n are the subscript expression values of the array.

1 is subtracted from each subscript value because the subscript starts with 1, not 0.

E is length of the element. For real, logical, and integer arrays, E = 1. For complex and double precision arrays, E = 2.

Examples:

| | Subscript | Location of Element Relative to Starting Location |
|---|---|---|
| INTEGER ALPHA (3) | ALPHA(2) | (2-1)X1 = 1 |
| REAL ALPHA (3,3) | ALPHA(3,1) | (3-1+3X(1-1))X1 = 2 |
| COMPLEX ALPHA (3,3,3) | ALPHA(3,2,1) | (3-1+3X(2-1+3X(1-1)))X2 = 10 |

# EXPRESSIONS AND ASSIGNMENT STATEMENTS 2

## EXPRESSIONS

FORTRAN expressions are arithmetic, masking, logical and relational. Arithmetic and masking expressions yield numeric values, and logical and relational expressions yield truth values.

### ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of unsigned 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.

The FORTRAN arithmetic operators are:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| ** | exponentiation |

An arithmetic expression may consist of a single constant, variable, array element, or function reference. If X is an expression, then (X) is an expression. If X and Y are expressions, then the following are expressions:

```
X + Y
X*Y
-X
+ X
X-Y
X/Y
X**Y
```

All operations must be specified explicitly. For example, to multiply two variables A and B, the expression A*B must be used. AB, (A)(B), or A.B will not result in multiplication.

| Expression | Value |
|---|---|
| `3.78542` | Real constant 3.78542 |
| `A(2*J)` | Array element A (2*J) |
| `BILL` | Variable BILL |
| `SQRT(5.0)` | $\sqrt{5.}$ |
| `A+B` | Sum of the values A and B |
| `C*D/E` | Product of C times D divided by E |
| `J**I` | Value of J raised to the power of I |
| `(200 - 50)*2` | 300 |

## EVALUATION OF EXPRESSIONS

The sequence in which an expression is evaluated is governed by the following rules, listed in descending precedence:

1. References to external functions are evaluated.

2. Arithmetic statement functions and intrinsic functions are expanded.

3. Subexpressions delimited by parentheses are evaluated, beginning with the innermost subexpressions.

4. Subexpressions defined by arithmetic, relational, and logical operators are evaluated according to the following precedence hierarchy:

| | |
|---|---|
| ** | (exponentiation) |
| / * | (division or multiplication) |
| + − | (addition or subtraction) |
| .GT. .GE. .LT. .LE. .EQ. .NE. | (relationals) |
| .NOT. | (logical) |
| .AND. | (logical) |
| .OR. | (logical) |

5. Subexpressions containing operators of equal precedence are evaluated from left to right. However, individual operations that are mathematically associative and/or commutative may be reordered by the compiler to perform optimizations such as removal of repeated subexpressions or improvement of functional unit usage. The evaluation of the expression A/B*C is guaranteed to algebraically equal AC ÷ B, not A ÷ BC, but the specific order of evaluation here is indeterminate. Subexpressions containing integer divisions are not reordered within the * / precedence level because the truncation resulting from an integer division renders these operations non-associative.

Unary addition and subtraction are treated as follows:

| | |
|---|---|
| +n | the same as n |
| -n | negate n |

An array element (a subscripted variable) used in an expression requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the arguments or subscripts.

The evaluation of an expression having any of the following conditions is undefined:

Negative-value quantity raised to a real, double precision, or complex exponent

Zero-value quantity raised to a zero-value exponent

Infinite or indefinite operand (Appendix D)

Element for which a value is not mathematically defined, such as division by zero

If the error traceback option (T) is selected on the FTN control statement (section 10), the first three conditions produce informative diagnostics during execution. If the traceback option is not selected, a mode error message is printed (Appendix D).

In the case of invalid exponentiation, a diagnostic might be issued by one of the library routines ALOG, EXP, or DEXP when the exponent is real, complex, or double precision, and the base is integer, real or double precision.

Two operators must not be used together. A*-B and Z/+X are not allowed. However, a unary + or - can be separated from another operator in an expression by using parentheses. For example,

| | |
|---|---|
| `A*(-B)` and `Z/(+X)` | Valid expressions |
| `B*-A` and `X/-Y*Z` | Invalid expressions |

Each left parenthesis must have a corresponding right parenthesis.

Example:

| | |
|---|---|
| `(F + (X * Y)` | Incorrect, right parenthesis missing |
| `(F + (X * Y))` | Correct |

Examples:

In the expression

A-B*C

B is multiplied by C, and the product is subtracted from A.

The expression A/B-C*D**E is evaluated as follows:

D is raised to the power of E.

A is divided by B.

C is multiplied by the result of D**E.

The product of C*D**E is subtracted from the quotient of A divided by B.

The expression -A**C is evaluated as 0-A**C; A is first raised to the power of C and the result is then subtracted from zero.

The expression A*B*C may be evaluated as ((A*B)*C), ((A*C)*B) or (A*(B*C)), since the operator * is associative.

The expression A**B**C is evaluated as ((A**B)**C), since the operator ** is not associative.

Dividing an integer by another integer yields a truncated result; 11/3 produces the result 3. Therefore, when an integer expression is evaluated from left to right, J/K*I may give a different result than I*J/K.

Example:

```
I = 4     J = 3     K = 2

J/K*I        I*J/K

3/2*4 = 4    4*3/2 = 6
```

An integer divided by an integer of larger magnitude yields the result 0.

Example:

```
N = 24     M = 27     K = 2

N/M*K

24/27*2 = 0
```

Examples of valid expressions:

```
A

3.14159

B + 16.427

(XBAR +(B(I,J+I,K) /3.0))

-(C + DELTA * AERO)
```

```
(-B - SQRT(B**2-(4*A*C)))/(2.0*A)

GROSS - (TAX*0.04)

TEMP + V(M,AMAX1(A,B))*Y**C/ (H-FACT(K+3))
```

## TYPE OF ARITHMETIC EXPRESSIONS

An arithmetic expression may be of type integer, real, double precision, or complex. The order of dominance from highest to lowest is as follows:

Complex

Double Precision

Real

Integer

Table 2-1. Mixed Type Arithmetic Expressions with + – * / Operators

| 1st operand \ 2nd operand | Integer | Real | Double Precision | Complex | Typeless Operand |
|---|---|---|---|---|---|
| Integer | Integer | Real | Double Precision | Complex | Integer |
| Real | Real | Real | Double Precision | Complex | Real |
| Double Precision | Double Precision | Double Precision | Double Precision | Complex | Double Precision |
| Complex | Complex | Complex | Complex | Complex | Complex |
| Typeless Operand | Integer | Real | Double Precision | Complex | Integer |

When an expression contains operands of different types, type conversion takes place during evaluation. Before each operation is performed, operands are converted to the type of the dominant operand. Thus the type of the value of the expression is determined by the dominant operand. For example, in the expression A*B-I/J, A is multiplied by B, I is divided by J as integer, converted to real, and subtracted from the result of A multiplied by B.

Octal and Hollerith constants, as well as references to shifting or masking functions, are typeless operands. When these operands are used, type is not converted. When these operands are the only operands in an expression, they are treated as if they were type integer, and the result is type integer.

Variables into which Hollerith constants are stored should be of type INTEGER to ensure proper results when used in subsequent arithmetic or logical expressions. For example, if the variables are REAL, expressions involving these variables are evaluated using floating point arithmetic.

## EXPONENTIATION

In exponentiation, the following types of base and exponent are permitted:

| Base | Exponent |
|---|---|
| Integer | Integer, Real, Double Precision, Complex, Typeless |
| Real | Integer, Real, Double Precision, Complex, Typeless |
| Double Precision | Integer, Real, Double Precision, Complex, Typeless |
| Complex | Integer, Typeless |
| Typeless | Integer, Real, Double Precision, Complex, Typeless |

The exponentation is evaluated from left to right. The expression A**B**C is evaluated as ((A**B)**C)

In an expression of the form A**B the type of the result is determined as follows:

| Type of A | Type of B | Type of Result of A**B |
|---|---|---|
| Integer | Integer<br>Real<br>Double<br>Complex<br>Typeless | Integer<br>Real<br>Double<br>Complex<br>Integer |
| Real | Integer<br>Real<br>Double<br>Complex<br>Typeless | Real<br>Real<br>Double<br>Complex<br>Real |
| Double | Integer<br>Real<br>Double<br>Complex<br>Typeless | Double<br>Double<br>Double<br>Complex<br>Double |
| Complex | Integer<br>Typeless | Complex<br>Complex |
| Typeless | Integer<br>Real<br>Double<br>Complex<br>Typeless | Integer<br>Real<br>Double<br>Complex<br>Integer |

The expression -2**2 is equivalent to 0-2**2. An exponent may be an expression. The following examples are all acceptable.

B**2.

B**N

B**(2*N-1)

(A+B)**(-J)

A negetive exponent must be enclosed in parentheses:

A**(-B)

NSUM**(-J)

When the exponent is of a type other than integer, exponentiation is performed by means of a call to FORTRAN Common Library routines. The value of the result in these cases is determined according to the formula:

$$x^y = e^{y(\ln(x))}$$

where ln is the natural logarithm function.

Examples:

| Expression | Type | Result |
|---|---|---|
| `CVAB**(I-3)` | Real**Integer | Real |
| `D**B` | Real**Real | Real |
| `C**I` | Complex**Integer | Complex |
| `BASE(M,K)**2.1` | Double Precision **Real | Double Precision |
| `K**5` | Integer**Integer | Integer |
| `314D-02**3.14D-02` | Double Precision **Double Precision | Double Precision |

## RELATIONAL EXPRESSIONS

$$\boxed{a_1 \quad op \quad a_2}$$

| | |
|---|---|
| $a_1,a_2$ | Arithmetic or masking expression |
| op | Relational operator |

A relational expression is constructed from arithmetic or masking expressions and relational operators. Arithmetic expressions may be type integer, real, double precision, or complex. The relational operators are:

| | |
|---|---|
| .GT. | Greater than |
| .GE. | Greater than or equal to |
| .LT. | Less than |
| .LE. | Less than or equal to |
| .EQ. | Equal to |
| .NE. | Not equal to |

The enclosing decimal points are part of the operator and must be present.

Two expressions separated by a relational operator constitute a basic logical element. The value of this element is either true or false. If the expressions satisfy the relation specified by the operator, the value is true; if not, it is false. For example:

```
X+Y .GT. 5.3
```

If X+Y is greater than 5.3 the value of the expression is true. If X+Y is less than or equal to 5.3 the value of the expression is false.

A relational expression can have only two operands combined by one operator. $a_1$ op $a_2$ op $a_3$ is not valid.

Relational operands may be of type integer, real, double precision, or complex, but not logical. With complex operands, the relational operators .EQ. and .NE. test for equality on both the real and imaginary parts; for all other relational operators only the real parts are compared.

Examples:

| | |
|---|---|
| `J.LT.ITEM` | |
| `580.2 .GT. VAR` | |
| `B .GE. (2.7,5.9E3)` | real part of complex number is used in evaluation |
| `E.EQ..5` | |
| `(I) .EQ. (J(K))` | |
| `C.LT. 1.5D4` | most significant part of double precision number is used in evaluation |

Relational expressions are evaluated according to the rules governing arithmetic expressions. Each expression is evaluated and compared with zero to determine the truth value. For example, the expression p.EQ.q is equivalent to the question, does p - q = 0? q is subtracted from p and the result is tested for zero. If the difference is zero or minus zero the relation is true. Otherwise, the relation is false.

If p is 0 and q is -0 the relation is true.

Expressions are evaluated from left to right. Parentheses enclosing an operand do not affect evaluation; for example, the following relational expressions are equivalent:

```
A.GT.B

A.GT.(B)

(A).GT.B

(A).GT.(B)
```

Examples:

```
REAL A
A.GT.720

INTEGER I,J
I.EQ.J(K)

(I).EQ.(N*J)

B.LE.3.754

Z.LT.35.3D+5
```

```
AMT .LT. (1.,6.55)

DOUBLE PRECISION BILL, PAY
BILL .LT. PAY

A+B.GE.Z**2

300.+B.EQ.A-Z

.5+2. .GT. .8+AMNT
```

Examples of invalid expressions:

| | |
|---|---|
| `A .GT. 720 .LE. 900` | 2 relational operators must not appear in a relational expression |
| `B .LE. 3.754 .EQ. C` | |

## LOGICAL EXPRESSIONS

$$\boxed{L_1 \text{ op } L_2 \text{ op } L_3 \text{ op} \ldots L_n}$$

| | |
|---|---|
| $L_1 \ldots L_n$ | logical operand or relational expression |
| op | logical operator |

A logical expression is a sequence of logical constants, logical variables, logical array elements, or relational expressions separated by logical operators and possibly parentheses. After evaluation, a logical expression has the value true or false.

Logical operators:

| | |
|---|---|
| .NOT. or .N. | logical negation |
| .AND. or .A. | logical multiplication |
| .OR. or .O. | inclusive OR |

The enclosing decimal points are part of the operator and must be present.

The logical operators are defined as follows (p and q represent LOGICAL expressions):

| | |
|---|---|
| .NOT.p | If p is true, .NOT.p has the value false. If p is false, .NOT.p has the value true. |
| p.AND.q | If p and q are both true, p.AND.q has the value true. Otherwise, false. |
| p.OR.q | If either p or q, or both, are true then p.OR.q has the value true. If both p and q are false, then p.OR.q has the value false. |

Truth Table

| p | q | p .AND. q | p .OR. q | .NOT. p |
|---|---|---|---|---|
| T | T | T | T | F |
| T | F | F | T | F |
| F | T | F | T | T |
| F | F | F | F | T |

If precedence is not established explicitly by parentheses, operations are executed in the following order:

.NOT. .AND. .OR.

Example:

```
      PROGRAM LOGIC(OUTPUT,TAPE6=OUTPUT)
C
C   THIS PROGRAM PRINTS OUT A TRUTH TABLE FOR LOGICAL
C        OPERATIONS WITH P AND Q
C
      LOGICAL P,Q,LOGNEG,LOGMLT,LOGSUM,TABLE(4,2)
      DATA TABLE/.TRUE.,.TRUE.,.FALSE.,.FALSE.,.TRUE.,.FALSE.,.TRUE.,
     1.FALSE./
      WRITE(6,10)
   10 FORMAT(61H1           P       Q        .NOT. Q    P .AND Q    P .O
     1R. Q   /10X, 51(1H-))
      DO 20 I = 1,4
      LOGNEG = .NOT. TABLE(I,2)
      LOGMLT = TABLE(I,1) .AND. TABLE(I,2)
      LOGSUM = TABLE(I,1) .OR. TABLE(I,2)
   20 WRITE(6,30) (TABLE(I,J),J=1,2), LOGNEG, LOGMLT, LOGSUM
   30 FORMAT(1H0, 5(L11))
      STOP
      END
```

Output:

| P | Q | .NOT. Q | P .AND Q | P .OR. Q |
|---|---|---|---|---|
| T | T | F | T | T |
| T | F | T | F | T |
| F | T | F | F | T |
| F | F | T | F | F |

The operator .NOT. which indicates logical negation appears in the form:

.NOT. p

.NOT. can appear in combination with .AND. or .OR. only as follows (p and q are logical expressions):

p .AND..NOT. q

p .OR..NOT. q

p .AND.(.NOT. q )

p .OR.(.NOT. q )

.NOT. can appear adjacent to itself only when the second operator is enclosed in parentheses, as in .NOT. (.NOT.p).

Two logical operators can appear in sequence only in the forms .OR..NOT. and .AND..NOT.

Valid logical expressions, where M, L, and Z are logical variables, are:

.NOT.L

.NOT.(X .GT. Y)

X .GT. Y .AND..NOT.Z

(L) .AND. M

Invalid logical expressions, where P and R are logical variables, are:

| | |
|---|---|
| .AND. P | .AND. must be preceded by a logical expression |
| K .EQ. 1 .OR. 2 | .OR. must be followed by a logical expression |
| P .AND. .OR.R | .AND. always must be separated from .OR. by a logical expression |

Examples:

A, X, B, C, J, L, and K are type logical.

| Expression | Alternative Form |
|---|---|
| `A .AND. .NOT. X` | `A .A. .N. X` |
| `.NOT.B` | `.N.B` |
| `A.AND.C` | `A .A.C` |
| `J.OR.L.OR.K` | `J.O.L.O.K` |

Examples:

`B-C` ≤ `A` ≤ `B+C` is written as `B-C .LE. A .AND. A .LE. B+C`
`FICA` > `176.` and `PAYNB = 5889.` is written `FICA .GT. 176. .AND. PAYNB .EQ. 5889.`

## MASKING EXPRESSIONS

Masking expressions are similar to logical expressions, but the elements of the masking expressions are any type of variable, constant, or expression other than logical.

Examples:

```
J .AND. N        .NOT. 55        .NOT. (B)        KAY .OR. 63
```

Masking operators are identical in appearance to logical operators but meanings differ. In order of dominance from highest to lowest, they are:

| | |
|---|---|
| `.NOT. or .N.` | Complement the operand |
| `.AND. or .A.` | Form the bit-by-bit logical product (AND) of two operands |
| `.OR. or .O.` | Form the bit-by-bit logical sum (OR) of two operands |

The enclosing decimal points are part of the operator and must be present. Masking operators are distinguished from logical operators by non-logical operands.

Examples:

| Expression | Alternative Form |
|---|---|
| `B .OR. D` | `B .O. D` |
| `A .AND. .NOT. C` | `A .A. .N. C` |
| `BILL .AND. BOB` | `BILL .A. BOB` |
| `I .OR. J .OR. K .OR. N` | `I .O. J .O. K .O. N` |
| `(.NOT. (.NOT.(.NOT. A .OR. B)))` | `(.N.(.N.(.N. A.OR. B)))` |

The operands may be any type variable, constant, or expression (other than logical).

Examples:

| | |
|---|---|
| `TAX .AND. INT` | |
| `.NOT. 55` | |
| `734 .OR. 82` | |
| `A .AND. 77B` | Extract the low order 6 bits of A |
| `B .OR. C` | Logical sum of the contents of B and C |
| `M .AND. .NOT. 77B` | Clear the low order 6 bits of M. |

In masking operations operands are considered to have no type. If either operand is type COMPLEX, operations are performed only on the real part. If the operand is DOUBLE PRECISION only the most significant word is used. The operation is performed bit-by-bit on the entire 60-bit word. For simplicity, only 10 bits are shown in the following examples. Masking operations are performed as follows:

J = 0101011101 and I = 1100110101

J .AND. I

The bit-by-bit logical product is formed

J 0101011101

I 1100110101

0100010101 Result after masking

J .OR. I

The bit-by-bit logical sum is formed

J 0101011101

I 1100110101

1101111101 Result after masking

.NOT. Complement the operand

.NOT. I

I 1100110101

0011001010 Result after masking

.NOT. must not immediately precede .AND. or .OR.

Using the following values:

| | | |
|---|---|---|
| A | 77770000000000000000 | octal constant |
| D | 00000000777777777777 | octal constant |
| B | 00000000000000001763 | octal form of integer constant |
| C | 20045000000000000000 | octal form of real constant |

Masking operations produce the following octal results:

| | | |
|---|---|---|
| .NOT. A | is | 00007777777777777777 |
| A .AND. C | is | 20040000000000000000 |
| A .AND. .NOT. C | is | 57730000000000000000 |
| B .OR. .NOT. D | is | 77777777000000001763 |

Invalid example:

```
LOGICAL A
A .AND. B .OR. C
```
masking expression must not contain logical operand

Example:

```
      PROGRAM MASK (INPUT,OUTPUT)
1     FORMAT (1H1,5X,4HNAME,///)
      PRINT 1
2     FORMAT (3A10,I1)
3     READ 2,LNAME,FNAME,ISTATE,KSTOP
      IF(KSTOP.EQ.1)STOP
C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD
      IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.777700000000000
     K00000B))  GO TO 3
11    FORMAT(5X,2A10)
10    PRINT 11,LNAME,FNAME
      GO TO 3
      END
```

## ASSIGNMENT STATEMENTS

An assignment statement evaluates an expression and assigns this value to a variable or array element. The statement is written as follows:

v = expression

v is a variable or an array element

The meaning of the equals sign differs from the conventional mathematical notation. It means replace the value of the variable on the left with the value of the expression on the right. For example, the assignment statement A=B+C replaces the current value of the variable A with the value of B+C.

## ARITHMETIC ASSIGNMENT STATEMENTS

```
      7
      v = arithmetic expression
```

Replace the current value of v with the value of the arithmetic expression. The variable or array element can be any type other than logical.

Examples:

| | |
|---|---|
| `A=A+1` | replace the value of A with the value of A+1 |
| `N=J-100*20` | replace N with the value of J-100*20 |
| `WAGE=PAY-TAX` | replace WAGE with the value of PAY less TAX |
| `VAR=VALUE+(7/4)*32` | replace the value of VAR with the value of VALUE+(7/4)*32 |
| `B(4)=B(1)+B(2)` | replace the value of B(4) with the value of B(1)+B(2) |

If the type of the variable on the left of the equals sign differs from that of the expression on the right, type conversion takes place. The expression is evaluated, converted to the type of the variable on the left, and then replaces the current value of the variable. The type of an evaluated arithmetic expression is determined by the type of the dominant operand. Below, the types are ranked in order of dominance from highest to lowest:

Complex

Double Precision

Real

Integer

In the following tables, if high order bits are lost by truncation during conversion, no diagnostic is given.

**CONVERSION TO INTEGER**

| | Value Assigned | Example | Value of IFORM After Evaluation |
|---|---|---|---|
| Integer = Integer | Value of integer expression replaces v. | IFORM = 10/2 | 5 |
| Integer = Real | Value of real expression, truncated to 48-bit integer, replaces v. | IFORM = 2.5*2+3.2 | 8 |
| Integer = Double Precision | Value of double precision expression, truncated to 48-bit integer, replaces v. | IFORM = 3141.593D3 | 3141593 |
| Integer = Complex | Value of real part of complex expression truncated to 48-bit integer, replaces v. | IFORM = (2.5,3.0) + (1.0,2.0) | 3 |

**CONVERSION TO DOUBLE PRECISION**

| | Value Assigned | Example | Value of SUM After Evaluation |
|---|---|---|---|
| Double Precision = Integer | Value of integer expression, truncated to 48 bits, is converted to real and replaces most significant part. Least significant part set to 0. | SUM = 7*5 | 35.D0 |
| Double Precision = Real | Value of real expression replaces most significant part; least significant part is set to 0. | SUM = 7.5*2 | 15.D0 |

CONVERSION TO DOUBLE PRECISION (CONTINUED)

| | Value Assigned | Example | Value of SUM After Evaluation |
|---|---|---|---|
| Double Precision = Double Precision | Value of double precision expression replaces v. | SUM = 7.322D2 – 32.D -1 | 7.29D2 |
| Double Precision = Complex | Value of real part of complex expression replaces v. Least significant part is set to 0. | SUM = (3.2,7.6) + (5.5,1.0) | 8.7D0 |

CONVERSION TO COMPLEX

| | Value Assigned | Example | Value of AFORM After Evaluation |
|---|---|---|---|
| Complex = Integer | Value of integer expression, truncated to 48 bits, is converted to real, and replaces real part of v. Imaginary part is set to 0. | AFORM = 2 + 3 | (5.0,0.0) |
| Complex = Real | Value of real expression replaces real part of v. Imaginary part set to 0. | AFORM = 2.3 + 7.2 | (9.5,0.0) |
| Complex = Double Precision | Most significant part of double precision expression replaces real part of v. Imaginary part set to 0. | AFORM = 20D0 + 4.4D1 | (64.0,0.0) |
| Complex = Complex | Value of complex expression replaces variable. | AFORM = (3.4,1.1) + (7.3,4.6) | (10.7,5.7) |

### CONVERSION TO REAL

| | Value Assigned | Example | Value of AFORM After Evaluation |
|---|---|---|---|
| Real = Integer | Value of integer expression, truncated to 48 bits, is converted to real and replaces v. | AFORM = 200 + 300 | 500.0 |
| Real = Real | Value of real expression replaces v. | AFORM = 2.5 + 7.2 | 9.7 |
| Real = Double Precision | Value of most significant part of expression replaces v. | AFORM = 3421.D - 04 | .3421 |
| Real = Complex | Value of real part of complex expression replaces v. | AFORM = (9.2,1.1) – (2.1,5.0) | 7.1 |

## LOGICAL ASSIGNMENT

7

Logical variable or array element = Logical or relational expression

Replace the current value of the logical variable or array element with the value of the expression.

Examples:

```
LOGICAL LOG2
I = 1
LOG2 = I .EQ.0
```

LOG2 is assigned the value .FALSE. because I≠0

```
LOGICAL NSUM,VAR
BIG = 200.
VAR = .TRUE.
NSUM = BIG .GT. 200. .AND. VAR
```

NSUM is assigned the value .FALSE.

```
LOGICAL A,B,C,D,E,LGA,LGB,LGC
REAL F,G,H
A = B.AND.C.AND.D
A = F.GT.G.OR.F.GT.H
A = .NOT.(A.AND..NOT.B).AND.(C.OR.D)
LGA = .NOT.LGB
LGC = E.OR.LGC.OR.LGB.OR.LGA.OR.(A.AND.B)
```

## MASKING ASSIGNMENT

```
      7
      v = masking expression
```

Replace the value of v with the value of the masking expression. v can be any type other than logical. No type conversion takes place during replacement. If the type is double precision or complex, the value of the expression is assigned to the first word of the variable; and the least significant or imaginary part set to zero.

Examples:

```
B = D .AND. Z .OR. X
SUM = (1.0,2.0) .OR. (7.0,7.0)
NAME = INK .OR. JAY .AND. NEXT
J(3) = N .AND. I
A = B.OR. (C.AND. Z)

INTEGER I,J,K,L,M,N(16)
REAL B,C,D,E,F(15)

N(2) = I.AND.J
B = C.AND.L
F(J) = I.OR..NOT.L.AND.F(J)
I = .NOT.I
N(1) = I.OR.J.OR.K.OR.L.OR.M
```

## MULTIPLE ASSIGNMENT

7

$v_1 = v_2 = . . . = v_n$ = expression

Replace the value of several variables or array elements with the value of the expression. For example, X = Y = Z= (10+2)/SUM(1) is equivalent to the following statements:

```
Z = (10 + 2)/SUM(1)

Y = Z

X = Y
```

The value of the expression is converted to the type of the variable or array element during each replacement.

Examples:

```
NSUM = BSUM = ISUM = TOTAL = 10.5 - 3.2
```

1. TOTAL is assigned the value 7.3
2. ISUM is assigned the value 7
3. BSUM is assigned the value 7.0
4. NSUM is assigned the value 7

Multiple assignment is legal in all types of assignment statements.

# SPECIFICATION STATEMENTS 3

Specification statements are non-executable; they define the type of a variable or array, specify the amount of storage allocated to each variable according to its type, specify the dimensions of arrays, define methods of sharing storage, and assign initial values to variables and arrays. The specification statements are:

| | |
|---|---|
| IMPLICIT | The IMPLICIT statement must precede other specification statements. |
| Type<br>DIMENSION<br>COMMON<br>EQUIVALENCE<br>EXTERNAL<br>LEVEL | If any of these statements appears after the first executable statement or statement function definition, the specification statement is ignored and a fatal diagnostic is printed. |

The DATA statement, which is not a specification statement, is also described in this section. The DATA statement must follow all other specification statements except statement function definitions and FORMAT statements; it can occur after the first executable statement.

## TYPE STATEMENTS

A type statement defines a variable, array, or function to be integer, real, complex, double precision, or logical. An explicit type statement can be used to supply dimension information. The word TYPE may be used as a prefix.

In the absence of an explicit type statement, the type of a symbolic name is implied by the first character of the name: I, J, K, L, M, or N imply type integer and any other letter implies type real, unless an IMPLICIT statement is used to change this normal implied type.

Basic external and intrinsic functions are implicitly typed, and need not appear in a type statement in the user's program. The type of each library function is listed in section 8.

## EXPLICIT TYPE DECLARATIONS

There are five explicit type statements: INTEGER, REAL, COMPLEX, DOUBLE PRECISION, and LOGICAL.

### INTEGER

```
      7
      INTEGER name1, name2 ,..., namen
```

The symbolic names listed are declared as type integer.

Example:

```
INTEGER SUM, RESULT, ALIST
```

The variables SUM, RESULT and ALIST are all declared as type integer.

## REAL

7

REAL name$_1$, name$_2$, . . . , name$_n$

The symbolic names listed are declared as type real.

Example:

```
REAL NEXT(7), ITEM
```

NEXT is declared as an array with 7 real elements, and ITEM is declared as a real variable.

## COMPLEX

7

COMPLEX name$_1$, name$_2$, . . . , name$_n$

The symbolic names listed are declared as type complex.

Example:

```
COMPLEX ALPHA, NAM, MASTER, BETA
```

The variables ALPHA, NAM, MASTER, BETA are declared as type complex.

## DOUBLE PRECISION

7

DOUBLE PRECISION name$_1$, name$_2$, . . . , name$_n$

The symbolic names listed are declared as type double precision. DOUBLE can be used instead of DOUBLE PRECISION.

Example:

```
DOUBLE PRECISION ALIST, JUNR, BOX4
```

The variables ALIST, JUNR, BOX4 are declared as type double precision.

## LOGICAL

```
      7
      LOGICAL name1, name2, ..., namen
```

The symbolic names listed are declared as type logical.

Example:

```
LOGICAL P,Q,NUMBR4
```

The variables P, Q and NUMBR4 are declared as type logical.

# IMPLICIT TYPE STATEMENT

```
      7
      IMPLICIT type1 (ac1 , ..., acn), ..., typen (ac1 , ..., acn)
```

type LOGICAL, INTEGER, REAL, DOUBLE PRECISION, DOUBLE, or COMPLEX

$ac_i$ Single alphabetic characters, or ranges of characters represented by the first and last character separated by a minus sign.

This statement specifies the type of variables, arrays, and functions beginning with the letters ac. Only one IMPLICIT statement may appear in a program unit, and it must precede other specification 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.

Example 1:

```
COMPLEX FUNCTION RHO (CDHS)
IMPLICIT INTEGER(A-D,R)
REAL ASUM
ASUM = BOR + ROR * ANEXT
```

The variables BOR, ROR, ANEXT, and the formal parameter CDHS are of type integer, ASUM is type real, and the function RHO is of type complex.

Example 2:

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      IMPLICIT INTEGER (A-F,H)
      DIMENSION E(3,4)
      COMMON A(1),B,C,D,  F,G,H
      EQUIVALENCE (A,E,I)
      NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,VLIST)
      STOP
      END
```

The arrays A and E and the variables B, C, D, F, H, and I are of type integer; G is type real.

## DIMENSION STATEMENT

```
      7
      DIMENSION name1(d1),...,namen(dn)
```

$d_i$ Array declarator, 1-3 integer constants separated by commas. If name is a dummy parameter, d can be 1-3 integer constants or integer dummy parameters intermixed.

$name_i$ Symbolic name of an array.

The DIMENSION statement is a nonexecutable statement which defines symbolic names as array names and specifies the bounds of the array. More than one array can be declared in a single DIMENSION statement. Dummy parameter arrays specified within a procedure subprogram can have adjustable dimension specifications. (A further explanation of adjustable dimension specifications appears under Procedure Communication in section 7). Within the same program unit, only one definition of an array is permitted.

The number of computer words reserved for an array is determined by the type of the array and the product of the subscripts. For real, integer and logical arrays, the number of words in an array equals the number of elements in the array. For complex and double precision arrays, the number of words reserved is twice the product of the subscripts. No array can exceed 131,071 words.

Example:

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

BETA is an array containing six elements; however, BETA has been defined as COMPLEX and two words are used to contain each complex element; therefore, 12 computer words are reserved.

Example:

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

These statements could be combined into one statement with 24 words reserved for array NIL:

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

Example:

```
DIMENSION ASUM(10,2)
.
.
.
DIMENSION ASUM (3), VECTOR (7,7)
```

The second specification of ASUM is ignored, and an informative message is printed. The specification for VECTOR is valid and is processed.

## COMMON STATEMENT

```
7
COMMON/ /v1,...,vn
```

```
7
COMMON/blkname1/v1, . . . ,vn/blkname2/v1, . . .vn. . ./blknamen/v1, . . . ,vn
```

```
7
COMMON v1,...,vn
```

| | |
|---|---|
| $blkname_i$ | Block name or number. A block name is a symbolic name of 1-7 letters or digits beginning with a letter. A block number is 1-7 digits; it must not contain any alphabetic characters. Leading zeros are ignored. 0 is a valid block number. The same block name or number can appear more than once in a COMMON statement or a program unit; the loader links all variables in blocks having the same name or number into a single labeled common block. |
| $v_i$ | Variable or array name which can be followed by constant subscripts that declare the dimensions. The variable or array names are assigned to blkname. The COMMON statement can contain one or more block specifications. |

| | |
|---|---|
| // | Denotes a blank common block. If blank common is the first block in the statement, slashes can be omitted. |

Variables or arrays in a main program or subprogram can share the same storage locations with variables or arrays in other subprograms by means of the COMMON statement. Variables and array names are stored in the order in which they appear in the block specification.

COMMON is a non-executable statement. See section 1 for proper location of COMMON statements relative to other statements in the program unit. The COMMON specification provides up to 125 storage blocks that can be referenced by more than one subprogram. A block of common storage can be labeled by a name or a number. A COMMON statement without a name or number refers to a blank common block. Variables and array elements can appear in both COMMON and EQUIVALENCE statements. A common block of storage can be extended by an EQUIVALENCE statement; however, no common block can exceed 131,071 words.

All members of a common block must be allocated to the same level of storage; a fatal diagnostic is issued if conflicting levels are declared. If only some members of a common block are declared in a LEVEL statement, the remaining members of that common block are allocated automatically to the same level, and an informative diagnostic is issued.

Block names can be used elsewhere in the program as variable or array names, and they can be used as subprogram names. Numbered common is treated like labeled common. Data stored in common blocks by the DATA statement is available to any subprogram using these blocks.

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.

Example:

```
COMMON/BLACK/A(3)
DATA A/1.,2.,3./

COMMON/100/I(4)
DATA I/4,5,6,7/
```

Data may not be entered into blank common blocks by the DATA declaration.

The COMMON statement may contain one or more block specifications:

```
COMMON/X/RAG,TAG/APPA/Y,Z,B(5)
```

RAG and TAG are placed in block X. The array B and Y.Z are placed in block APPA.

Any number of blank common specifications can appear in a program. Blank. named and numbered common blocks are cumulative throughout a program. as illustrated by the following example:

```
COMMON A,B,C/X/Y,Z,D//W,R
.
.
.
COMMON M,N/CAT/ALPHA,BINGO//ADD
```

These statements have the same effect as the single statement:

```
COMMON A,B,C,W,R,M,N,ADD/X/Y,Z,D/CAT/ALPHA,BINGO
```

Within subprograms, dummy arguments are not allowed in the COMMON statement.

If dimension information for an array is not given in the COMMON statement, it must be declared in a type or DIMENSION statement in that program unit.

Examples:

```
COMMON/DEE/Z(10,4)
```

Specifies the dimensions of the array Z and enters Z into labeled common block DEE.

```
COMMON/BLOKE/ANARAY,B,D
DIMENSION ANARAY(10,2)

COMMON/Z/X,Y,A
REAL X(7)

COMMON/HAT/M,N,J(3,4)
DIMENSION J(2,7)
```

In the last example, J is defined as an array (3,4) in the COMMON statement. (2,7) in the DIMENSION statement is ignored and an error message is printed.

The length of a common block, in computer words, is determined by the number and type of the variables and array elements in that block. In the following statements, the length of common block A is 12 computer words. The origin of the common block is Q(1).

```
REAL Q,R
COMPLEX S
COMMON/A/Q(4),R(4),S(2)
```

| | **Block A** | |
|---|---|---|
| origin | Q(1) | |
| | Q(2) | |
| | Q(3) | |
| | Q(4) | |
| | R(1) | |
| | R(2) | |
| | R(3) | |
| | R(4) | |
| | S(1) | real part |
| | S(1) | imaginary part |
| | S(2) | real part |
| | S(2) | imaginary part |

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.

Example:

```
COMMON/SUM/A,B,C,D  main program

COMMON/SUM/E(3),D   subprogram
```

If the subprogram does not use variables A,B, and C, array E is necessary to space over the area reserved by A,B, and C.

Alternatively, correspondence can be ensured by placing unused variables at the end of the common list.

```
COMMON/SUM/D,A,B,C  main program

COMMON/SUM/D        subprogram
```

If program units share the same common block, they may assign different names and types to the members of the block; but the block name or numbers must remain the same.

Example:

```
COMPLEX C
COMMON/TEST/C(20)
```

The block named TEST consists of 40 computer words.

The subprogram may use different names for variables and arrays as in:

```
SUBROUTINE ONE
COMPLEX A
COMMON/TEST/A(10),G(10),K(10)
```

The length of TEST is 40 words. The first 10 elements (20 words) of the block represented by A are complex elements. Array G is the next 10 words, and array K is the last 10 words. Within the subprogram, elements of G are treated as floating point; elements of K are treated as integer.

## EQUIVALENCE STATEMENT

```
      7
      EQUIVALENCE (glist1),...,(glistn)
```

Each $glist_i$ consists of two or more variables, array elements, or array names, separated by commas.

Array elements must have integer constant subscripts. Dummy arguments must not appear in an equivalence statement. Equivalenced variables must be in the same level of storage. If members of an equivalence class are assigned to conflicting storage levels, a fatal error results. If a member of an equivalence class is declared in a LEVEL statement, the other members of the class are automatically allocated to the same level, and an informative diagnostic is issued.

EQUIVALENCE is a non-executable statement and must appear before all executable statements in a program unit. If it appears after the first executable statement, a fatal diagnostic is printed.

EQUIVALENCE assigns two or more variables in the same program unit to the same storage location (as opposed to COMMON which assigns two variables in different program units to the same location). Variables or array elements not mentioned in an EQUIVALENCE statement are assigned unique locations.

Example:

```
DIMENSION JAN(6),BILL(10)
EQUIVALENCE (IRON,MAT,ZERO), (JAN(5),BILL(2)),(A,B,C)
```

The variables IRON, MAT and ZERO share the same location, the fifth element in array JAN and the second element in array BILL share the same location, and the variables A,B and C share the same location.

When an element of an array is referred to in an EQUIVALENCE statement, the relative locations of the other array elements are, thereby, defined also.

Example:

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

This EQUIVALENCE statement causes storage to be shared by the first element in Y and the fourth element in B and, similarly, the variable X and the fourth element in Y. Storage will be as follows:

| | | |
|---|---|---|
| B(1,1) | | |
| B(2,1) | | |
| B(3,1) | | |
| B(1,2) | Y(1) | |
| B(2,2) | Y(2) | |
| B(3,2) | Y(3) | |
| | Y(4) | X |

The elements of a glist constitute an **equivalence group**. When an equivalence group contains an element that appears in another equivalence group, these groups are merged and their elements constitute an **equivalence class.**

Example:

```
DIMENSION A(100)
EQUIVALENCE (A,B), (C,A(50)), (D,E), (F,C)
```

These statements establish the following equivalence groups:

$\{A,B\}$, $\{A,C\}$, $\{C,F\}$, $\{D,E\}$

and the following equivalence classes:

$\{A,B,C,F\}$, $\{D,E\}$

The statement EQUIVALENCE (A,B),(B,C) has the same effect as EQUIVALENCE (A,B,C).

When no array subscript is given, it is assumed to be 1.

```
DIMENSION ZEBRA(10)
EQUIVALENCE (ZEBRA,TIGER)
```

means the same as the statements:

```
DIMENSION ZEBRA(10)
EQUIVALENCE (ZEBRA(1),TIGER)
```

A logical, integer, or real entity equivalenced to a double precision or complex entity shares the same location as the real or most significant part of the complex or double precision entity. If an array with single-word elements is equivalenced to an array with double-word elements, the single word elements are stored contiguously, so that each half of a double-word element is equivalent to a different single-word element.

An array with multiple dimensions may be referenced with a single subscript. The location of the element in the array may be determined by the following method:

```
DIMENSION A(K,M,N)
```

The position of element A(k,m,n) is given by:

A(k+K*(m-1+M*(n-1)))*E)

E is 1 if A is real, integer or logical; E is 2 if A is complex or double precision.

Example:

```
DIMENSION AVERAG(2,3,4),TERM(7)
EQUIVALENCE (AVERAG(8),TERM(2))
```

Elements AVERAG (2,1,2) and TERM(2) share the same locations.

Two or more arrays can share the same storage locations.

Example:

```
      DIMENSION ITIN(10,10),TAX(100)
      EQUIVALENCE(ITIN(1),TAX(1))
      .
      .
      .
  500 READ (5,40)ITIN (1)
      .
      .
      .
  600 READ (5,70) TAX (1)
```

The EQUIVALENCE declaration assigns the first elements of arrays ITIN and TAX to the same location. READ statement 500 stores the array ITIN in consecutive locations. Before READ statement 600 is executed, all operations involving ITIN should be completed; as the values of array TAX are read into the storage locations previously occupied by ITIN.

Lengths of arrays need not be equal.

Examples:

```
DIMENSION ZERO1(10,5),ZERO2(3,3)
EQUIVALENCE (ZERO1,ZERO2)
```

is a legal EQUIVALENCE statement

```
EQUIVALENCE (ITEM,TEMP)
```

The integer variable ITEM and the real variable TEMP share the same location; therefore, the same location may be referred to as either integer or real. However, the integer and real internal formats differ; therefore the values will not be the same.

## EQUIVALENCE AND COMMON

Variables, array elements, and arrays may appear in both COMMON and EQUIVALENCE statements. A common block of storage may be extended by an EQUIVALENCE statement.

Example:

```
COMMON/HAT/A(4),C
DIMENSION B(5)
EQUIVALENCE (A(2),B(1))
```

Common block HAT will extend from A(1) to B(5):

| | | | |
|---|---|---|---|
| /HAT/ | Origin | A(1) | |
| | | A(2) | B(1) |
| | | A(3) | B(2) |
| | | A(4) | B(3) |
| | | C | B(4) |
| | | | B(5) |

EQUIVALENCE statements which extend the origin of a common block are not allowed, however.

Example:

```
COMMON/DESK/E,F,G
DIMENSION H(4)
EQUIVALENCE (E,H(3))
```

The above EQUIVALENCE statement is illegal because H(1) and H(2) extend the start of the common block DESK:

| | | | |
|---|---|---|---|
| /DESK/ | | | H(1) |
| | | | H(2) |
| | Origin | E | H(3) |
| | | F | H(4) |
| | | G | |

An element or array is brought into COMMON if it is equivalenced to an element in COMMON. Two elements in COMMON must not be equivalenced to each other.

Examples:

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

COMMON /HAT/ A(4),C /X/ Y,Z
EQUIVALENCE (C,Y)              illegal
```

As stated in section 1, the result of indexing outside of array bounds is unpredictable. Since the compiler attempts to minimize the size of equivalence classes in common blocks to the smallest subset of the block that includes all members named in associated EQUIVALENCE statements, all members of a common block will not necessarily be considered as one array. The programming practice of intentionally referencing locations outside a known array can produce unintentional results as shown in the following example.

```
      COMMON/ /A(4), B, D, E
      DIMENSION AA(4)
      EQUIVALENCE (AA, A(2))
      D=2.
      E=2.
      DO 10 I=1, 6
   10 AA(I)=D*E
      PRINT *, E
```

When these statements are compiled under OPT=0, E will have a value of 8. on exit. Under OPT=1 or 2, the evaluation of D*E will be moved out of the loop since AA and D (or E) are not recognized as being in the same equivalence class. If the program is to produce the same results under all OPT levels, AA must be dimensioned to include the entire common block in the equivalence class.

## LEVEL STATEMENT

7

LEVEL n, $a_1$, ..., $a_n$

$a_i$ Variables or array names.

n Unsigned integer 1, 2, or 3 indicating level to which list is to be allocated.

§
- 1 Small core memory resident (SCM)
- 2 Large core memory resident (LCM). Directly addressable (or word addressable)
- 3 Large core memory resident, accessed by block transfer to or from small core memory through MOVLEV subroutine call (section 8)

‡
- 1 Central memory resident
- 2 Central memory resident
- 3 Extended core storage resident, accessed by block transfer to or from central memory through MOVLEV subroutine call

The LEVEL statement specifies the storage level of variables or array names. 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. LEVEL statements must precede the first executable statement in a program unit. Names of variables and arrays which do not appear in a LEVEL statement are allocated to central memory.

No dimension or type information is included in the LEVEL statement.

Variables and arrays appearing in a LEVEL statement can appear in DATA, DIMENSION, EQUIVALENCE, COMMON, type, SUBROUTINE and FUNCTION statements. Data assigned to levels 2 and 3 must be members of common blocks or dummy arguments.

For all levels, no single array or common block can exceed 131,071 words. If the total ECS, LCM, or UEM field length accessed by the entire program exceeds 131,071 words, the LCM = I parameter must be specified on the compiler call. When ECS and LCM blocks are loaded, the length assigned at compile time is rounded up to a multiple of eight words.

Data assigned to level 3 can be referenced only in: COMMON, DIMENSION, type, EQUIVALENCE, DATA, CALL, SUBROUTINE, and FUNCTION statements. Level 3 items cannot be used in expressions.

No restrictions are imposed on the way in which reference is made to variables or arrays allocated to levels 1 and 2.

If the level of any variable is multiply defined, the level first declared is assumed; and a warning diagnostic is printed.

All members of a common block or equivalence class must be assigned to the same level; a fatal diagnostic is issued if conflicting levels are declared. If some, but not all, members of a common block or equivalence class are declared in a LEVEL statement, all are assigned to the declared level, and an informative diagnostic is printed.

---

§ Applies only to Control Data CYBER 170 Model 176, CYBER 170 800 Series, CYBER 70 Model 76, and 7600 computers.

‡ Applies only to Control Data CYBER 70 Models 71, 72, 73 and 74, CYBER 170 Models 171, 172, 173, 174, 175, and 6000 Series computers.

If a variable or array name declared in a LEVEL statement appears as an actual argument in a CALL statement, the corresponding dummy argument must be allocated to the same level in the called subprogram.

Example:

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

The LEVEL statement allocates arrays E and B to extended core storage. They are assigned to a named common block, ECSBLK. Starting at location CM (the first word address of the array CM), 1000 words of central memory are transferred to the two arrays E and B in extended core storage by the library routine MOVLEV.

## EXTERNAL STATEMENT

```
      7
      EXTERNAL name1,...,namen
```

| | |
|---|---|
| $name_1$,...,$name_n$ | Subprogram names |

Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program. The subprogram can be user-supplied or can reside in the FORTRAN library. If an actual argument is the name of an intrinsic function, and the user supplies a subprogram with the same name as the intrinsic function, the user-supplied function will be used. Otherwise, the subprogram is supplied by the library. If the subprogram does not exist on the FORTRAN library and is not supplied by the user, a loader error occurs.

In a chain of nested subroutine calls, where a subprogram name is passed as an argument to lower level subroutines, the EXTERNAL statement need only appear in the highest level program unit in which a CALL statement containing the subprogram name appears.

Any name used as an actual argument in a call is assumed to be a variable or array unless it appears in an EXTERNAL statement. An EXTERNAL statement must be used even if the subprogram concerned is a basic external function, such as SQRT.

Example:

**Calling Program**

```
      EXTERNAL SIN, SQRT
      CALL SUBRT(2.0,SIN,RESULT)
      WRITE (6,100) RESULT
  100 FORMAT (F7.3)
      CALL SUBRT(2.0,SQRT,RESULT)
      WRITE (6,100)RESULT
      STOP
      END
```

**Subprogram**

```
      SUBROUTINE SUBRT (A,B,C)
      X=A+3.14159/2.
      C=B(X)
      RETURN
      END
```

First the sine, then the square root are computed; and in each case, the value is returned in RESULT.

The EXTERNAL statement must precede the first executable statement, and always appears in the calling program. (It cannot be used with statement functions.)

A function call that provides values for an actual argument does not need an EXTERNAL statement.

Example:

**Calling Program**

```
      CALL SUBRT(SIN(X),RESULT)
```

**Subprogram**

```
      SUBROUTINE SUBRT(A,B)
      .
      .
      .
      B=A
      .
      .
      .
      END
```

An EXTERNAL statement is not required because the function SIN is not the argument of the subprogram; the evaluated result of SIN(X) becomes the argument.

## DATA STATEMENT

7

DATA vlist$_1$/dlist$_1$/,vlist$_2$/dlist$_2$/, . . . , vlist$_n$/dlist$_n$/

7

DATA (vlist = dlist), . . . , (vlist = dlist)

vlist — List of array names, array elements, variable names, and implied DO loops, separated by commas. Unless they appear in an implied DO loop, array elements must have integer constant subscripts.

dlist One or more of the following forms separated by commas:

constant
(constant list)
rf*constant
rf*(constant list)
rf(constant list)

| | |
|---|---|
| constant list | List of constants separated by commas. |
| rf | Positive integer constant. The constant or constant list is repeated the number of times indicated by rf. |

The data statement is non-executable and must follow all specification statements except statement function definitions, NAMELIST statements, and FORMAT statements. It can occur after the first executable statement. It assigns initial values to variables or array elements. Only variables assigned values by the DATA statement have specified values when program execution begins. The DATA statement cannot be used to assign values in blank common or to dummy arguments.

The number of items in the data list should agree with the number of variables in the variable list. If the data list contains more items than the variable list, excess items are ignored, and an informative diagnostic is printed. If the data list contains fewer items than the variable list, remaining variables are not defined, and an informative diagnostic is printed.

The type of the constant in the data list should agree with the type associated with the corresponding name in the variable list. If the types do not agree, the form of the value stored is determined by the constant used in the DATA statement rather than by the type of the name in the variable list.

When constants in a data list are enclosed in parentheses and preceded by an integer constant, the list is repeated the number of times indicated by the integer constant. If the repeat constant is not an integer, a fatal error message is printed.

The forms:

rf * (real constant, real constant)

and:

rf (real constant, real constant)

are interpreted as a repeated specification of two real constants, not as a single complex constant. In order to specify the repetition of a complex constant, another set of parentheses must be used:

rf * ((real constant, real constant))

or:

rf ((real constant, real constant))

Example:

| | |
|---|---|
| 2*(1.0, 2.0) | Means repeat the real constants 1.0 and 2.0 twice |
| 2*((1.0, 2.0)) | Means repeat the complex constant (1.0, 2.0) twice |

An unsubscripted array name implies the entire array in the order it is stored in memory.

Example:

```
INTEGER   B(10)
DATA B/000077B,000064B,3*000005B,5*000200B/
```

The following octal constants are stored in ARRAY B:

```
 77B
 64B
  5B
  5B
  5B
200B
200B
200B
200B
200B
```

Example:

```
      PROGRAM DATA C (OUTPUT,TAPE6=OUTPUT)
      COMPLEX Z(3),Z1
      REAL A(4)
      LOGICAL L
      NAMELIST/OUT/I,L,X,Z1,A,Z
      DATA I,L,X,Z1,A,Z/5,.TRUE.,3.1415926536,(2.1,-3.),2*(1.,2.),
     1   3*((1.,-1.5))/
      WRITE(6,OUT)
      STOP
      END
```

---

```
$OUT

I      = 5,

L      = T,

X      = .31415926536E+01,

Z1     = (.21E+01,-.3E+01),

A      = .1E+01, .2E+01, .1E+01, .2E+01,

Z      = (.1E+01,-.15E+01), (.1E+01,-.15E+01), (.1E+01,-.15E+01),
$END
```

Example:

The following are examples of alternative (nonstandard) forms of the DATA statement:

```
DATA (X=3.),(Y=5.)

INTEGER ARAY(5)
DATA (A=7.),(B=200.),(ARAY=1,2,7,50,3)

COMMON/BOX/ARAY4(3,4,5)
DATA (ARAY4(1,3,5)=22.5)
```

Hollerith constants of any length can be included in the data list. If the constant corresponds to a one-word variable (integer, real, or logical), the variable is set to the first 10 characters of the Hollerith constant. If the constant corresponds to an array element, that element, and as many succeeding elements as necessary, are filled by the constant.

Example:

```
DIMENSION JEF(10)
DATA JEF(3) / 40HI THINK, THEREFORE I AM,  .......  I THINK/
```

The third through sixth elements of the array JEF are set as follows:

```
JEF(3) = 10HI THINK, T
JEF(4) = 10HHEREFORE I
JEF(5) = 10H AM,  .....
JEF(6) = 10H..  I THINK
```

Subsequent explicit setting of elements in the array, however, overrides implicit setting by means of a lengthy Hollerith constant.

Example:

```
 INTEGER ABC(5)
 DATA ABC(3), ABC(4) /30HONCE UPON A MIDNIGHT DREARY     ,
+10HA TIME  .../
```

"he third and fourth elements of the array ABC are set as follows:

```
ABC(3) = 10HONCE UPON
ABC(4) = 10HA TIME  ...
```

If an unsubscripted array name appears in a variable list, each item in the data list sets exactly one array element, even if Hollerith constants occur that exceed the array element length. The first data list item sets the first array element, the second data list item sets the second array element, and so forth.

Initially, the full length of each Hollerith constant is assigned to successive array elements; subsequent data list items might cause overwriting of all or part of the constant.

Example:

```
DIMENSION J(4)
DATA J/20H1234567890ABCDEFGHIJ,  20HKLMNOPQRSTUVWXYZ+-*$/
```

The elements of array J are set as follows:

```
J(1) = 10H1234567890
J(2) = 10HKLMNOPQRST
J(3) = 10HUVWXYZ+-*$
J(4) = undefined
```

The first Hollerith constant in the data list sets elements J(1) and J(2), and the second constant sets elements J(2) and J(3), overriding the previous setting of J(2). J(4) remains undefined.

Because of this method of storing Hollerith constants appearing in DATA statements, there are only two ways in which an array can be set to a long character string:

The string can be specified as one long constant:

```
DATA J/40H1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ+-*$/
```

or the string can be broken up into constants of 10 characters each:

```
DATA J/10H1234567890,  10HABCDEFGHIJ,  10HKLMNOPQRST, 10HUVWZYZ+-*$/
```

If variables containing Hollerith data are to be compared with Hollerith constants, the variables should be of type INTEGER so that the actual bit value is used and no conversion is performed.

## IMPLIED DO IN DATA LIST

The implied DO can be used as a shortened notation for specifying items in the variable list of a DATA statement. The implied DO in a DATA statement has the following form:

(varlist,i=$m_1$,$m_2$,$m_3$)

where:

| | |
|---|---|
| varlist | an array element name or another implied DO. If it is an array element name, its subscript expressions must be of the form<br><br>M*i±N<br><br>where M and N are unsigned non-zero integer constants. ±N can be omitted. |
| i | a simple integer variable called the index variable |
| $m_1$,$m_2$,$m_3$ | unsigned integer constants specifying the initial value, terminal value, and increment, respectively, for the index variable; if $m_3$ and the preceding comma are omitted, the value of $m_3$ is assumed to be 1. |

The range of the implied DO is varlist. Within the range, the value of the variable i must not be redefined. If varlist contains more implied DOs, those implied DOs are considered to be nested within the containing implied DO; the nested implied DO is completely processed for each value of i in the containing implied DO. Implied DOs can be nested a maximum of three deep.

When an implied DO is encountered in a DATA statement, the elements in its range are initialized for index variable i with the value $m_1$. The index variable is then increased by $m_3$ and, if i is less than or equal to $m_2$, the range of varlist is initialized for the new value of i. This procedure continues until the value of the index variable exceeds $m_2$.

Example 1:

```
REAL ANARAY(10)
DATA (ANARAY(I),I = 1,10)/1.,2.,3.,7*2.5/
```

The values stored in array ANARAY are:

| | |
|---|---|
| ANARAY(1) | 1. |
| | 2. |
| | 3. |
| | 2.5 |
| | 2.5 |
| | 2.5 |
| | 2.5 |
| | 2.5 |
| | 2.5 |
| ANARAY(10) | 2.5 |

When an implied DO is used to store values into arrays, only one array name can be used within the implied DO nest.

Example 2:

```
DIMENSION UNIT (10,10)
DATA (UNIT(I, I), I=1, 10)/10*1./
```

These two statements declare a matrix and preset the diagonal elements to ones.

Example 3:

```
DIMENSION AR(10)
DATA (AR(2*I+1), I=1, 4)/4*3.5/
```

These two statements declare a ten-word array and preset elements AR(3), AR(5), AR(7), and AR(9) to 3.5.

Example 4:

```
DIMENSION AMASS(10,10,10), A(10), B(5)
DATA (AMASS(6,K,3),K=1,10)/4*(-2.,5.139),6.9,10./
DATA (A(I),I=5,7)/2*(4.1),5.0/
DATA B/5*0.0/
```

These statements dimension arrays AMASS, A, and B and preset elements as follows:

ARRAY AMASS:

```
AMASS(6,1,3) = -2.
AMASS(6,2,3) = 5.139
AMASS(6,3,3) = -2.
AMASS(6,4,3) = 5.139
AMASS(6,5,3) = -2.
AMASS(6,6,3) = 5.139
AMASS(6,7,3) = -2.
AMASS(6,8,3) = 5.139
AMASS(6,9,3) = 6.9
AMASS(6,10,3) = 10.
```

ARRAY A

```
A(5) = 4.1
A(6) = 4.1
A(7) = 5.0
```

ARRAY B:

```
B(1) = 0.0
B(2) = 0.0
B(3) = 0.0
B(4) = 0.0
B(5) = 0.0
```

Example 5:

Invalid: `DATA (A(I), B(I), I=1, 3)/1., 2., 3., 4., 5., 6./`

Example 6:

The statements:

```
DIMENSION D3(4),POQ(5,5)
DATA (D3 = 5.,6.,7.,8.),(((POQ(I,J),I=1,5),J=1,5)=25*0.)
```

Initialize:

D3(1) = 5.
D3(2) = 6.
D3(3) = 7.
D3(4) = 8.

and set the entire array POQ to zero.

# FLOW CONTROL STATEMENTS 4

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

| | |
|---|---|
| ASSIGN | PAUSE |
| GO TO | STOP |
| IF | END |
| DO | RETURN |
| CONTINUE | |

Control can be transferred only to an executable statement.

A statement can be identified by an integer, 1-99999, 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, computed, and assigned. The ASSIGN statement is used in conjunction with the assigned GO TO and is therefore described in the GO TO statement group.

### UNCONDITIONAL GO TO STATEMENT

```
      7
      GO TO sn
```

sn       is a label of an executable statement.

This statement transfers control to the statement labeled sn which must be an executable statement in the current program unit.

Example:

```
 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
```

## COMPUTED GO TO STATEMENT

7
GO TO $(sn_1, sn_2, \ldots, sn_m)$, iv

7
GO TO $(sn_1, sn_2, \ldots, sn_m)$, eam

7
GO TO $(sn_1, sn_2, \ldots, sn_m)$ iv

7
GO TO $(sn_1, sn_2, \ldots, sn_m)$ eam

$sn_i$ is a label on an executable statement.

iv is an integer variable.

eam is an arithmetic or masking expression.

The computed GO TO statement transfers control to one of the statements referenced in the parentheses. If the variable iv has a value of one, control transfers to the statement labeled $sn_1$; if the value is i, control transfers to the statement labeled $sn_i$.

The variable iv can be replaced by an expression. The value of the expression is truncated and converted to an integer, if necessary, and used in place of iv. The comma separating the statement label from the variable or expression is optional.

The variable must not be specified by an ASSIGN statement. If it is specified by an ASSIGN statement, the object code is incorrect, but no compilation error message is issued.

If the value of the variable or expression is less than one or larger than the number of statement numbers in parentheses, the transfer of control is undefined and a fatal error results at execution time.

Example 1:

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

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`<br>`GO TO(100,150,300),K` | Statement 150 is executed next. |

Example 3:

| | |
|---|---|
| `K=2`<br>`X=4.6`<br>`.`<br>`.`<br>`.`<br>`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
```

Execution of the last example causes a fatal error during execution because fewer than four numbers are specified in the list of statement labels.

## ASSIGN STATEMENT

```
7
ASSIGN sn TO iv
```

sn    is a label of an executable statement.

iv    is an integer variable.

The ASSIGN statement assigns a statement label to a variable used in an assigned GO TO. The integer constant assigned to iv represents the label of an executable statement to which control may be transferred by an assigned GO TO statement. Once iv is used in an ASSIGN statement, it must not be referenced in any statement, other than an assigned GO TO or another ASSIGN, until it has been redefined.

The assignment must be made prior to the execution of the assigned GO TO statement and sn (the label of an executable statement) must be in the same program unit as both the ASSIGN and assigned GO TO statements.

Example:

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

Control transfers to statement 10

## ASSIGNED GO TO STATEMENT

```
      7
      GO TO iv, (sn1,...,snm)
```

```
      7
      GO TO iv (sn1,...,snm)
```

iv — is an integer variable.

$(sn_1, \ldots, sn_m)$ — is a list of all the statement labels to which control can be passed by this assigned GO TO. Upon execution of the assigned GO TO, iv must be assigned to one of the labels in the list.

The assigned GO TO statement transfers control to the statement label last assigned to iv by the execution of a prior ASSIGN statement. All the statement labels in the list must be in the same program unit with both the ASSIGN and the assigned GO TO statements. Omitting the list of statement labels causes a fatal error. If a statement label is omitted from the list or the value of iv is defined by a statement other than an ASSIGN statement, the results are unpredictable. (Control is transferred to the absolute memory address represented by the low order 18 bits of iv.) The comma after iv is optional.

Example:

```
   ASSIGN 50 TO JUMP
10 GO TO JUMP,(20,30,40,50)
   .
20 CONTINUE
   .
30 CAT=ZERO+HAT
   .
   .
   .
40 CAT=10.1-3.
   .
   .
   .
50 CAT=25.2+7.3
```

Statement 50 is executed immediately after statement 10.

# ARITHMETIC IF STATEMENT

The arithmetic IF statement has a three-branch and a two-branch form. In both cases, zero is defined as a word containing all bits set to zero or all bits set to one (+0 or -0). If the type of the evaluated expression is complex, only the real part is tested.

## THREE-BRANCH ARITHMETIC IF STATEMENT

```
      7
      IF (eam) sn1, sn2, sn3
```

eam — is an arithmetic or masking expression.

$sn_1, sn_2, sn_3$ — are labels on executable statements.

The three-branch IF statement transfers control to the statement labeled $sn_1$ if the value of the expression is less than zero, to the statement labeled $sn_2$ if it is equal to zero, or to the statement labeled $sn_3$ if it is greater than zero.

Example:

```
      PROGRAM IF (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      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
```

## TWO-BRANCH ARITHMETIC IF STATEMENT

```
      7
      IF (eam) sn1, sn2
```

eam — is an arithmetic or masking expression.

$sn_1, sn_2$ — are labels on executable statements.

The two-branch IF statement transfers control to one of two executable statements. Control is transferred to the statement labeled $sn_1$ if the value of the expression is not equal to zero and to the statement labeled $sn_2$ if it is equal to zero.

Example:

```
      IF (I*J*DATA(K))100,101
  100 IF (I*Y*K)105,106
```

## LOGICAL IF STATEMENT

The logical IF statement has a standard form and a two-branch form.

### STANDARD-FORM LOGICAL IF STATEMENT

```
      7
      IF (elr) stat
```

elr   is a logical or relational expression.

stat   is any unlabeled executable statement other than DO, END, or another standard-form logical IF.

The standard-form logical IF allows for conditional execution of a statement. If the logical or relational expression is true, stat is executed. If the expression is false, stat is skipped.

Examples:

```
      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.LE. 2.5) CASH=150.
   70 B=A+C-TEMP
```

If A is less than or equal to 2.5, the value of CASH is replaced by 150. If A is greater than 2.5, CASH remains unchanged.

```
      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.

## TWO-BRANCH LOGICAL IF STATEMENT

```
      7
      IF (elr) sn1,sn2
```

elr    is a logical or relational expression.

$sn_1, sn_2$    are labels on executable statements.

The two-branch logical IF allows for transfer of control to one of two executable statements. If the value of the logical or relational expression is true, control is transferred to the statement labeled $sn_1$. If the value of the expression is false, control is transferred to the statement labeled $sn_2$.

Example:

```
IF(K.EQ.100)60,70
```

If K is equal to 100, statement 60 is executed; otherwise statement 70 is executed.

## DO STATEMENT

```
      7
      DO sn iv=m1,m2,m3
```

```
      7
      DO sn iv=m1,m2
```

sn    Terminal statement label; 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 any arithmetic or two-branch logical IF, a GO TO, RETURN, END, STOP, PAUSE, or another DO statement.

iv    Control variable; an integer variable.

| | | |
|---|---|---|
| $m_1$ | Initial parameter. | Indexing parameters: unsigned integer or octal constants or integer variables with positive non-zero values at execution such that neither $m_1+m_3$ nor $m_2+m_3$ is larger than $2^{17}-1$. If the indexing parameters exceed these constraints, the results are unpredictable. If $m_3$ is not specified, its value is assumed to be 1. |
| $m_2$ | Terminal parameter. | |
| $m_3$ | Incrementation parameter. | |

The DO statement makes it possible to repeat groups of statements and to change the value of an integer variable during the repetition.

## DO LOOPS

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

1. iv is assigned the value of $m_1$.

2. The range of the DO loop is executed.

3. iv is incremented by the value of $m_3$.

4. iv is compared with $m_2$. If the value of iv is less than or equal to the value of $m_2$, the sequence of operations starting at step 2 is repeated. If the value of iv is greater than the value of $m_2$, the DO is said to have been satisfied, the control variable becomes undefined, and control passes to the statement following sn. If $m_1$ is greater than or equal to $m_2$, the range of the DO loop is executed once.

A transfer out of the range of a DO loop is permissible at any time. When such a transfer occurs, the control variable remains defined at its most recent value in the loop. If control eventually is returned to the same range, the statements executed while control is out of the range are said to define the extended range of the DO. The extended range should not contain a DO statement. Subroutines or functions invoked within the range of a DO can contain DO statements, however.

The control variable must not be redefined in the range of a DO; such redefinition causes a fatal-to-execution diagnostic to be issued. The control variable should likewise not be redefined in the extended range; such redefinition causes the results of execution to be unpredictable.

The indexing parameters should not be redefined in either the range or the extended range of a DO. In either case, the results of execution are unpredictable. Redefinition in the range of the DO causes an informative diagnostic to be issued.

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, 10. Statement 300 is then executed.

Example 2:

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

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-IVAR) 20,10,10
   20 CONTINUE
   10 IN = I
```

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 9.

Example 4:

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

The DO loop is executed only once (with I = 5) because J is larger than K.

## NESTED DO LOOPS

When a DO loop entirely contains another DO loop, the grouping is called a DO nest. DO loops can be nested to 50 levels. 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 statement.

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 only from within the range (or extended range) of the innermost DO, and the label cannot be referenced in any GO TO or IF statement in the nest except in the range of the innermost DO.

A DO loop can be entered only through 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.


The use of and return from a subprogram within a DO loop are permitted. A transfer back into the range of an innermost DO loop is allowed if a transfer has been made from the same loop.

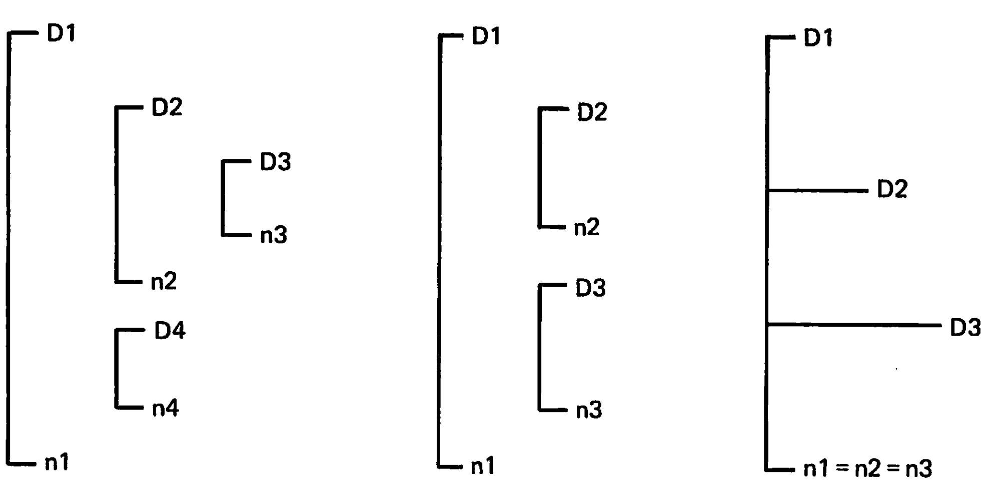

The extended range of an inner DO loop must be outside the outermost DO loop.

Example 1:

```
    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.

Example 2:



DO loops can be nested completely within an outermost loop or can share a terminal statement. The diagrams in example 2 might be represented by the following code:

```
      DO 1 I=1,10,2
      .
      .
      DO 2 J=1,5
      .
      .
      .
      DO 3 K=2,8
      .
      .
      .
    3 CONTINUE
      .
      .
      .
    2 CONTINUE
      .
      .
      .
      DO 4 L=1,3
      .
    4 CONTINUE
      .
    1 CONTINUE
```

```
      DO 100 L=2,LIMIT
      .
      .
      DO 10 J=1,10
      .
      .
      .
   10 CONTINUE
      .
      .
      .
      DO 20 K=K1,K2
      .
      .
      .
   20 CONTINUE
      .
      .
      .
  100 CONTINUE
```

```
      DO 5 I=1,5
      DO 5 J=I,10
      DO 5 K=J,15
      .
      .
      .
    5 A = B*C
```

Example 3:

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

Since statement 10 is the terminal statement for more than one DO loop, it can be referenced in a GO TO or IF statement in the range of the innermost DO. If 10 is referenced in one of the outer loops, control is transferred out of the range with undefined results.

Example 4:

```
      DO 10 K=1,100
      IF(DATA(K)-10.)20,10,20
   20 DO 30 L=1,20
      IF(DATA(L)-FACT*K-10.)40,30,40
   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
      GO TO 103
  104 CONTINUE
```

When an IF statement is used to bypass several inner loops, different terminal statements are required for each loop.

## CONTINUE STATEMENT

```
    5 7
   sn CONTINUE
```

sn is a statement label.

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 provided.

Example 1:

```
      DO 10 I = 1,11
      IF (A(I)-A(I+1))20,10,10
   20 ITEMPP = A(I)
      A (I) = A (I+1)
   10 CONTINUE
```

Example 2:

```
      DO 20 I=1,20
    1 IF (X(I) - Y(I))2,20,20
    2 X(I)=X(I)+1.0
      Y(I)=Y(I)-2.0
      GO TO 1
   20 CONTINUE
```

## PAUSE STATEMENT

```
      7
      PAUSE
```

```
      7
      PAUSE n
```

```
      7
      PAUSE ≠c...c≠
```

n is a string of 1-5 octal digits.

c...c is a string of 1-70[†] characters.

When a PAUSE statement is encountered during execution, the program halts and PAUSE n, or PAUSE c...c, appears as a dayfile message on the operator console, and at the user terminal[††] if the job is executing interactively. 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, 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 the NOS 1 Time-Sharing System, the user types STOP[†††] to terminate execution. Any other type-in causes execution to continue.

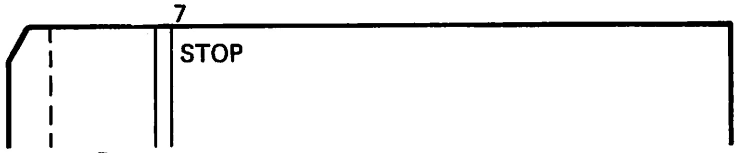
---

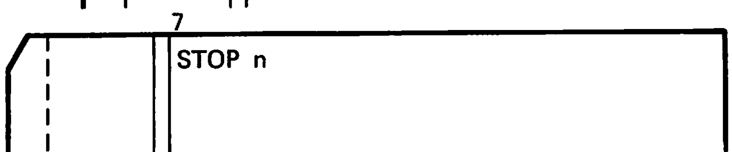
[†] Only 40 characters for SCOPE 2.

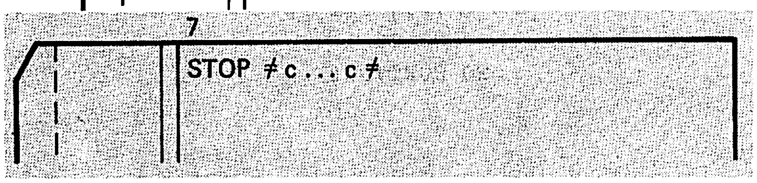
[††] Does not apply to SCOPE 2.

[†††] Applies to TELEX only. For IAF, a terminating character must be used; for most terminals the terminating character is CTRL/T or the ")" character.

## STOP STATEMENT



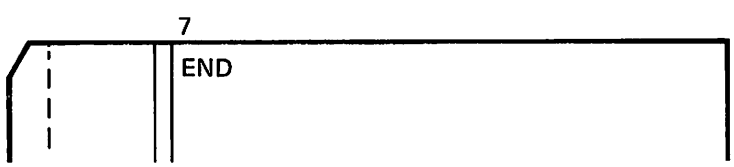




n      is a string of 1-5 octal digits.

c. . . c   is a string of 1-70 characters.

The STOP statement terminates program execution. When a STOP statement is encountered during execution, STOP n or STOP c. . . c is displayed in the dayfile, 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.

## 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 follow a $ statement separator, be labeled, and be continued. If control flows into or branches to an END statement, it is treated as if a RETURN statement had preceded the END statement.

If the END statement is not continued (all three characters are on the same line) no scanning for possible continuation information is performed and any information after the END statement is considered part of the next program unit. If the END statement is continued (all three characters not on one line), any comment statements and blank lines following the END statement are listed with the current program unit.

The following examples are interpreted as the end of one program unit, followed by another program unit beginning with an illegal continuation line of either . FILE 3 or . = 4.

```
    END                END
   .FILE 3             . = 4
```

## RETURN STATEMENT

```
RETURN
```

```
RETURN i
```

i is a dummy argument which appears in the RETURNS list in the SUBROUTINE statement.

The RETURN statement terminates the execution sequence within a program unit and normally returns control to the current calling program unit. In a main program, execution of the program terminates and control returns to the operating system when a RETURN is encountered.

When a RETURN statement is encountered in a function subprogram, control returns to the referencing program unit and the evaluation of the expression is completed using the value returned from the function. Since control must return to the referencing expression, a RETURN i statement in a function subprogram causes a fatal error at compilation time.

In a subroutine subprogram, a RETURN statement transfers control to the next executable statement following the CALL statement in the calling program unit.

A RETURN i in a subroutine transfers control to the calling program statement label corresponding to i in the RETURNS list. It allows control to return to an executable statement other than the one immediately following the CALL statement and can only be used in a subroutine subprogram.

The RETURNS list is described in more detail under Subroutine Subprogram and Calling a Subroutine Subprogram in section 7.

Example 1:

```
   A = SUBFUN (D,E)          FUNCTION SUBFUN(X,Y)
10 DO 200 I = 1,5            SUBFUN = X/Y
   .                         RETURN
   .                         END
   .
```

When the RETURN statement is encountered in the function subprogram, control is returned to the statement referencing the subprogram, and the value calculated by SUBFUN is stored in A.

Example 2:

**Calling Program**

```
     .
     .
     .
     CALL PGM1(A,B,C),
    XRETURNS (5,10)
     .
     .
     .
 5   B=SQRT(A*C)
     .
     .
     .
10   CALL PGM2 (D,E)
     .
     .
     .
```

**Subprogram**

```
     .
     .
     .
     SUBROUTINE PGM1(X,Y,Z),
    XRETURNS (M,N)
     U=X**Y
     X=Z+X*Y
  20 IF (U+X) 25, 30, 35
  25 RETURN M          Return is to statement 5 in calling program.
  30 RETURN N          Return is to statement 10 in calling program.
  35 Z=Z+(X*Y)
     RETURN            Return is to statement following CALL PGM1.
     END
```

Example 2 shows both forms of the RETURN statement in a subroutine subprogram.

# INPUT/OUTPUT STATEMENTS 5

Processing resulting 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 designator)

Unformatted (READ and WRITE without format designator)

List-directed (READ, WRITE, PRINT and PUNCH with an asterisk as the format designator)

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

Buffered (BUFFERIN and BUFFEROUT)

Mass storage input/output (Subroutines READMS, WRITMS, OPENMS, CLOSMS, and STINDX; see section 8)

CYBER Record Manager interface routines (see section 8)

In addition, there are the core-to-core transfer statements ENCODE and DECODE and the file motion statements REWIND, BACKSPACE, and ENDFILE, all 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 8. Format specifications and input/output lists are discussed in section 6.

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, the first 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 | PUNCH | Punched in Hollerith format at job termination |
| OUTPUT | Printed 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.

All files used by input/output statements or the mass storage subroutines must be declared on the PROGRAM statement (discussed in section 7). Files processed by CYBER Record Manager routines, however, must not be declared on the PROGRAM statement. The PROGRAM statement also allows 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 subroutines should be processed only by these routines. Files processed by buffer statements should be processed only by these statements in a given program (REWIND, ENDFILE, and BACKSPACE are permitted for these files).

A file should not be processed both by unformatted operations on the one hand and by formatted, namelist, or list directed operations on the other. However, if a file is 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 (see section 16) must be included in the job.

After every formatted, list directed, namelist, or unformatted READ, end-of-file status should be checked by a call to the EOF function (section 8). If end-of-file is encountered, and EOF is not called, the contents of the input/output list are undefined.

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. (Under the NOS 1 Time-Sharing System, the first character is printed.) The second character appears in the first print position. Carriage control characters are listed in section 6.

The following mnemonics are used throughout this manual in descriptions of input/output statements and subprograms:

u
: Input/output unit designator, used to determine the logical file name of the file to be used for input and output. The file name is derived from u depending on its value. The value can be one of the following:

  Integer constant of one or two digits (leading zeros are discarded). The compiler associates these numbers with file names of the type TAPEu, where u is the file designator (refer to PROGRAM statement, section 7).

  Simple integer variable name with a value of:

  0 - 99 or

  A display code file name (L format, left justified with binary zero fill). This is the logical file name.

fn
: Format designator; a FORMAT statement number or the name of an array, variable, or array element containing the format specification. The statement number must identify a FORMAT statement in the program unit containing the input/output statement.

iolist
: Input/output list specifying items to be transmitted (section 6).

## FORMATTED INPUT/OUTPUT

For formatted input/output, a format designator must be present in the input/output statement. The input/output list is optional. Each formatted input/output statement transfers one or more records.

## FORMATTED OUTPUT STATEMENTS

### PRINT

```
      7
      PRINT fn,iolist
```

```
      7
      PRINT fn
```

```
      7
      PRINT(u,fn) iolist
```

```
      7
      PRINT(u,fn)
```

This statement transfers information from the storage locations named in the input/output list to the file named OUTPUT (if u is omitted) or the file specified by u, according to the specification in the format designator fn. At the end of a job, if the user has not specified an alternate assignment, the file OUTPUT is sent to the printer.

```
    5 7
      PROGRAM PRINT (OUTPUT)
      A=1.2
      L=3HYES
      N=19
      PRINT 4,A,L,N
    4 FORMAT (G20.6,A10,I5)
```

The iolist can be omitted. For example,

```
      PRINT 20
   20 FORMAT (30H THIS IS THE  END OF THE REPORT)
```

**PUNCH**

```
      7
      PUNCH fn,iolist
```

```
      7
      PUNCH fn
```

```
      7
      PUNCH(u,fn) iolist
```

```
      7
      PUNCH(u,fn)
```

Data is transferred from the storage locations specified by iolist to the file PUNCH (if u is omitted) or the file specified by u. At the end of a job, if the user has not specified an alternate assignment, the file PUNCH is output on the standard punch unit as Hollerith codes, 80 characters or less per card in accordance with format specification fn. If the card image is longer than 80 characters, a second card is punched with the remaining characters.

```
    5 7
      PROGRAM PUNCH (INPUT,OUTPUT,PUNCH)
    2 READ 3,A,B,C
    3 FORMAT (3G12.6)
      ANSWER = A + B - C
      IF (A .EQ. 99.99) STOP
      PRINT 4, ANSWER
    4 FORMAT (G20.6)
      PUNCH 5,A,B,C,ANSWER
    5 FORMAT (3G12.6,G20.6)
      GO TO 2
      END
```

The iolist can be omitted. For example,

```
      PUNCH 30
   30 FORMAT (10H LAST CARD)
```

**WRITE**

```
WRITE (u,fn) iolist
```

```
WRITE (u,fn)
```

```
WRITE fn,iolist
```

```
WRITE fn
```

The formatted WRITE statement transfers information from the storage locations named in the input/output list to the file named OUTPUT (if u is omitted) or the file specified by u, according to the FORMAT specification, fn. At the end of a job, if the user has not specified an alternate assignment, the file OUTPUT is sent to the printer.

```
      PROGRAM RITE (OUTPUT,TAPE6=OUTPUT)
      X=2.1
      Y=3.
      M=7
      WRITE (6,100) X,Y,M
  100 FORMAT (2F6.2,I4)
      STOP
      END
```

The iolist can be omitted. For example,

```
      WRITE (4,27)
   27 FORMAT (32H THIS COLUMN REPRESENTS X VALUES)
```

## FORMATTED READ

```
READ (u,fn) iolist
```

```
      7
READ (u,fn)
```

```
      7
READ fn,iolist
```

```
      7
READ fn
```

These statements transmit data from unit u, or the file INPUT (if u is omitted), to storage locations named in iolist according to FORMAT specification fn. The number of words 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 FORTRAN records will be bypassed. The number of records bypassed is one plus the number of slashes interpreted in the FORMAT statement. Except for information read into H specifications in the FORMAT statement, the data in the records skipped is ignored.

The user should test for an end-of-file after each READ statement to avoid input/output errors. If an attempt is made to read on unit u and an EOF was encountered on the previous read operation on this unit, execution terminates and an error message is printed. (Refer to section 8, EOF Function.)

Example 1:

```
      PROGRAM IN (INPUT,OUTPUT,TAPE4=INPUT,TAPE7=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
```

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 (INPUT,OUTPUT)
      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 the file INPUT to the variables X, Y, and Z, according to the specifications in the FORMAT statement labeled 5.

## 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. Unformatted input/output cannot take place with coded tapes.

### UNFORMATTED WRITE

```
      7
      WRITE (u) iolist
```

```
      7
      WRITE (u)
```

Example:

```
PROGRAM OUT(OUTPUT,TAPE10)
DIMENSION A(260),B(4000)
.
.
.
WRITE (10) A,B
END
```

This statement is used to output binary records. Information is transferred from the list variables, 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.

### UNFORMATTED READ

```
      7
      READ (u) iolist
```

```
      7
      READ (u)
```

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. 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, READ (u) spaces over one record.

```
PROGRAM AREAD (INPUT,OUTPUT,TAPE2)
READ (2) 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

```
      7
      READ(u,*) iolist
```

```
      7
      READ*, iolist
```

Data is transmitted from unit u or the file INPUT (if u is omitted) 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 continues with the same data record when no formatted READ statements have intervened. If, however, a formatted READ statement has intervened, the remainder of the original record is destroyed. This is because a single working storage area is created for all formatted, list directed, and NAMELIST input/output files. For files referenced in list directed READ statements, the maximum record length in characters should be specified in the PROGRAM statement (section 7). This specification creates a separate working storage area for the file, which is different from the default area.

When a list directed READ follows a formatted READ or a formatted READ follows a list directed READ, a new data record is always read. Unformatted input/output does not require a working storage area and therefore does not affect list directed input/output.

Input data consists of a string of values separated by one or more blanks, or by a comma or slash, either of which may 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 Hollerith values. 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 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. |
| Hollerith values | A string of characters (which may include blanks) enclosed by two ≠ characters. The ≠ character can be represented within the string by two successive ≠ characters. Hollerith values can only be read into integer variables or arrays. Values less than 10 characters long are left justified and the word blank filled. Values longer than 10 characters are truncated to 10 characters. |

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 specification of a repeated constant.

A null may be input in place of a constant when the value assigned to 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 or slashes separated by an arbitrary number of blanks. Nulls may 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; when the next input statement is executed for this specified unit, the character following the slash becomes the first input character for the second READ. When the list is exhausted and no slash has been encountered, the next list directed read on the same unit begins after the following value separator.

Input values must correspond in type to variables in the input/output list. Although the format of a real value can be the same as that of an integer value, a repeated integer constant should not be read into variables of different types.

For example:

```
READ(5,*) I, J, X, Y
```

can read correctly:

```
2*7, 2*7 but not 4*7
```

assuming that I and J are integer and X and Y are real.

Repeated constants or repeated null values should be used entirely by one read.

The only Hollerith constants permitted are those enclosed in the symbol ≠. Hollerith constants can contain embedded blanks. The paired symbols ≠≠ can be used to represent a single ≠ within a character constant. A character cannot have a repeat count associated with it, and it must be read into an integer variable or array. A character constant of less than 10 characters is padded on the right with blanks to fill the word. Only the first 10 characters are used if the constant exceeds 10 characters.

Example 1:

```
PROGRAM LDR(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
NAMELIST/OUT/CAT,BIRD,DOG
READ(5,*)CAT,BIRD,DOG
WRITE(6,OUT)
STOP
END
```

Input:

```
13.3, -5.2, .01
```

Output:

```
$OUT

CAT      = .133E+02,

BIRD     = -.52E+01,

DOG      = .1E-01,

$END
```

Example 2:

```
      PROGRAM LDIN(INPUT=100/80,OUTPUT=100)
      CALL CONNEC(5LINPUT)
      CALL CONNEC(6LOUTPUT)
      WRITE 8
8     FORMAT(" INPUT:",14X,"OUTPUT:"//)
11    READ *,J,K
      IF(EOF(5LINPUT).NE.0) STOP
      PRINT 22,J,K
22    FORMAT(T20,2I5)
      GO TO 11
      END
```

```
INPUT:                          OUTPUT:

1  2      A pair of numbers

                                1        2
,3/       Comma divides 2 and 3

                                3        2
,4/       Null before comma (after previous slash)

                                3        4
,,        Two nulls

                                3        4
,,        Two nulls

                                3        4
,5

                                3        5
,,        One delimiter, one null
6         Second input value

                                3        6
/         Terminates for the null and 6
/         Null list

                                3        6
1*,7      One null, then a 7

                                3        7
2*8       Two 8's

                                8        8
```

```
1 2,3/,4/,,,,,5,,6/1*,7 2*8    All together now

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

## LIST DIRECTED OUTPUT

```
      7
      WRITE(u,*) iolist
```

```
      WRITE*,iolist
```

```
      PRINT (u,*) iolist
```

```
      PRINT*,iolist
```

```
      PUNCH(u,*) iolist
```

```
      PUNCH*,iolist
```

Data is transferred from storage locations specified by the iolist to the designated file in a manner consistent with list directed input.

PRINT and WRITE both output data to the file OUTPUT if no unit designator is specified. PUNCH outputs to the file PUNCH if no unit designator is specified.

For files referenced in list directed WRITE, PRINT, and PUNCH statements, the maximum record length in characters should be specified in the PROGRAM statement (section 7).

List directed output is consistent with the input; however, null values, as well as slashes and repeated constants 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.

If a list item is an integer variable, array or array element, it is output in either integer or Hollerith format, depending on its contents. If the upper 12 bits are all zeros (for positive numbers) or all ones (for negative numbers) the item is output in integer format. In all other cases, the item is output in Hollerith format (delimited by ≠ characters). For an array, only the first element is checked, and the whole array is output accordingly. Therefore, a left-justified character string stored in an integer variable is usually output in Hollerith format. Also, no integers with absolute value greater than $2^{48}-1$ can be written by list directed output statements.

For list directed PRINT statements, a blank is output as the first character (carriage control) of each record and also as the first character when a long record is continued on another line; for list directed WRITE statements, a blank is output as the first character of each record only.

List directed WRITE statements include the ≠ symbols with the character output; therefore, they should be used if the list directed record output is to be input subsequently with a list directed READ statement.

On a connected file under NOS 1, 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 SCOPE 2 and NOS/BE 1, a comma as the last character of an iolist is ignored.

Example 1:

```
PROGRAM H(OUTPUT=/80)
X = 3.6
PRINT*,≠THE VALUE OF SQRT(≠, X, ≠) IS =≠, SQRT(X)
WRITE*,≠SAME WITH WRITE, SQRT(≠, X, ≠) IS =≠ ,SQRT(X)
STOP
END
```

Output:

```
THE VALUE OF SQRT(3.6) IS =1.897366596101
≠SAME WITH WRITE, SQRT(≠ 3.6 ≠) IS =≠ 1.897366596101
```

Example 2:

```
PROGRAM LOW (OUTPUT=/80,TAPE6=OUTPUT)
INTEGER J(4)
COMPLEX Z(2)
DOUBLEPRECISION Q
DATA J,Z,Q/1,-2,3,-4,(7.,-1.),(-3.,2.),1.D-5/
WRITE(6,*)J
WRITE(6,*)Z,Q
STOP
END
```

Output:

```
1 -2 3 -4
(7.,-1.) (-3.,2.) .00001
```

Example 3:

```
PROGRAM K (INPUT,OUTPUT)
PRINT *, " TYPE IN X",
READ *, X
PRINT *, " TYPE IN Y"
READ *, Y
END
```

Terminal listing under NOS 1:

```
TYPE IN X ? 1.234
TYPE IN Y
? 5.678
```

## NAMELIST

The NAMELIST statement permits input and output of groups of variables and arrays with an identifying name. No format specification is used.

```
      7
      NAMELIST/group name1 /a1 ,...,an/ .../group namen/a1 ,...,an
```

| | |
|---|---|
| group name | Symbolic name which must be enclosed in slashes and must be unique within the program unit. |
| $a_1,...,a_n$ | List of variables or array names separated by commas. Arrays with adjustable dimensions cannot appear. |

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 may be declared only once, and it may not be used for any purpose other than a NAMELIST name in the program unit. It may appear in any of the input/output statements in place of the format number. The group name cannot, however, be used in an ENCODE or DECODE statement in place of the format number. When a NAMELIST group name is used, the list must be omitted from the input/output statement.

A variable or array name may 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. A set of data items may consist of any subset of the variable names in the NAMELIST. The value of variables not included in the subset remain unchanged. Variables need not be in the order in which they appear in the defining NAMELIST statement.

Example:

```
      PROGRAM NMLIST (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      NAMELIST/SHIP/A,B,C,I1,I2
      READ(5,SHIP)
      IF (EOF(5)) 10,20
   10 PRINT*, ≠ NO DATA FOUND ≠
      STOP
   20 IF (C .LE. 0.) 40,30
   30 A = B + C
      I1 = I2 + I1
      WRITE (6,SHIP)
   40 STOP
      END
```

Input record:

```
 2
 $SHIP A=12.2,B=20.,C=3.4,I1=8,I2=50$
```

Output:

```
$SHIP
A        = .234E+02,
B        = .2E+02,
C        = .34E+01,
I1       = 58,
I2       = 50,
$END
```

INPUT

```
      7
      READ (u,group name)
```

```
      7
      READ group name
```

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, control returns to the statement following the READ; however, a subsequent read on the same file will result in a fatal error. Consequently, a NAMELIST read should be followed by a test for end-of-file.

Data items succeeding $ NAMELIST group name are read until another $ is encountered.

Blanks must not appear:

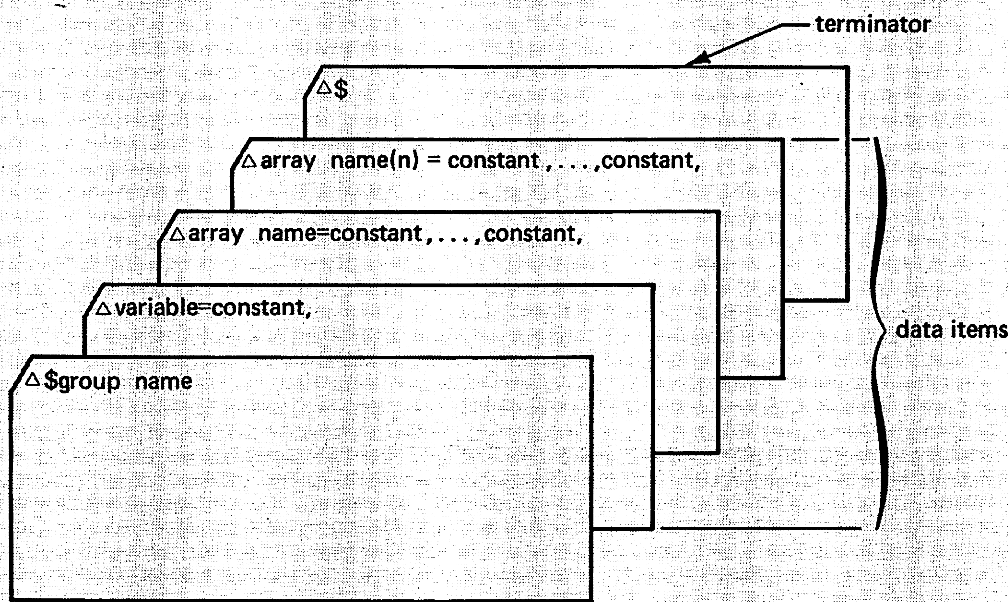
Between $ and NAMELIST group name

Within array names and variable names

Blanks may be used freely elsewhere.

More than one record can be used as input data in a NAMELIST group. The first column of each record is ignored. All input records containing data should end with a constant followed by a comma; however the last record may be terminated by a $ without the final comma.

A sample format for NAMELIST data on cards is as follows:




Data items separated by commas may be in three forms:

variable = constant

array name = constant,...,constant

array name (unsigned integer constant subscripts)=constant,...,constant

Omitting a constant constitutes a fatal error.

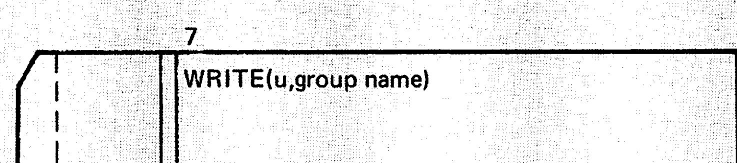
Constants can be preceded by a repetition factor and an asterisk.

Example:

`5*(1.7,-2.4)` five complex constants.

Constants may be integer, real, double precision, complex or logical. Logical constants must be of the form: .TRUE. .T. T .FALSE. .F. or F. A logical variable may be replaced only by a logical constant. A complex variable may be replaced only by a complex constant. A complex constant must have the form (real constant, real constant). Any other variable may be replaced by an integer, real or double precision constant; the constant is converted to the type of the variable.

## OUTPUT



```
      PRINT(u,group name)
```

```
      PUNCH(u,group name)
```

```
      WRITE group name
```

```
      PRINT group name
```

```
      PUNCH group name
```

All variables and arrays, and their values, in the list associated with the NAMELIST group name are output on the designated file, 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 at the top of a new page.

Example:

```
      PROGRAM NAME(INPUT,OUTPUT,TAPE5=INPUT,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
```

No data appears in column 1 of any record. If a constant would cross column 80, the columns up to and including 80 are filled with blanks instead and the constant begins in column 8?; 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 statement). Logical constants appear as T or F. Elements of an array are output in the order in which they are stored.

Records output by a WRITE (u, group name) statement may be read by a READ (u, group name) statement using the same NAMELIST name.

Example:

```
NAMELIST/ITEMS/X,Y,Z
.
.
.
WRITE (6,ITEMS)
```

Output record:

```
$ITEMS
X        = .7342E+03,
Y        = .23749E+04,
Z        = .2225E+02,
$END
```

This output may be read later in the same program using the following 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)
```

```
 2
$HAT       BAT=2,3,8*4,DOT=1.05$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:

```
 2
$DAY GAY(3)=7.2,GAY(5)=3.0,BAY=2.3,RAY=77.2$
```

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 may 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:

```
 2
$BETA ALPHA(3)=7.,8.,9.,DELTA=2.$
```

In storage

```
ALPHA(3)          7.
ALPHA(4)          8.
ALPHA(5)          9.
DELTA             2.
```

Data initialized by the DATA statement can be changed later in the program by the NAMELIST statement.

Example:

```
PROGRAM COSTS (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
DATA TAX,INT,ACCUM,ANET/23.,10,500.2,17.0/
NAMELIST/RECORDS/TAX,INT,ACCUM,ANET
FIRST = TAX + INT
SECOND = FIRST * SUM
.
.
.
READ(5, RECORDS)
.
.
.
```

Input Record:

```
 2
 $RECORDS TAX=27., ACCUM=666.2$
```

Example:

```
DIMENSION Y(3,5)
LOGICAL L
COMPLEX Z
NAMELIST/HURRY/I1,I2,I3,K,M,Y,Z,L
READ (5,HURRY)
```

Input record:

```
$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,
12.0E1,13,4*14,Z=(1.,2.),K=16,M=17$
```

produce the following values:

```
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 STATEMENTS

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 statement 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.

A file processed by buffer statements must not be processed in the same program by formatted or unformatted input/output statements, or by mass storage or CYBER Record Manager subroutines. ENDFILE, REWIND, and BACKSPACE are valid, however. The user buffer size specification in the PROGRAM statement (section 7) should be set to zero for these files; field length requirements are reduced for each file for which this is done.

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 8). 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.

Under SCOPE 2, the block of memory specified by the buffer statement is not used as the actual buffer; all buffers are maintained in an operating system buffer area in LCM. Therefore, the execution of a buffer statement results in transfer of data between system buffers in LCM and the SCM area specified in the buffer statement.

On a CYBER 170 Model 176, a FILE control statement (section 16) specifying SBF=NO must be provided if a level 2 or 3 (LCM) variable is used in a buffer statement.

### BUFFER IN

```
      7
      BUFFER IN (u,p) (a,b)
```

u   Unit designator.

p   Integer constant or simple integer variable specifying the magnetic tape data conversion mode. The parameter is applicable only when tape is assigned to the unit. Zero designates conversion (coded mode) and one designates no conversion (binary mode). For the tape file characteristic, such as parity, refer to the NOS/BE Reference Manual. Use of coded SI tapes under NOS results in a job abort by the system. Under NOS/BE, coded SI 9-track tapes are written in binary mode.

a   First variable or array element of block of memory to which data is to be transmitted.

b   Last variable or array element of block of memory to which data is to be transmitted. If u is a unit designator for 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 an input/output error occurs. Under SCOPE 2, the additional word is not needed because no error status word is written.

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.

After UNIT has been referenced, the number of words transferred to memory can be obtained by a call to the function LENGTH or the subroutine LENGTHX (section 8). 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.

For files with record type S, 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-partition). End-of-section is transparent to files with other record types.

The UNIT function should be used to test for end-of-file after BUFFER IN; the EOF function cannot be used for this purpose.

Example 1:

```
DIMENSION CALC(51)
BUFFER IN (1,0) (CALC(1),CALC(50))
```

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).

Example 2:

```
      PROGRAM TP (TAPE1=0,OUTPUT)
      INTEGER REC(513),RNUMB
      REWIND 1
      DO 4 RNUMB = 1,10000
    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 (7H0RECORD,I5/(1X,10A10))
    5 STOP
      END
```

Binary information is transferred from logical unit 1 into storage beginning at the first word of the array, REC(1), and extending through the last word of the array, 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

```
      7
      BUFFER OUT (u,p) (a,b)
```

u,p,a,b are the same as for BUFFER IN

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 is the terminal address of the record (LWA) – starting address (FWA) + 1, except for fixed-length records (RT=F on FILE statement), in which case 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.

## ENCODE AND DECODE

The ENCODE and DECODE statements are used to reformat data in memory; information is transferred under FORMAT specifications from one area of memory to another.

ENCODE is similar to a formatted WRITE statement, and DECODE is similar to a formatted READ statement. Data is transmitted under format specifications, but ENCODE and DECODE transfer data internally; no peripheral equipment is involved. For example, data can be converted to a different format internally without the necessity of writing it out on tape and rereading under another format.

### ENCODE

```
      7
      ENCODE (c,fn,v) iolist
```

v  Variable or array name which supplies the starting location of the record to be encoded.

c  Unsigned integer constant or simple integer variable specifying the length of each record.

The first record starts with the leftmost character of the location specified by v and continues for c characters, 10 characters per computer word. If c is not a multiple of 10, the record ends before the end of the word is reached; and the remainder of the word is blank filled. Each new record begins with a new computer word. There is no intrinsic limit on c, except if v is a level 2,variable c must be less than or equal to 150. (ENCODE does not write a zero-byte terminator.) Additionally the target variable area v is blank filled for c characters prior to transfer.

fn  Format designator, statement label or integer variable, which must not be a NAMELIST group name or an *.

iolist  List of variables to be transmitted to the location specified by v.

Example:

```
    5 7
      PROGRAM ENCDE (OUTPUT)
      INTEGER A(2),ALPHA(4)
      DATA A,B,C/10HABCDEFGHIJ,10HKLMNOPQRST,5HUVWXY,7HZ123456/
      ENCODE (40,1,ALPHA)A,B,C
    1 FORMAT (2A4,A5,A6)
      PRINT 2,ALPHA
    2 FORMAT (20H1CONTENTS OF ALPHA =,4A10)
      STOP
      END
```

In memory after ENCODE statement has been executed:

| ABCDKLMNUV | WXYZ12345 | | |
|---|---|---|---|
| ALPHA (1) | ALPHA (2) | ALPHA (3) | ALPHA (4) |

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 length specified by c, remaining characters in the record are blank filled.

For example, in the following program which is similar to program ENCDE above, the format statement has been changed, so that two records are generated by the ENCODE statement. A(1) and A(2) are written with the format specification 2A4, the / indicates a new record, and the remaining portion of the 40 character record, c, is blank filled. B and C are written into the second record with the specification A5 and A6, and the remaining characters are blank filled. The dimensions of the array ALPHA must be increased to 8 to accommodate two 40-character records.

```
    5 7
      PROGRAM TWO (OUTPUT)
      INTEGER A(2),ALPHA(8)
      DATA A,B,C/10HABCDEFGHIJ,10HKLMNOPQRST,5HUVWXY,7HZ123456/
      ENCODE (40,1,ALPHA)A,B,C
    1 FORMAT (2A4/A5,A6)
      PRINT 2,ALPHA
    2 FORMAT (20H1CONTENTS OF ALPHA =,8A10)
      STOP
      END
```

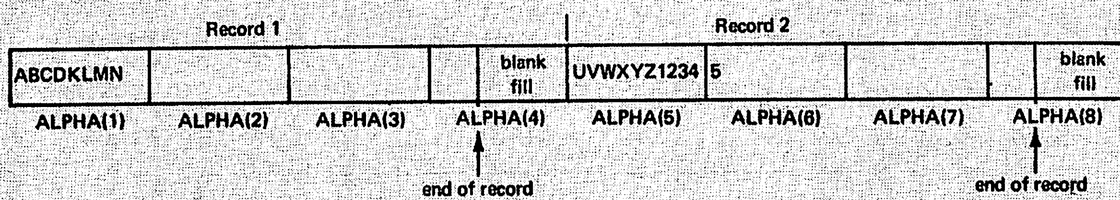
Output:

```
CONTENTS OF ALPHA =ABCDKLMN                              UVWXYZ12345
```

If this same ENCODE statement is altered to:

```
      ENCODE (33,1,ALPHA)A,B,C
    1 FORMAT (2A4/A5,A6)
```

the contents of ALPHA remain the same. When a record ends in the middle of a word the remainder of the word is blank filled (each new record starts at the beginning of the word).




The array in core must be large enough to contain the total number of characters specified in the ENCODE statement. For example, if 70 characters are generated by the ENCODE statement, the array starting at location v (if v is a single word element) must be dimensioned at least 7. If 27 characters are generated, the array must be dimensioned 3. If only 6 characters are generated, v can be a 1-word variable.

ENCODE may be used to calculate a field definition in a FORMAT specification at object time. Assume that in the statement FORMAT (2A10,Im) the programmer wishes to specify m at some point in the program. The following program permits m to vary in the range 2 through 9.

```
      IF(M.LT.10.AND.M.GT.1)1,2
    1 ENCODE (10,100,SPECMAT)M
  100 FORMAT (7H(2A10,I,I1,1H))
      .
      .
      .
      PRINT SPECMAT,A,B,J
```

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 specification I2, through I9 according to the value of m.

The following program is another example of forming FORMAT statements internally:

```
      PROGRAM IGEN (OUTPUT,TAPE6=OUTPUT)
      DO 9 J=1,50
      ENCODE (10,7,FMT)J
    7 FORMAT (2H(I,I2,1H))
    9 WRITE (6,FMT)J
      STOP
      END
```

In memory, FMT is first (I 1) then (I 2), then (I 3), etc.

A variable should not be encoded or decoded upon itself, as this gives unpredictable results.

## DECODE

```
      7
      DECODE (c,fn,v) iolist
```

c, fn, and v are the same as for ENCODE.

iolist is the list to receive variables from the location specified by v. iolist conforms to the syntax of an input/output list.

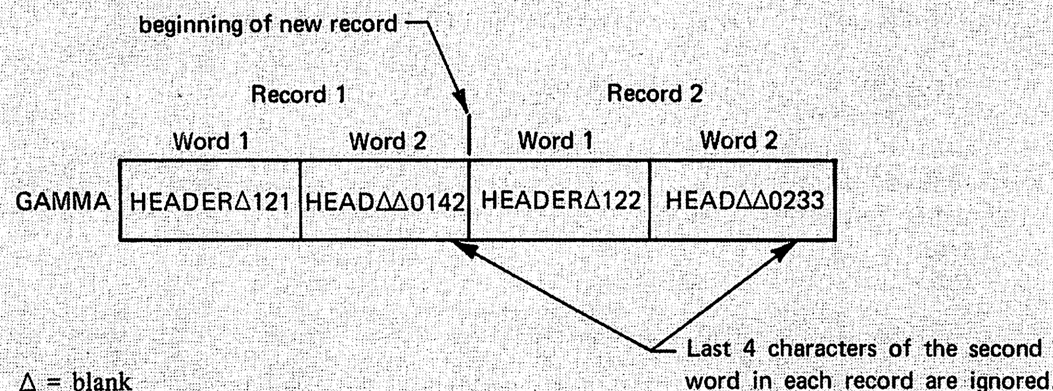
DECODE is a core-to-core transfer of data similar to formatted READ. Display code characters in a variable or an array, v, are converted under format specifications and stored in the list variables, iolist. DECODE reads from a string of display code characters in an array or variable in memory; whereas the READ statement reads from an input device. Both statements convert data according to the format specification, fn. Using DECODE, however, the same information can be read several times with different DECODE and FORMAT statements.

Starting at the named location, v, data is transmitted according to the specified format and stored in the list variables. If the number of characters per record is not a multiple of 10 (a display code word contains 10 display code characters) the balance of the word is ignored. However, if the number of characters specified by the list and the format specification exceeds the number of characters per record, an execution error message is printed. DECODE processing 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.
(Note: Internal records are not processed as zero-byte terminated records.)

Example:

c ≠ multiple of 10

```
    DECODE (16,1,GAMMA) IX,L,M,N
  1 FORMAT (2A8)
```




Data transmitted under this DECODE specification would appear in storage as follows:

```
IX=HEADER  1
 L=21HEAD
 M=HEADER  1
 N=22HEAD
```

The following illustrates one method of packing the partial contents of two words into one. Information is stored in core as:

```
LOC(1)SSSSSxxxxx
.
.
.
LOC(6)xxxxxDDDDD
```

To form SSSSSDDDDD in storage location NAME:

```
  DECODE(10,1,LOC(6)) ITEMP
1 FORMAT(5X,A5)
  ENCODE(10,2,NAME)LOC(1), ITEMP
2 FORMAT(2A5)
```

The DECODE statement places the last 5 display code characters of LOC(6) into the first 5 characters of ITEMP. The ENCODE statement packs the first 5 characters of LOC(1) and ITEMP into NAME.

Using the R specification, the example above could be shortened to:

```
  ENCODE(10,1,NAME)LOC(1),LOC(6)
1 FORMAT(A5,R5)
```

## FILE MANIPULATION STATEMENTS

Three statements can be used to manipulate files: REWIND, BACKSPACE, and ENDFILE.

### REWIND

```
      7
      REWIND u
```

The REWIND operation 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, the statement acts as a do-nothing statement. (Refer to BACKSPACE/REWIND, section 16 for further information.)

Example:

```
      REWIND 3
```

### BACKSPACE

```
      7
      BACKSPACE u
```

Unit u is backspaced one record. When the file is positioned at beginning of information, this statement acts as a do-nothing statement. If BACKSPACE is the first operation on a file positioned at beginning-of-information, a non-fatal Record Manager error results. A backspace operation should not follow a list directed read on a file.

Example:

```
      DO 1 LUN = 1,10,3
    1 BACKSPACE LUN
```

Files TAPE1, TAPE4, TAPE7, and TAPE10 are backspaced one record.

### ENDFILE

```
      7
      ENDFILE u
```

An end of partition is written on the designated unit.

Note: When ENDFILE is used on a file defined with W type records, an end-of-partition is not physically written but is marked in the control word. For all other record types a level 17 zero-length PRU is written.

To ensure file integrity, 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 = 6LOUTPUT
END FILE IOUT
```

End of partition is written on the file OUTPUT.

End of partition is the file boundary recognized by the EOF function (section 8).

For records written by an unformatted WRITE statement, an end-of-partition boundary is detected as an end of section (end-of-record) by the operating system.

# INPUT/OUTPUT LISTS AND FORMAT STATEMENTS 6

This chapter covers input/output lists and FORMAT statements. Input/output statements are covered in section 5.

## 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, a record is skipped. Only Hollerith information from the FORMAT statement can be output with a null (empty) output list.

A list item consists of a variable name, an array name, an array element name, or an implied DO list. On output the data list can include arbitrarily long Hollerith constants and arithmetic expressions. No expression in an input/output list can reference a function if such reference would cause any input/output operations (including DEBUG output) to be executed or would cause the value of any element of the input/output statement to be changed.

Multiple lists can appear, separated by commas, each enclosed in parentheses.

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.

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

Examples:

```
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)
READ (2,202)DELTA
READ (4,102)DELTA(5*J+2,5*I-3,5*K),C,D(I+7)
READ (3,2)A,(B,C,D),(X,Y)
```

An implied DO list is a list followed by a comma and an implied DO specification, all enclosed in parentheses.

An implied DO specification takes one of the following forms:

$i = m_1,m_2,m_3$          $i = m_1,m_2$

The elements i, $m_1$, $m_2$, and $m_3$ have the same meaning as in the DO statement. The range of an implied DO specification is that of the implied DO list. The values of i, $m_1$, $m_2$, and $m_3$ must not be changed within the range of the implied DO list by a READ statement.

On input or output, the list is scanned and each variable 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.

Reading more data than is in the input stream produces unpredictable values. The EOF function described in section 8 can be used to test for the end of the data.

## IMPLIED DO IN I/O LIST

Input/output of array elements may 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)
```

The general form for an implied DO loop is:

$$( \ldots ((\text{list},i_1=m_1,m_2,m_3),i_2=j_1,j_2,j_3), \ldots , i_n=k_1,k_2,k_3)$$

m,j,k are unsigned integer constants or predefined positive integer variables. If $m_3$, $j_3$ or $k_3$ is omitted, a one is used for incrementing.

$i_1 \ldots i_n$ are integer control variables. A control variable should not be used twice in the same implied DO nest, but array names, array elements, and variables may appear more than once. The value of a control variable within an implied DO specification is defined only within that specification; it should not be referenced outside the specification.

The first control variable ($i_1$) defined in the list is incremented first. $i_1$ is set equal to $m_1$ and the associated list is transmitted; then $i_1$ is incremented by $m_3$, until $m_2$ is exceeded. When the first control variable reaches $m_2$, it is reset to $m_1$; the next control variable at the right ($i_2$) is incremented; and the process is repeated until the last control variable ($i_n$) has been incremented, until $k_2$ is exceeded.

The general form for an array is:

```
(((A(I,J,K),I=m1,m2,m3),J=n1,n2,n3),K=k1,k2,k3)
```

Example:

```
READ (2,100) ((A(JV,JX),JV=2,20,2),JX=1,30)
READ (2,200) (BETA(3*JON+7),JON=JONA,JONB,JONC)
READ (2,300) (((ITMLIST(I,J+1,K-2),I=1,25),J=2,N),K=IVAR,IVMAX,4)
```

An implied DO loop can be used to transmit a simple variable more than one time. For example, the list item (A(K),B,K=1,5) causes the variable B to be transmitted five times. An input list of the form K,(A(I),I=1,K) is permitted, and the input value of K is used in the implied DO loop. The index variable in an implied DO list must be an integer variable.

Examples of simple implied DO loop list items:

```
    READ (1,400) (A(I),I=1,10)
400 FORMAT (E20.10)
```

The following DO loop would have the same effect:

```
  DO 5 I=1,10
5 READ (1,400) A(I)
```

Example:

CAT,DOG, and RAT will be transmitted 10 times each with the following iolist

```
(CAT, DOG, RAT, I=1,10)
```

Implied DO loops may be nested.

Example:

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

Equivalent to the following:

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

The list is similar to the nest of DO loops:

```
  DO 5 K=1,7
  DO 5 J=1,4
  DO 5 I=1,3
5 READ (3,100) MATRIX(I,J,K)
```

Example:

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)
```

Example:

```
      READ (2,100) ((((A(I,J,K),B(I,L),C(J,N),I=1,10),J=1,5),
     X K=1,8),L=1,15),N=2,7)
```

Data is transmitted in the following sequence:

```
A(1,1,1),    B(1,1),  C(1,2),    A(2,1,1),  B(2,1),    C(1,2)...
...A(10,1,1), B(10,1), C(1,2),    A(1,2,1),  B(1,1),    C(2,2)...
...A(10,2,1), B(10,1), C(2,2),...A(10,5,1), B(10,1),   C(5,2)...
...A(10,5,8), B(10,1), C(5,2),...A(10,5,8), B(10,15),  C(5,2)...
```

Data can be read into or written from part of an array by using the implied DO loop.

Examples:

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

Data (consisting of one constant per record) is read into the first 10 elements of the array MATRIX. The following statements would have the same effect:

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

In this example, numbers are read from unit 5, one from each record, into the elements MATRIX(1) through MATRIX(10) of the array MATRIX. The READ statement is encountered each time the DO loop is executed; and a new record is read for each element of the array. Each execution of a READ statement reads at least one record regardless of the FORMAT statement.

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

In the above statements, the implied DO loop is part of the READ statement; therefore, the FORMAT statement specifies the format of the data input and determines when a new record will be read.

If statement 100 FORMAT (F7.2) had been 100 FORMAT (4F20.10), only three records would be read.

To read data into an entire array, it is necessary only to name the array in a list without any subscripts.

Example:

```
DIMENSION B (10,15)
READ (12,13) B
```

is equivalent to

```
READ (12,13) ((B(I,J),I=1,10),J=1,15)
```

The entire array B will be transmitted in both cases.

## FORMAT STATEMENT

Input and output can be formatted or unformatted. Formatted information consists of strings of display code characters. Unformatted information consists of strings of binary word values in the form in which they normally appear in storage. A FORMAT statement or variable format specification is required to transmit formatted information.

```
    5 7
   sn FORMAT (fs1,...,fsn)
```

| | |
|---|---|
| sn | Statement label which must appear |
| $fs_1,...,fs_n$ | Set of one or more field specifications separated by commas and slashes and optionally grouped by parentheses |

Note that the syntax sn FORMAT (, that is, a statement label followed by the word FORMAT followed by a left parenthesis, is understood by the FORTRAN compiler to be a FORMAT statement, regardless of previous conditions or uses of the word FORMAT in the user program.

Example:

```
    READ (5,100) INK,NAME,AREA
100 FORMAT (10X,I4,I2,F7.2)
```

FORMAT is a non-executable statement which specifies the format of data to be moved between input/output device and main memory. It is used in conjunction with formatted input and output statements, and it may appear anywhere in the program.

The FORMAT specification is enclosed in parentheses. Blanks are not significant except in Hollerith field specifications.

Generally, each item in an input/output list is associated with a corresponding field specification 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 field specifications. Double variables correspond to one floating point field specification (D, E, F, G) or two of any other kind. The D field specification corresponds to exactly one list item or half of a complex item.

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:

```
    INTEGER N
    READ (5,100) N
100 FORMAT (F10.2)
```

A floating point number is assigned to the variable N which could cause unpredictable results if N is referenced later as an integer.

## DATA CONVERSION

The following types of data conversions are available:

| | |
|---|---|
| srEw.d | Single precision floating point with exponent |
| srEw.dEe | Single precision floating point with explicitly specified exponent length |
| srEw.dDe | Single precision floating point with explicitly specified exponent length |
| srFw.d | Single precision floating point without exponent |
| srGw.d | Single precision floating point with or without exponent |
| srDw.d | Double precision floating point with exponent |
| rIw | Decimal integer conversion |
| rIw.z | Decimal integer with minimum number of digits specified |
| rLw | Logical conversion |
| rAw | Character conversion |
| rRw | Character conversion |
| rOw | Octal integer conversion |
| rOw.z | Octal integer with minimum number of digits specified |
| rZw | Hexadecimal conversion |
| srVw.d | Variable type conversion |

E, F, G, D, I, L, A, R, O, and Z are the codes which indicate the type of conversion.

| | |
|---|---|
| w | Non-zero, 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 | Non-zero, unsigned integer constant specifying the number of digits in the exponent. |
| r | Non-zero, unsigned integer constant less than $2^{17}$-1 specifying the number of times the conversion code is to be repeated. |
| s | Optional scale factor. |
| z | Unsigned integer constant specifying the minimum number of digits to be output. |

The field width w must be specified for all conversion codes. If d is not specified for w.d, it is assumed to be zero. w must be ≥ d.

Field separators are used to separate specifications and groups of specifications. The format field separators are the slash (/) and the comma. The slash is also used to specify demarcation of formatted records.

## CONVERSION SPECIFICATION

Leading blanks are not significant in numeric input conversions; other blanks are treated as zeros. Plus signs can be omitted. An all-blank field is considered to be minus zero, except for logical input, where an all-blank field is considered to be FALSE. When an all-blank field is read with a Hollerith input specification, each blank character is translated into a display code 55 octal.

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. 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.

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(F8.3,F4.1))
```

Data of differing types may be read by the same FORMAT statement. For example:

```
10 FORMAT (I5,F15.2)
```

specifies two numbers, the first of type integer, the second of type real.

```
   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.

### Iw and Iw.z INPUT

The I conversion is used to input decimal integer constants.

```
Iw    Iw.z
```

w is a decimal integer constant designating the total number of characters in the field including signs and blanks. z is ignored on input.

The plus sign may be omitted for positive integers. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. An all blank field is considered to be minus 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 terminate execution.

Example:

```
   READ (2,10) I,J,K,L,M,N
10 FORMAT (I3,I7,I2,I3,I2,I4)
```

Input Record:

```
139|bb-15bb|18|bb7|bb|b1b4
 3     7    2   3  2   4
```

In storage:

| | |
|---|---|
| I contains 139 | L contains 7 |
| J contains -1500 | M contains -0 |
| K contains 18 | N contains 104 |

**Iw and Iw.z OUTPUT**

The I specification is used to output decimal integer values.

Iw    Iw.z

w is a decimal integer constant designating the total number of characters in the field including signs and blanks. If the integer is positive the plus sign is suppressed. Numbers in the range of $-(2^{59}-1)$ to $2^{59}-1$ ($2^{59}-1$=576 460 752 303 423 487) are output correctly.

z is a decimal integer constant designating the minimum number of digits output. Leading zeros are generated when the output value requires less than z digits. If z=0, a zero value will produce all blanks. If z=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. Not specifying z produces the same results as z=1.

The specification Iw or Iw.z outputs a number in the following format:

ba...a

| | |
|---|---|
| b | Minus sign if the number is negative, or blank if the number is positive |
| a...a | May be a maximum of 18 digits |

The output quantity is right justified with blanks on the left.

If the field is too short, all asterisks occupy the field.

Example:

```
   PRINT 10,I,J,K
10 FORMAT (I9,I10,I5.3)
```

I contains -3762
J contains +4762937
K contains +13

Result: bbb-3762|bbb4762937|bb013|

$\underbrace{\text{bbb-3762}}_{8}\underbrace{\text{bbb4762937}}_{10}\underbrace{\text{bb013}}_{5}$

1st blank taken as printer control character

Example:

```
WRITE (6,100)N,M,I
100 FORMAT (I5,I6,I9)
```

N contains +20
M contains -731450
I contains +205

Result: bb20|******|bbbbbb205|

$\underbrace{\text{bb20}}_{4}\underbrace{\text{******}}_{6}\underbrace{\text{bbbbbb205}}_{9}$

1st blank taken as printer control character

specification too small; * indicates field is too short

**Ew.d, Ew.dEe and Ew.dDe OUTPUT**

E specifies conversion between an internal real value and an external number written with exponent.

Ew.d    Ew.dEe    Ew.dDe

w is an unsigned integer designating the total number of characters in the field. w must be wide enough 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. 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.

d specifies the number of digits to the right of the decimal within the field.

e specifies the number of digits in the exponent and is limited to 6 or fewer digits.

The Ew.d specification produces output in the following formats:

| | |
|---|---|
| b.a...aE ± ee | For values where the magnitude of the exponent is less than one hundred |
| b.a...a ± eee | For values where the magnitude of the exponent exceeds one hundred |

b is a minus sign if the number is negative, and a blank if the number is positive

a...a are the most significant digits of the value correctly rounded

When the specification Ew.dEe or Ew.dDe is used, the exponent is denoted by E or D 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.

Examples:

```
      WRITE (2,10)A
   10 FORMAT (E10.3)
```

A contains -67.32 or +67.32

Result: -.673E+02 or b.673E+02

```
      WRITE (2,10)A
   10 FORMAT (E13.3)
```

Result: bbb-.673E+02 or bbbb.673E+02

If an integer variable is output under the Ew.d specification, results are unpredictable since the internal format of real and integer values differ. An integer value does not have an exponent and will be printed, therefore, as a very small value or 0.0.

### Ew.d, Ew.dEe and Ew.dDe INPUT

E specifies conversion between an external number written with an exponent and an internal real value.

Ew.d    Ew.dEe    Ew.dDe

w is an unsigned integer designating the total number of characters in the field, including 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:

$$(\text{integer subfield}) \times 10^{-d} \times 10^{\text{(exponent subfield)}}$$

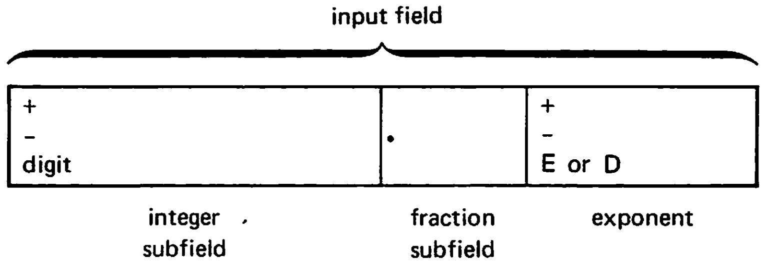
For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as: $3267 \times 10^{-8} \times 10^{5} = 3.267$.

If an external decimal point is provided, it overrides d. If d does not appear it is assumed to be zero. e, if specified, has no effect on input.

In the input data, leading blanks are not significant; other blanks are interpreted as zeros.

An input field consisting entirely of blanks is interpreted as minus zero.

The following diagram illustrates the structure of the input field:




The integer subfield begins with a + or - sign, a digit, or a blank; and it may 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 may contain a string of digits.

The exponent subfield may 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 | E 03 | E03 | E+ 3 | E3 | + 3 | +3 | D3 | D+3 | D+ 3 |
| ••••• | •••• | ••• | •••• | •• | ••• | •• | •• | ••• | •••• |

The range, in absolute value, of permissible values is $10^{-293}$ to $10^{322}$ approximately. Smaller numbers are treated as zero; larger numbers cause a fatal error message.

Valid subfield combinations:

| | |
|---|---|
| +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 onlv |

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 may be read, converted, and stored. The following example illustrates a situation where numbers are read incorrectly, converted and stored; yet there is no immediate indication that an error has occurred:

```
      READ (3,20) A,B,C
   20 FORMAT (E9.3,E7.2,E10.3)
```

On the input record, quanities are in three adjacent fields, columns 1-24:

```
+6.47E-01-2.36+5.321E+02
    9      5     10

    9        7       10

+6.47E-01
+6.47E-01-2.36+5
+6.47E-01-2.36+5.321E+02bb
```

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 number .321E+02bb is read. Trailing blanks are 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.

Examples of Ew.d input specifications:

| Input Field | Specification | Converted Value | Remarks |
|---|---|---|---|
| +143.26E-03 | E11.2 | .14326 | All subfields present |
| 327.625 | E7.3 | 327.625 | No exponent subfield |
| 4.376 | E5 | 4.376 | No d in specification |
| -.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 | Ew.d | -0. | All subfields empty |
| E+06 | E10.6 | 0. | No integer or fraction subfield; zero stored regardless of exponent field contents |
| 1.bEb1 | E6.3 | 10. | Blanks are interpreted as zeros |

#### Fw.d OUTPUT

The F specification outputs a real number without a decimal exponent.

Fw.d

w is an unsigned integer which designates the total number of characters in the field including the sign (if negative) and decimal point. w must be $\geq$ d + 2.

d specifies the number of places to the right of the decimal point. When d is zero, only the digits to the left of the decimal and the decimal point are printed.

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, an R is printed.

The specification Fw.d outputs a number in the following format:

```
      ┌──── decimal point
b...a.a...a
```

b        Minus sign if the number is negative, or blank if the number is positive.

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 | bbbb32.694 |
| -32.694 | 12 FORMAT (1H ,F6.3) | PRINT 12,A | ****** |
| .32694 | 13 FORMAT (1H ,F4.3,F6.3) | PRINT 13,A,A | .327bb.327 |

The specification 1H  is the carriage control character.

#### Fw.d INPUT

On input F specification is treated identically to the E specification.

Examples of the F format specification:

| Input Field | Specification | Converted Value | Remarks |
|---|---|---|---|
| 367.2593 | F8.4 | 367.2593 | Integer and fraction field |
| -4.7366 | F7 | -4.7366 | No d in specification |
| .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, and may have P scaling |
| 5bbbb | F5.2 | 500.00 | No fraction subfield; input number converted as $50000 \times 10^{-2}$ |
| bbbbb | F5.2 | -0.00 | Blanks in input field interpreted as -0 |

### Gw.d INPUT

Input under control of G specification is the same as for the E specification. The rules which apply to the E specification apply to the G specification.

Gw.d

w Unsigned integer which designates the total number of characters in the field including E, digits, sign, and decimal point

d Number of places to the right of the decimal point

Example:

```
   READ (5,11) A,B,C
11 FORMAT (G13.6,2G12.4)
```

### Gw.d OUTPUT

Output under control of the G specification is dependent on the size of the floating point number being converted. The number is output under the F conversion unless the magnitude of the data exceeds the range which permits effective use of the F. In this case, it is output under E conversion with an exponent.

Gw.d

w Unsigned integer which designates the total number of characters in the field including digits, signs and decimal point, the exponent E, and any leading blanks.

d Number of significant digits output.

If a number is output under the G specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field E±00). 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.

Example:

```
    WRITE (7,200) YES
200 FORMAT (G10.3)
```

YES contains 77.132

Output: `b77.1bbbb` b denotes a blank

If the decimal point is not within the first d significant digits of the number, the exponential form is used (G is treated as if it were E).

Example:

```
    WRITE (4,100) EXIT
100 FORMAT (G10.3)
```

EXIT contains 1214635.1

Output: .121E+07

Example:

```
    READ (5,50) SAMPLE
    .
    .
    .
    WRITE (6,20) SAMPLE
 20 FORMAT (1X,G17.8)
```

| Data read by READ statement | Data Output | Format Option |
|---|---|---|
| .1415926535bE-10 | .14159265E-10 | E conversion |
| .8979323846 | .89793238 | F conversion |
| 2643383279. | .26433833E+10 | E conversion |
| -693.9937510 | -693.99375 | F conversion |

## Dw.d OUTPUT

Type D conversion is used to output double precision variables. 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 = 111111.11111
      B = 222222.22222
      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:

decimal point

b.a...a $\pm$eee    $-308 \leq eee \leq 337$

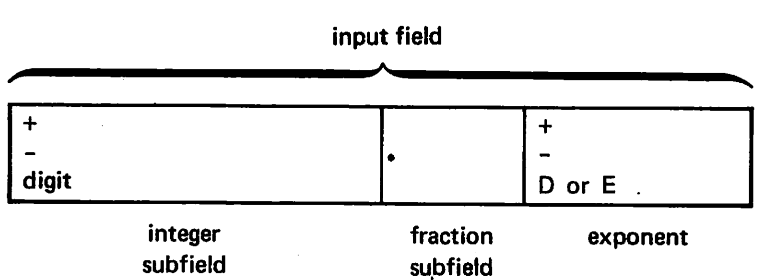
b.a...aD$\pm$ee    $0 \leq ee \leq 99$

| | |
|---|---|
| b | Minus sign if the number is negative, or blank if the number is positive |
| a...a | Most significant digits |
| ee | Digits in the exponent |

## Dw.d INPUT

D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield.

The following diagram illustrates the structure of the input field:



### Ow INPUT

Octal values are converted under the O specification.

Ow

w is an unsigned integer designating the total number of characters in the field. The input field may contain a maximum of 20 octal digits (0 through 7). Blanks are allowed and a plus or minus sign may precede the first octal digit. Blanks are interpreted as zeros and an all blank field is interpreted as minus zero. A decimal point is not allowed.

Example:

```
      INTEGER P,Q,R
      READ 10,P,Q,R
   10 FORMAT (O10,O12,O2)
```

Input Record:

$\underbrace{\texttt{3737373737}}_{10}\underbrace{\texttt{666b6644b444}}_{12}\underbrace{\texttt{-0}}_{2}$

Input storage (octal representation):

| | |
|---|---|
| P | 00000000003737373737 |
| Q | 00000000666066440444 |
| R | 77777777777777777777 |

### Ow OUTPUT

The O specification is used to output the internal representation in octal.

Ow      Ow.d

w is an unsigned integer designating the total number of characters in the field. If w is less than 20, the rightmost digits are output. For example, if the contents of location P were output with the following statement the digits 3737 would be output.

```
      WRITE (6,1) P
  100 FORMAT (1X,O4)
```

location P 00000000003737373737

If w is greater than 20, the 20 octal 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 statement

```
      WRITE (6,200) P
  200 FORMAT (1X,O22)
```

Output would appear as follows:

```
bb00000000003737373737                    b = blank
```

A negative number is output in one's complement internal form.

If d is specified, the number is printed with leading zero suppression and with a minus sign for negative numbers. At least d digits will be printed. If the number cannot be output in w octal digits, all asterisks will fill the field.

Example:

```
I = -11
WRITE (6,200) I
```

Output would appear as follows:

```
bb77777777777777777764
```

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.

### Zw INPUT and OUTPUT

Hexadecimal values are converted under the Z specification.

```
Zw
```

w is an unsigned integer designating the total number of characters in the field. The input field contains digits 0 through 9 and the letters A through F. A maximum of 15 hexadecimal digits is allowed; blanks and a plus or minus sign may precede the first hexadecimal digit.

Input values are stored right justified with zero fill. On output, if w is greater than 15, leading blanks are written. If w is less than 15, digits are taken from the right. If w is greater than the number of digits, leading zeros are written.

Example:

```
    READ 20, A,B
 20 FORMAT (Z6,Z2)
```

Input record:

```
1    6 8
AC53F93D
```

Location A:

000000000AC53F9

Location B:

00000000000003D

```
    WRITE 30,A,B
 30 FORMAT (1X,Z3,1X,Z3)
```

Output:

3F9 03D

### Aw INPUT

The A specification is used to input character data

Aw

w is an unsigned integer designating the total number of characters in the field.

Character information is stored as 6-bit display code characters, 10 characters per 60-bit word. For example, the digit 4 when read under A specification is stored as a display code 37. If w is less than 10, the input quantity is stored left justified in the word; the remainder of the word is filled with blanks.

Example:

```
    READ (5,100) J
100 FORMAT (A7)
```

Input record:

```
EXAMPLE
```

When EXAMPLE is read it is stored left justified in the 10 character word

```
1234567890
EXAMPLE
```

If w is greater than 10, the rightmost 10 characters are stored and remaining characters are ignored.

Example:

```
    READ (5,200)K
200 FORMAT (A13)
```

Input record:

```
1          13
SPECIFICATION
```

In storage:

```
12345678910
CIFICATION
```

```
   READ (5,10) L,M,N
10 FORMAT (A10,A10,A5)
```

Input record:

```
THIS IS AN EXAMPLE I KNOW
    10        10      5
```

In storage:

```
  12345678910
L THIS IS AN
M  EXAMPLE I
N  KNOW
```

## Aw OUTPUT

The A specification is used to output alphanumeric characters.

```
Aw
```

w is an unsigned integer designating the total number of characters in the field. If w is less than 10, the leftmost characters in the word are printed. For example, if the contents of location M in the Aw input example are output with the following statements:

```
    WRITE (6,300)M
300 FORMAT (1X,A4)
```

In storage:

M | E | X | A | M | P | L | E | | | |

Characters EXAM are output

If w is greater than 10, the characters are output right-justified in the field, with blanks on the left. For example, if M in the previous example is output with the following statements:

```
    WRITE (6,400)M
400 FORMAT (1X,A12)
```

Output is as follows:

bbEXAMPLEbbb    b = blank

## Rw INPUT

w is an unsigned integer designating the total number of characters in the field. The R specification is the same as the A specification unless w is less than 10. If w is less than 10, the input characters are stored right-justified, with binary zero fill on the left.

Example:

```
    READ (5,600) MOO,JAY
600 FORMAT (R10,R5)
```

Input record:

$\underbrace{\text{RESULTS OF}}_{10}\underbrace{\text{TEST}}_{5}$

In storage:

MOO | R | E | S | U | L | T | S | b | O | F |

JAY | 0 | ··· | 0 | 0 | b | T | E | S | T |    b = blank

Rw OUTPUT

Rw

w is an unsigned integer designating the total number of characters in the field.

This specification is the same as the A specification unless w is less than 10. If w is less than 10, the rightmost characters are output. For example, if JAY from the previous example is output with the following statements:

```
    WRITE (6,700) JAY
700 FORMAT (1X,R3)
```

Characters EST are output.

Lw INPUT

The L specification is used to input logical variables.

Lw

w is an unsigned integer designating the total number of characters in the field.

If the first non-blank character in the field is T, the logical value .TRUE. is stored in the corresponding list item, which should be of type logical. If the first non-blank character is F, the value .FALSE. is stored. If the first non-blank character is not T or F, a diagnostic is printed. An all blank field has the value .FALSE.

Lw OUTPUT

Lw

w is an unsigned integer designating the total number of characters in the field.

Variables output under the L specification should be of type logical. A value of .TRUE. or .FALSE. in storage 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)
```

Output:

```
bTbbFbbT
```

## SCALE FACTORS

The scale factor P is used to change the position of a decimal point of a real number when it is input or output. Scale factors may precede D, E, F and G format specifications.

| | | |
|---|---|---|
| nPDw.d | nPDw.dEe | nPDw.dDe |
| nPEw.d | nPEw.dEe | nPEw.dDe |
| nPFw.d | | |
| nPGw.d. | | |
| nP | | |

n is the scale factor which can be any integer constant. w is an unsigned integer constant designating the total width of the field. d determines the number of digits to the right of the decimal point.

A scale factor of zero is established when each FORMAT statement 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 specifications in that FORMAT statement until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G specifications, a zero scale factor (0P) must precede a specification.

Example:

```
15 FORMAT(2PE14.3,F10.2,G16.2,0P4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specification. The 0P scale factor restores normal scaling ($10^0 = 1$) for the subsequent specification 4F13.2.

A scaling factor may appear independently of a D, E, F or G specification. It holds for all subsequent D, E, F or G specifications within the same FORMAT statement,until changed by another scaling factor.

Example:

```
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).

### Fw.d SCALING

INPUT

The number in the input field is divided by $10^n$ and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is 314.1592 X $10^{-2}$ = 3.141592. However, if an exponent is read the scale factor is ignored.

OUTPUT

The number in the output field is the internal number multiplied by $10^n$. 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 may be represented on output under scaled F specifications as follows:

```
••••••••••••••••••••••••••••••
(-1PF13. 6)       -.314159
(   F13. 6)      -3.141593
( 1PF13. 6)     -31.415927
( 3PF13. 6)  -3141.592654
••••••••••••••••••••••••••••••
```

**Ew.d and Dw.d SCALING**

INPUT

Ew.d scaling on input is the same as Fw.d scaling on input.

OUTPUT

The effect of the scale factor nP is to shift the output coefficient left n places and reduce the exponent by n. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent such that: if $n \leqslant 0$, there will be exactly -n leading zeros and d + n significant digits after the decimal point; if $n > 0$, there will be exactly n significant digits to the left of the decimal point and d - n + 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 follows:

```
••••••••••••••••••••••••••••••
(-3PE20. 4)          -.0003E+04
(-1PE20. 4)          -.0314E+02
(   E20. 4)          -.3142E+C1
( 1PE20. 4)          -3.1416E+00
( 3PE20. 4)          -314.16E-C2
••••••••••••••••••••••••••••••
```

**Gw.d SCALING**

INPUT

Gw.d scaling on input is the same as Fw.d scaling on input.

OUTPUT

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 $N \geqslant 10^d$). In these cases, the scale factor has the same effect as described above 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 follows:

```
..............................     ..............................
(-3PG20. 6)          -3.14159      (-3PG20. 6)         -.000314E+00
(-1PG20. 6)          -3.14159      (-1PG20. 6)         -.031416E-02
(   G20. 6)          -3.14159      (   G20. 6)         -.314159E-03
( 1PG20. 6)          -3.14159      ( 1PG20. 6)        -3.141593E-04
( 3PG20. 6)          -3.14159      ( 3PG20. 6)        -314.1593E-06
( 5PG20. 6)          -3.14159      ( 5PG20. 6)        -31415.93E-08
( 7PG20. 6)          -3.14159      ..............................
..............................
```

## X SPECIFICATION

The X specification is used to skip characters in an input line or output line. On output, any character positions not previously filled during this record generation will be set to blank. It is not associated with a variable in the input/output list.

nX Number of characters, n, to be skipped. An optional plus sign may precede n.

0X is ignored, X is interpreted as 1X. The comma following X in the specification list is optional.

-nX Back up n characters, will not back up beyond the first column.

Example:

```
    WRITE (6,100) A,B,C                A = -342.743
100 FORMAT (F9.4,4X,F7.5,4X,I3)        B = 1.53190
                                       C = 22
```

Output record:

```
-342.743bbbb1.53190bbbbb22             b is a blank
```

on input n columns are skipped.

Example:

```
   READ (3,11) R,S,T
11 FORMAT (F5.2, 3X, F5.2, 6X, F5.2)

      or

11 FORMAT (F5.2, 3XF5.2, 6XF5.2)
```

Input record:

```
14.62bb$13.78bCOSTb15.97
```

In storage:

| | |
|---|---|
| R | 14.62 |
| S | 13.78 |
| T | 15.97 |

Example:

```
   INTEGER A
   WRITE (1,10) A,B,C
10 FORMAT (I2,6X,F6.2,6X,E12.5)
```

A contains 7
B contains 13.6
C contains 1462.37

Result: 7bbbbbbb13.60bbbbbbb.146237E+04

## nH OUTPUT

The H specification is used to output strings of alphanumeric characters and, like X, H is not associated with a variable in the input/output list.

nH

n — Number of characters in the string including blanks.

H — Denotes a Hollerith field. The comma following the H specification is optional.

For example, the statement:

```
  WRITE (6,1)
1 FORMAT (15HbENDbOFbPROGRAM)
```

can be used to output the following on the output listing.

```
END OF PROGRAM
```

Examples:

Source program:

```
   WRITE (3,20)
20 FORMAT (28HbBLANKSbCOUNTbINbANbHbFIELD.)
```

produces output record:

```
BLANKSbCOUNTbINbANbHbFIELD.
```

Source program:

```
   WRITE (2,30)A
30 FORMAT (6HbLMAX=,F5.2)
```

A contains 1.5

produces output record:

```
LMAX=b1.50
```

## nH INPUT

The H specification can be used to read Hollerith characters into an existing H field within the FORMAT statement.

Example:

Source program:

```
   READ (2,10)
10 FORMAT (27Hbbbbbbbbbbbbbbbbbbbbbbbbbbb)
```

Input record:

```
bTHIS IS A VARIABLE HEADING
```

After a READ statement, the FORMAT statement labeled 10 contains the alphanumeric information read from the input record; a subsequent reference to statement 10 in an output statement acts as follows:

```
WRITE (6,10)
```

produces the output line:

```
THIS IS A VARIABLE HEADING
```

## *...* ≠...≠

Character strings delimited by a pair of * or ≠ symbols can be used as alternate forms of the H specification for output. The paired symbols delineate the Hollerith field. This specification need not be separated from other specifications by commas. If the Hollerith field is empty, or invalidly delimited a fatal execution error occurs, and an error message is printed.

An asterisk cannot be output using the specification * *. For example,

```
  PRINT 1
1 FORMAT (*ABC*DE*)
```

The second * in the FORMAT statement causes the specification to be interpreted as *ABC* and DE*, which is not valid.

The H specification or ≠...≠ could be used to output this correctly:

```
   PRINT 1
 1 FORMAT (7H ABC*DE)
```

Output appears as follows: ABC*DE

```
   PRINT 2
 2 FORMAT (≠ ABC*DE≠)
```

Output appears as follows: ABC*DE

≠ can be represented within ≠...≠ by two consecutive ≠ symbols.

Example:

```
   PRINT 3
 3 FORMAT ( ≠DON ≠≠T ≠ )
```

Output examples:

```
   PRINT 10
10 FORMAT ( * SUBTOTALS*)
```

produces the following output:

```
   SUBTOTALS
```

```
   WRITE (6,20)
20 FORMAT ( ≠bRESULT OF CALCULATIONS IS AS FOLLOWS ≠ )
```

produces the following output:

```
   RESULT OF CALCULATIONS IS AS FOLLOWS
```

```
   PRINT 1,  ≠SQRT≠, SQRT(4.)
 1 FORMAT (A10,E10.2)
```

produces the following output:

```
   SQRT          2.0
```

Note: ≠ is output as " on some printers.

The *...* or ≠...≠ specification can be used to read alphanumeric data; however, the effect differs depending on whether *...* or ≠...≠ occurs in an actual FORMAT statement or in a format specification contained in a variable or array. When the READ statement contains a constant specifying a FORMAT statement, alphanumeric characters are read into the *...* or ≠...≠ specification. When a name occurs in the READ statement to specify the format information (variable format), characters in the input stream are skipped and no change is made in the *...* or ≠...≠ specification.

In FORMAT statements, the *...* or ≠...≠ specification is changed to nH... at compile time. This conversion does not occur with variable format specifications.

## END OF RECORD SLASH

The slash indicates the end of a record anywhere in the FORMAT specification. When a slash is used to separate field specification elements, 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, 7H OUTPUT)
```

Output:

```
        HEADING ______________ line 1
                _____ (blank) __ line 2
                _____ (blank) __ line 3
   INPUT OUTPUT ______________ line 4
```

Each line corresponds to a formatted record. The second and third records are blank and produce the line spacing illustrated.

Example:

```
      I=5
      J=6
      K=7
      WRITE (2,1)I,J,K
1     FORMAT (3I5/)
      WRITE (2,2)
2     FORMAT(* A BLANK LINE SHOULD PRECEDE THIS LINE*)
```

Output:

```
          5    6    7

      A BLANK LINE SHOULD PRECEDE THIS LINE
```

The variable list (I, J, K) is exhausted and processing continues until a variable conversion is encountered. Since the slash has been processed, it causes a blank line to be printed.

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.

```
      WRITE (3,11) A,B,C,D
   11 FORMAT (2E10.2/2F7.3)
```

In storage:

| | |
|---|---|
| A | -11.6 |
| B | .325 |
| C | 46.327 |
| D | -14.261 |

Output:

```
b-.12E+02bbb.33E+00
46.327-14.261
```

```
      WRITE (1,11) A,B,C,D
   11 FORMAT (2E10.2 / / 2F7.3)
```

Output:

```
b-.12E+02bbb.33E+00 ——————————————— line 1
     ——————————————————— (blank) ——— line 2
46.327-14.261 ————————————————————— line 3
```

The second slash causes the blank line.

## REPEATED FORMAT SPECIFICATION

Format specifications can be repeated by prefixing the control characters D, E, F, G, I, A, L, R, Z, and O with a non-zero, unsigned integer constant specifying the number of repetitions required.

| | | |
|---|---|---|
| 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 specifications 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 assumed to be one.

```
    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 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 the number of items in the input/output list is fewer than the number of format codes in the FORMAT statement, excess format codes are ignored.

If the number of items in the input/output list exceeds the number of format conversion codes 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.

A repetition factor can be used to indicate multiple slashes, n(/), where n is an unsigned integer constant indicating the number of slashes required and n-1 is the number of lines skipped on output.

Example:

```
   WRITE (3,15)(A(I),I=1,9)
15 FORMAT (8HbRESULTS4(/),(3F8.2) )
```

Format statement 15 is equivalent to: `15 FORMAT (8HbRESULTS / / / / (3F8.2) )`

Output:

```
RESULTS ________________________________________ line 1
        ______________________________ (blank)__ line 2
        ______________________________ (blank)__ line 3
        ______________________________ (blank)__ line 4
  3.62  -4.03  -9.78 ____________________________ line 5
 -6.33   7.12   3.49 ____________________________ line 6
  6.21  -6.74  -1.18 ____________________________ line 7
```

Example:

```
    READ (5,300)I,J,E,K,F,L,M,G,N,R
300 FORMAT (I3,2(I4,F7.3),I7)
```

is equivalent to storing data 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.

```
    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.

## PRINTER 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 1 Time-Sharing System; the first character is listed as data.

The printer control characters are as follows†:

| 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 |

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:

```
FORMAT (1H0,F7.3,I2,G12.6)

FORMAT (1H1,I5*RESULT = *,F8.4)

FORMAT (*1*,I4,2(F7.3) )

FORMAT (1X,I4,G16.8)
```

†This chart applies only to NOS/BE 1 and SCOPE 2. For corresponding information under NOS 1, refer to the reference manual for the subsystem under which the program is executed.

Example:

```
      PROGRAM CHARCON (OUTPUT)
      PRINT 10
   10 FORMAT (1H1,  5X, *HERE WE ARE AT THE TOP OF A NEW PAGE*)
      PRINT 20
   20 FORMAT (3(/))
      DO 30 I = 2,8
      IF (I .EQ. 4 .OR. I .EQ. 6) 40,50
   50 PRINT 60
   60 FORMAT (21X, ≠   X  X  ≠ / 1H+, 20X, ≠   ≡  ≡  ≠)
      GO TO 30
   40 PRINT 70
   70 FORMAT (20X, ≠ XXXXXXXXXX ≠ / 1H+, 19X, ≠ ≡≡≡≡≡≡≡≡≡≡ ≠)
   30 CONTINUE
      PRINT 80
   80 FORMAT (1H0, 5X, ≠BEGIN TIC TAC TOE ≠)
      STOP
      END
```

Output

```
 HERE WE ARE AT THE TOP OF A NEW PAGE




                     X  X
                     X  X
                    XXXXXXXXXX
                     X  X
                    XXXXXXXXXX
                     X  X
                     X  X

 BEGIN TIC TAC TOE
```

## Tn SPECIFICATION

This specification is tabulation control.

Tn

n Unsigned integer. If n = zero, column 1 is assumed.

When Tn is used, control skips columns right or left until column n is reached; then the next format specification is processed. Using card input, if $n > 80$, the column pointer is moved to column n, but a succeeding specification would read only blanks.

```
      READ 40, A, B, C
40    FORMAT (T1, F5.2, T11, F6.1, T21, F5.2)
```

Input:

```
      84.73bbbbb2436.2bbbb89.14.
```

A is set to 84.73, B to 2436.2, and C to 89.14.

```
      WRITE (31, 10)
10    FORMAT (T20,*LABELS*)
```

The first 19 characters of the output record are skipped and the next six characters, LABELS, are written on output unit number 31 beginning in character position 20.

With T specification, the order of a list need not be the same as the input or output record, and the same information can be read more than once.

When a T specification causes control to pass over character positions on output, positions not previously filled during this record generation are set to blanks; those already filled are left unchanged.

Example:

```
    5 7
      PROGRAM TEST (OUTPUT)
    1 FORMAT (12(10H0123456789))
      PRINT 1
      PRINT 60
   60 FORMAT (T80,*COMMENTS*,T60,*HEADING4*,T40,
     X       *HEADING3*,T20,*HEADING2*,T2,*HEADING1*)
      PRINT 10
   10 FORMAT (20X*THIS IS THE END OF THIS RUN*T52*...HONEST*)
      PRINT 1
      STOP
      END
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
HEADING1          HEADING2            HEADING3            HEADING4            COMMENTS
                   THIS IS THE END OF THIS RUN    ...HONEST
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Since the first character in a line output to the printer is used for printer control, T2 is output in the first print position.

The following example shows that it is possible to destroy a previously formed field. The specification T5 destroys part of the Hollerith specification 10H DISASTERS.

```
    1 FORMAT (10H DISASTERS, T5, 3H123)
      PRINT 1
```

produces the following output: DIS123ERS

## V SPECIFICATION†

When V is encountered in a FORMAT statement, the rightmost 6 bits from the current variable in the input/output list are interpreted as display code for a character to be used in place of the V as the conversion specification for the next variable in the input/output list. V can be used as a dummy specification for the following conversions: A, D, E, F, G, I, L, O, P, R, T, X and Z. It cannot be used as the E or D explicitly specifying exponent length; Ew.dVe is illegal.

Example:

```
      PROGRAM V (OUTPUT)
      INTEGER AFORMAT, RFORMAT
      AFORMAT = 1RA
      RFORMAT = 1RR
      NUM = 10H0123456789
      PRINT 10, AFORMAT, NUM
   10 FORMAT (T8, *FORMAT SPECIFICATION TAKEN FROM VARIABLE AFORMAT; NUM
     - OUTPUTS AS  *, V5 /)
      PRINT 20, RFORMAT, NUM
   20 FORMAT (T8, *FORMAT SPECIFICATION TAKEN FROM VARIABLE RFORMAT; NUM
     - OUTPUTS AS  *, V5)
      STOP
      END
```

Output:

```
   FORMAT SPECIFICATION TAKEN FROM VARIABLE AFORMAT; NUM OUTPUTS AS  01234

   FORMAT SPECIFICATION TAKEN FROM VARIABLE RFORMAT; NUM OUTPUTS AS  56789
```

## EQUALS SIGN†

When = is encountered in a FORMAT statement, the current variable in the input/output list supplies a positive integer value to be used in place of the = in the conversion specification for the next variable in the input/output list. The = can be used in place of a number anywhere within a FORMAT statement. Such use of = precludes compilation syntax checking of the FORMAT statement. V and = can be combined in one conversion specification.

Example:

```
      PROGRAM EQUALS (OUTPUT)
      INTEGER W(10)
      DATA W/1,2,3,4,5,6,7,8,9,10/
      NUM = 10H0123456789
      DO 10 I = 1,10
   10 PRINT 20, W(I), NUM
   20 FORMAT (T30, A=)
      STOP
      END
```

†If a V or = appears in a format specification field containing a repetition factor which does not modify a parenthesized specification list, the V or = assumes the value supplied by the first repetition, and retains that value until the current repetition count is completed.

Output:

```
0
01
012
0123
01234
012345
0123456
01234567
012345678
0123456789
```

A variable must exist in the input/output list each time an = or V is processed in the format statement.

Example:

```
      DIMENSION A(5),B(5)
      I3 = 3
      PRINT 1,I3,A,I3,B
    1 FORMAT(1X,5F10.=)
```

Two lines of five values each are printed; however, I3 must be repeated in the input/output list or the first value of B is used to replace the =.

Example:

```
      PROGRAM VEQUALS (OUTPUT)
      INTEGER FORMAT(2), W(10)
      DATA FORMAT/1RA, 1RR/, W/1,2,3,4,5,6,7,8,9,10/
      NUM = 10H0123456789
      DO 10 I = 1,2
      DO 10 J = 1,10
      K = J
      IF (I .EQ. 2) K = 11-J
   10 PRINT 20, FORMAT(I), W(K), NUM
   20 FORMAT (T20, V=)
      STOP
      END
```

```
Output:   0
          01
          012
          0123
          01234
          012345
          0123456
          01234567
          012345678
          0123456789
          0123456789
          123456789
          23456789
          3456789
          456789
          56789
          6789
          789
          89
          9
```

## EXECUTION TIME FORMAT SPECIFICATION

Variable format specifications can be read in as part of the data at execution time and used by READ, WRITE, PRINT, PUNCH, ENCODE, or DECODE statements later in the program. The format is read in as alphanumeric text under the A specification and stored in an array, simple variable or array element, or it may be included in a DATA statement. The format must consist of a list of format specifications enclosed in parentheses, but without the word FORMAT or the statement label.

For example, an input record could consist of the characters:

```
(E7.2,G20.5,F7.4,I3)
```

The name of the array containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The array name specifies the location of the first word of the format information.

For example, assume the following format specifications:

```
(E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

This information on an input record can be read by the statements of the program such as:

```
      DIMENSION IVAR(3)
      READ (2,1) IVAR
    1 FORMAT (3A10)
```

The elements of the input record are placed in storage as follows:

```
IVAR(1)        (E12.2,F8.
IVAR(2)        2,I7,2E20.
IVAR(3)        3,F9.3,I4)
```

A subsequent output statement in the same program can refer to these format specifications as:

```
WRITE (2,IVAR) A,B,I,C,D,E,J
```

Which produces exactly the same result as the statements:

```
   WRITE (2,10) A,B,I,C,D,E,J
10 FORMAT (E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

# PROGRAM UNITS, PROCEDURES, AND OVERLAYS 7

A program unit consists 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. Normally, a main program begins with a PROGRAM statement, but this statement can be omitted. A subprogram is a program unit that begins with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. An executable program contains one main program with or without subprograms. A program unit containing no FORTRAN statements other than an END statement is considered a null program; it is diagnosed and ignored.

A subprogram is defined separately and can be compiled independently of a main program. If the subprogram begins with a SUBROUTINE or FUNCTION statement, it is a procedure subprogram and can accept and use zero, one, or more values through a list of arguments, through common, or both. If the subprogram begins with a BLOCK DATA statement, it is a specification subprogram.

A procedure is a procedure subprogram, statement function, intrinsic function, or basic external function. Intrinsic functions and basic external functions are FORTRAN supplied procedures and are available to any programmer (see section 8). Statement functions and procedure subprograms are supplied by the programmer.

The differences between function and subroutine specification and use are summarized in table 7-1.

TABLE 7-1. DIFFERENCES BETWEEN A FUNCTION AND SUBROUTINE SUBPROGRAM

| | **Function** | **Subroutine** |
|---|---|---|
| How Used | The name appearing in an expression is used as the reference. | A CALL statement is used as the reference. |
| Arguments | One or more arguments must be included. | Arguments need not be present. |
| How Typed | Name is typed implicitly by first letter or explicitly by the type designation appearing before the word FUNCTION. | No type is associated with the name. |

Functions return a single value through the function name. Function subprograms defined by the programmer also can return values through a list of arguments, through common, or both.

Table 7-2 summarizes the terminology of the overlapping categories of procedures and subprograms.

TABLE 7-2. PROCEDURE AND SUBPROGRAM INTERRELATIONSHIPS

| | Statement Function | Intrinsic Function | Basic External Function | Function Subprogram | Subroutine Subprogram | Block Data Subprogram |
|---|---|---|---|---|---|---|
| Procedure | yes | yes | yes | yes | yes | no |
| External procedure | no | no | yes | yes | yes | N/A |
| Subprogram | no | no | no | yes | yes | yes |
| Function | yes | yes | yes | yes | no | no |
| External function | no | no | yes | yes | N/A | N/A |
| Who defines | user | compiler | compiler | user | user | user |
| Where defined | within program unit | compiler | library | external to calling program unit | external to calling program unit | external to calling program unit |
| N/A = not applicable | | | | | | |

Programmer written procedures (statement functions, function subprograms, and subroutine subprograms) are discussed below as a group. FORTRAN supplied procedures (intrinsic functions and basic external functions) are discussed in detail in section 8. The only subprogram that is not a procedure is the block data subprogram. Since it is not executable, it is discussed separately.

# MAIN PROGRAMS

A main program can contain any FORTRAN statements except FUNCTION, SUBROUTINE, ENTRY or BLOCK DATA; it should have a PROGRAM statement and an END statement; it must have at least one executable statement. One main program is required in any executable FORTRAN program. No program can have more than one main program, except an overlay structured program, which has one main program in each overlay.

## PROGRAM STATEMENT FORMAT

7

PROGRAM name (fpar$_1$, fpar$_2$, . . . , fpar$_n$)

name Symbolic name (1 to 7 letters and digits, beginning with a letter). Cannot be used elsewhere in the program as a user-defined name.

fpar$_i$ The fpar can be any of the following forms:

| | |
|---|---|
| file | File name (1-6 letters or digits beginning with a letter) for each file required by the main program or its subprograms; the maximum number of file names is 50. |
| file=n | n† is an integer or octal constant specifying the buffer length; default length is 2003 octal words. |
| file=/r | r is the maximum length in characters for list directed, formatted, and NAMELIST records; default limit is 150 characters. |
| file=n/r | n/r defines both buffer and record lengths. |
| $file_a$=$file_b$ | $File_a$ is made equivalent to previously defined $file_b$. |

In a program structured for overlays, the $fpar_i$ parameter list is used only in the PROGRAM statement for the main overlay. It is not used in primary and secondary overlay PROGRAM statements.

## PROGRAM STATEMENT USAGE

The PROGRAM statement defines the program name that is used as the entry point name and the object deck name for the loader. Optionally, 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.

All file names used in standard FORTRAN input/output statements (including mass storage subroutines) must be listed in the PROGRAM statement. File names referenced by CYBER Record Manager interface subroutines must not be listed in the PROGRAM statement. If a file name is referenced in a standard FORTRAN input/output statement in a main program, but is not specified in the PROGRAM statement, a warning diagnostic is issued at compile time. If a file name is referenced in a standard FORTRAN input/output statement in a subprogram, but is not specified in the PROGRAM statement of the main program, a diagnostic is issued when the file is used at execution time.

File names on the PROGRAM statement must satisfy the following conditions:

The file name INPUT must be declared if a READ statement without a unit designator is included in the program.

The file name OUTPUT must be declared if a PRINT statement without a unit designator is included in the program.

The file name PUNCH must be declared if a PUNCH statement without a unit designator is included in the program.

The file name TAPEu (u is an integer constant 0-99) must be declared if any input/output statement involving unit u appears in the program, and u is an integer value. At execution time, if u is a variable, there must be a file name TAPEu for each integer value u assumes.

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.

†n is ignored if specified in a program run under SCOPE 2.

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.

FORTRAN input/output statements use the buffer areas established by the file name specified in the PROGRAM statement. 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.

For files not referenced by BUFFER statements, the following values of n† are minimal if default record and block types are used.††

| | |
|---|---|
| For terminals: | n=number of words in the largest record plus one. |
| For mass storage input/output files: | n≥65. Large records and sequential reading/writing execute faster with a larger buffer. |

For sequential files:

| Format | Minimum Value of n |
|---|---|
| SI tape | 128 for formatted.<br>513 for unformatted. |
| I, X tape | 513 for unformatted. |
| S tape | 513 for formatted or unformatted. |
| L tape | ≥ maximum block length. |
| mass storage | 65 for formatted.<br>513 for unformatted. |

Record length, r, 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.

Care should be taken when specifying a record length for file OUTPUT that the length is not so short as to inhibit output of any run-time diagnostics.

When file names are made equivalent, the buffer length and record size specified apply to both files.

Examples:

```
PROGRAM ORB (INPUT,OUTPUT=1000,TAPE1=INPUT,TAPE2=OUTPUT,TAPE4=1000/2000)
```

All input/output statements that reference TAPE1 reference INPUT instead, and all listable output normally recorded on TAPE2 is transmitted to the file named OUTPUT. TAPE4 has a buffer length of 1000 words with a maximum of 2000 characters per record.

```
PROGRAM JIM(INPUT,TAPE19=INPUT)
```

TAPE19=INPUT must be preceded in the same statement by INPUT. TAPE19 becomes the name for the file INPUT.

†Does not apply to SCOPE 2.
††See section 16 Input/Output Implementation.

```
      PROGRAM SAMPLE (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      .
      .
      .
      READ(5,100)A,B,C
  100 FORMAT (3F7.3)
```

This statement reads from logical unit 5; it is declared in the PROGRAM statement as TAPE5 which is equivalent to INPUT.

```
      .
      .
      .
      WRITE(6,200)A,B,C
  200 FORMAT (1H1,3F7.3)
```

Logical unit 6 is declared as TAPE6 in the PROGRAM statement and equivalent to OUTPUT.

## BLOCK DATA SUBPROGRAM

```
      7
      BLOCK DATA
```

```
      7
      BLOCK DATA name
```

name    identifies the BLOCK DATA subprogram if more than one is compiled.

The block data subprogram is a nonexecutable specification subprogram that can be used to enter data into labeled or numbered common (but not blank common) prior to program execution. The name BLKDAT. is assigned to the block data subprogram if it is not named by the user.

The block data subprogram contains only IMPLICIT, LEVEL, type, DIMENSION, COMMON, EQUIVALENCE, DATA, and END statements. A valid BLOCK DATA subprogram must contain at least one COMMON statement and one DATA statement. Any executable statements are ignored and a warning is issued. All DATA statements must follow the specification statements. Data can be entered into more than one block of common in a block data program. The specifications in a BLOCK DATA subprogram take effect when the binary output file (specified by the control statement B option) is loaded.

Example:

```
      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
```

Z is in block CAT and S and T are in DEF, although no initial data values are defined for them.

## PROCEDURES

The category of procedure to be used is determined by its particular capabilities and the needs of the program being written. If the program requires the evaluation of a standard mathematical function, a FORTRAN supplied intrinsic function or a basic external function can be used. If a single computation is needed repeatedly, a user-written statement function can be included in the program. If a number of statements are required to obtain a single result, a function subprogram is written. If a number of calculations are required to obtain several values, a subroutine is written.

Procedure Communication (later in this section) contains details on how to use procedures and how procedures use arguments or common to communicate.

### SUBROUTINE SUBPROGRAM

7

SUBROUTINE name $(p_1,p_2,\ldots,p_n)$

7

SUBROUTINE name

7

SUBROUTINE name $(p_1,p_2,\ldots,p_n)$, RETURNS $(b_1,b_2,\ldots,b_m)$

7

SUBROUTINE name, RETURNS $(b_1,b_2,\ldots,b_m)$

| | |
|---|---|
| name | Symbolic name of the subroutine. |
| $p_1,\ldots,p_n$ | Dummy arguments that must agree in order, number, type, and level with the actual arguments passed to the subprogram at execution time. |
| $b_1,\ldots,b_m$ | Dummy statement label arguments that must agree in order and number with the actual statement labels passed to the subroutine at execution time. |

The argument lists are optional and limited to a maximum combined total of 63 parameters.

A subroutine subprogram is executed when a CALL statement is encountered in a program unit. A subroutine subprogram must not directly or indirectly call itself. The subroutine subprogram communicates with the calling program unit through a list of arguments passed with the CALL statement or through common. Calling a Subroutine Subprogram later in this section contains more CALL statement details.

The SUBROUTINE statement contains the symbolic name that is used as the main entry point of the subprogram. (The ENTRY statement specifies an alternate entry point in the subprogram.) The subprogram name is not used to return results to the calling program, does not determine the type, and must not appear in any other statement in the same subprogram.

Subroutine subprograms can contain any statements except PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. They begin with a SUBROUTINE statement, should have at least one RETURN 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 when a RETURN, RETURN i or END is encountered.

Dummy arguments which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. If an array name without subscripts is used as an actual argument in a CALL statement and the corresponding dummy argument is not declared an array in the subprogram, the first element of the array is used in the subprogram. Adjustable dimensions are permitted in subroutine subprograms (details are given later in this section under Using Arrays).

The RETURNS list allows control to be returned to the calling program somewhere other than at the executable statement immediately following the CALL statement. The CALL statement specifies actual statement labels to replace the dummy statement label arguments in the RETURNS list. The actual statement labels must correspond in order and number with the dummy statement label arguments. The dummy statement label argument i is the statement to which control transfers when RETURN i is executed.

The RETURN statement in section 4 and the CALL statement in this section give further details.

Example 1:

**Calling Program**

```
      .
      .
      .

      IF (A-B) 10,20,20
   10 CALL ERROR1
   20 RESULT=(A*CAT) +375.2-ZERO
      .
      .
      .
```

**Subprogram**

```
      SUBROUTINE ERROR1
      WRITE (6,1)
    1 FORMAT (5X,22H NUMBER IS OUT OF RANGE)

      RETURN
      END
```

The subroutine ERROR1 is called and executed if A-B is less than zero. Control returns to statement 20. This example also illustrates that arguments need not be used.

Example 2:

```
     Calling Program              Subprogram

      .
      .                             SUBROUTINE PGM1(X,Y,Z),
      .                            XRETURNS (M,N)
      CALL PGM1(A,B,C),             U=X**Y
     XRETURNS (5,10)                X=Z+X*Y
      .                          20 IF (U+X) 25, 30, 35
      .                          25 RETURN M       Return is to statement 5 in calling program
      .                          30 RETURN N       Return is to statement 10 in calling program
    5 B=SQRT(A*C)                35 Z=Z+(X*Y)
      .                             RETURN         Return is to statement following CALL PGM1
      .                             END
      .
   10 CALL PGM2 (D,E)
      .
      .
      .
```

This example illustrates the use of the RETURNS list as well as the use of the normal RETURN statement.

## FUNCTION SUBPROGRAM

```
      7
      FUNCTION name (p1,...,pn)
```

```
      7
      type FUNCTION name (p1,...,pn)
```

name — Symbolic name of the subprogram.

$p_1,\ldots,p_n$ — Dummy arguments that should agree in order, number, type and level with the actual arguments in the calling program. At least one argument is required; a maximum of 63 is allowed.

type — The type may be REAL, INTEGER, DOUBLE, DOUBLE PRECISION, COMPLEX, or LOGICAL.

A function subprogram performs a set of calculations when its name appears in an expression in a referencing program unit. Execution of the function subprogram must result in a value being defined for the function name. A function subprogram can modify the value of one or more of its arguments or store data in common.

Dummy arguments which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. If an array name without subscripts is used as an actual argument in the function reference and the corresponding dummy argument has not been declared an array in the subprogram, the first element of the array is used in the subprogram. Adjustable dimensions are permitted in function subprograms (details are given in Using Arrays later in this section).

The FUNCTION statement contains the subprogram symbolic name that is used as the entry point when the function is referenced. The ENTRY statement specifies an alternate entry point in the function. The function name must not appear in any nonexecutable statements other than the FUNCTION statement in the subprogram. The type of the function name must be the same in the referencing program and the referenced function subprogram. When type is omitted, the type of the function result is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when no explicit type appears in the FUNCTION statement.

The function subprogram can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined. The function subprogram begins with a FUNCTION statement, should have at least one RETURN statement, and has an END statement that is treated as a RETURN if executed. Control is returned to the referencing program when either a RETURN or END is encountered. A RETURN i in a function subprogram causes a fatal error at compilation time.

A function subprogram can have the same name as that of an intrinsic or basic external function supplied by FORTRAN. Section 8 defines the conditions under which programmer supplied routines override the FORTRAN supplied routines.

Example:

**Calling Program**

```
      .
      .
      .
      DIMENSION ARY (5,5)
      .
      .
      .
   10 RES=DIAG(ARY,5)**2
      .
      .
      .
```

**Subprogram**

```
      FUNCTION DIAG (A,N)
      DIMENSION A(N,N)
      DIAG=A(1,1)
      DO 70 I=2,N
   70 DIAG=DIAG*A(I,I)
      RETURN
      END
```

The statement labeled 10 contains the reference to function DIAG. The statement labeled 70 sets the function name to a value. At the end of the function subprogram execution, RES will have the value of DIAG squared.

## BASIC EXTERNAL FUNCTION

A basic external function is a predefined procedure included with the system. Section 8 contains further details.

## INTRINSIC FUNCTION

An intrinsic function is a compiler-defined procedure that is inserted in the referencing program at compile time. Section 8 contains further details.

## STATEMENT FUNCTION

```
      7
      name (p1,p2,p3,..., pn) = expression
```

| | |
|---|---|
| name | Type of the function is determined by the type of the function name. |
| $p_1, \ldots, p_n$ | Dummy arguments must be simple variable names. At least one argument is required; a maximum of 63 is allowed. These arguments should agree in order, number, type, and level with the actual arguments used in the function reference. |
| expression | Any expression may be used. It may contain references to intrinsic or basic external functions, statement functions, or function subprograms. Names in the expression that do not represent arguments are normal variables having the same value as they have outside the function |

A statement function is a user-defined, single-statement computation and applies only to the program unit containing the definition. Since the statement function only defines the function, the value is computed when the function is referenced and the actual arguments are substituted for the dummy arguments in the definition.

During compilation, the statement function definition is retained by the compiler. Whenever the function is referenced, instructions are generated in-line to evaluate the function (as opposed to FUNCTION subprograms for which an external procedure is used at each reference). The expansion of a statement function is the same as writing the expression in place of the reference. Thus the statement function does not reduce execution speed or efficiency.

Statement function names must not appear in DIMENSION, EQUIVALENCE, COMMON or EXTERNAL statements; they can appear in a type declaration but cannot be dimensioned. Statement function names must not appear as actual or dummy arguments. If the function name is type logical, the expression must be logical. If the function name is not type logical, the expression must not be a relational or logical expression. For other types, if the function names and expression differ, conversion is performed as part of the evaluation of the function. For example, in the program segment:

```
LSUM(I,J) = OR(I,J)
A = OR(15,50)
B = LSUM(15,50)
```

OR is typeless and LSUM is a statement function of type INTEGER. In the first function evaluation, no conversion takes place; the binary value is assigned to A. In the second function evaluation, the value is converted to floating point before being assigned to B.

A statement function must precede the first executable statement and it must follow all specification statements. A statement function must not reference itself either directly or indirectly.

Examples:

| Statement Function Definitions | Statement Function References |
|---|---|
| `ADD(X,Y,C,D)=X+Y+C+D` | `RES1=GROSS-ADD(TAX,FICA,INS,RES3)` |
| `AVERGE(O,P,Q,R)=(O+P+Q+R)/4` | `GRADE=AVERGE(TEST1,TEST2,TEST3,` `TEST4)+MID` |
| `LOGICAL A,B,EQV` `EQV(A,B)=(A.AND.B).OR.` `(.NOT.A.AND..NOT.B)` | `TEST=EQV(MAX,MIN).AND.ZED` |
| `COMPLEX Z` `Z(X,Y)=(1.,0.)*EXP(X)*COS(Y)` `+(0.,1.)*EXP(X)*SIN(Y)` | `RESULT=(Z(BETZ,GAMMA(I+K))**2-1.)` `/SQRT(TWOPIE)` |

Example 1:

The statement function can be used to substitute a FORTRAN supplied function name in a program containing an alternate name for this function.

```
SINF(X)=SIN(X)       Statement function definition.
     .
     .
     .
A=SINF(3.0+B)+7.     Statement function reference.
```

The above sequence generates exactly the same object code as:

```
A=SIN(3.0+B)+7.
```

Example 2:

To compute one root of the quadratic equation $ax^2+bx+c=0$, given values of a, b and c, an arithmetic statement function can be defined as follows:

```
ROOT (A,B,C)=(-B+SQRT(B*B-4.*A*C))/(2.0*A)
```

When the function is used in an expression, actual arguments are substituted for the dummy arguments A, B, C.

```
RESA = ROOT (6.5,7.,1.)
```

is equivalent to writing:

```
RESA = (-7.+SQRT(7.*7.-4.0*6.5*1.0))/(2.0*6.5)
```

Wherever the statement function ROOT (A, B, C) is referenced, the definition of that function – in this case (-B+SQRT(B*B-4.*A*C))/(2.*A) – is evaluated using the current values of the arguments A, B, C.

# PROCEDURE COMMUNICATION

The procedures defined by a statement function or a procedure subprogram are executed when they are referenced in a program unit.

## PASSING VALUES TO A PROCEDURE

Values can be passed between a calling program unit and a procedure as actual arguments in an argument list or through common. Arrays with adjustable dimensions can be used to pass values of arguments. Arguments passed to a procedure must agree with the procedure definition in order, number, type, length, and level.

### USING ARGUMENTS

Arguments used for communication between procedures are either actual or dummy (formal). The arguments appearing in a subroutine CALL statement or a function reference are the actual arguments. The corresponding dummy arguments appear in the SUBROUTINE or FUNCTION statement. If a RETURNS list is used, the actual statement label arguments appear in the CALL statement and the dummy statement label arguments appear in the SUBROUTINE and RETURN statements.

The actual arguments allowed for a particular procedure are given in the discussion of the procedure reference.

Dummy arguments are used as variable, array or external procedure subprogram names within the subprogram and can be used to return values to the calling program. The dummy arguments are replaced by the actual arguments when the procedure is executed. Since all names are local to the program unit containing them, the same dummy argument name can be used in more than one program unit. A dummy argument must not appear in COMMON, EQUIVALENCE, or DATA statements within a program unit.

Dummy arguments representing array names must appear within the subprogram in a DIMENSION or type statement giving dimension information. If dummy arguments are not dimensioned, they cannot be referenced as arrays in a subprogram.

In a subprogram, the definition of a dummy argument that is associated with a constant actual argument or with any expression except a variable or array element is prohibited.

If a subprogram reference causes a dummy argument to be associated with an entity in common in the referenced subprogram, definition of the dummy argument or of the entity in common is prohibited. If a subprogram reference causes two dummy arguments to be associated, the definition of either in the referenced subprogram is prohibited.

Example 1:

**Calling Program**

```
        .
        .
        .
W(I,J)=FA+FB-GRATER(C-D,3*AX/BX)
        .
        .
        .
```

**Subprogram**

```
  FUNCTION GRATER(A,B)
  IF (A.GT.B)1,2
1 GRATER=A-B
  RETURN
2 GRATER=A+B
  RETURN
  END
```

This example shows the normal use of arguments in a function subprogram. The actual argument C-D is used in place of the dummy argument A and 3*AX/BX is substituted for dummy argument B when the function subprogram is executed.

Example 2:

```
CALL SUBA(1.5)
```

```
SUBROUTINE SUBA(R)
IF (R.NE.0) R = 0
```

This example contains a prohibited definition of a dummy argument, R, which is associated with a constant actual argument.

Example 3:

```
CALL SUBB  (X, X)
```

```
SUBROUTINE SUBB (A, B)
  .
  .
  .
A = Y
Z = B
```

This example contains a prohibited definition of a dummy argument, A, which has been previously associated with another dummy argument, B, in the referencing program unit.

Example 4:

```
COMMON X
CALL SUBC  (X)
.
.
.
```

```
SUBROUTINE SUBC  (B)
COMMON A
  .
  .
  .
A = Y
Z = B
```

This example contains a prohibited definition of an entity in common, A, which is associated with a dummy argument, B, in the same subprogram.

## USING COMMON

Common can be used to transfer values between a calling program unit and a subprogram. Passing values through common is more efficient than passing values through arguments in a CALL statement or function reference. If a dummy argument in a subprogram is associated with an entity in a common block in the same subprogram, the definition of either is prohibited.

Example:

```
      PROGRAM CMN (INPUT,OUTPUT)
      COMMON NED (10)
      READ 3,NED
    3 FORMAT (10I3)
      CALL JAVG
      STOP
      END
      SUBROUTINE JAVG
C THIS SUBROUTINE COMPUTES THE AVERAGE OF THE FIRST 10 ELEMENTS IN
C  COMMON
      COMMON N(10)
      ISTORE = 0
      DO 1 I = 1,10
    1 ISTORE = ISTORE + N(I)
      ISTORE = ISTORE/10
      PRINT 2,ISTORE
    2 FORMAT (*1AVERAGE = *,I10)
      RETURN
      END

      AVERAGE =          45
```

The array NED in program CMN and the array N in subroutine JAVG share the same locations in common. NED(1) shares the same location with N(1), NED(2) with N(2), etc. The values read into locations NED(1) through NED(10) are available to subroutine JAVG. JAVG computes and prints the average of these values.

## USING ARRAYS

The array dimensions in a subprogram must be the same as those in the calling routine if the subscripts are to agree between the called and calling program units. If a dummy argument is not dimensioned, it cannot be referenced as an array in the subprogram.

If any of the entries in a subscript of a type or DIMENSION statement is an integer variable name, the array is called an adjustable array. The variable names are called adjustable dimensions. Such an array can only appear in a procedure subprogram. The dummy argument list of the subprogram must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined prior to calling the subprogram and cannot be redefined during execution of the subprogram. The absolute size of the actual array may not be exceeded. For every array appearing in an executable program, there must be at least one constant array dimension associated through subprogram references.

In a subprogram, an array name that appears in a COMMON statement must have fixed dimension specifications.

## REFERENCING A FUNCTION

A function is referenced when the name appears in an expression. A function must not directly or indirectly reference itself. The reference can appear anywhere in an expression that an operand of the same type can be used.

When a statement function or intrinsic function is referenced, instructions are generated in-line to evaluate the function. The value is computed with the actual arguments substituted for the dummy arguments in the definition.

When a function subprogram or a basic external function is referenced, control is transferred to the function subprogram and the values of the actual arguments are substituted for the dummy arguments. Control is returned to the referencing program unit when a RETURN is encountered.

Actual arguments in a function subprogram reference may be an expression, constant (including Hollerith), variable, array name, array element name, subroutine subprogram name, external function name (not intrinsic function or statement function), or function reference (the function reference is a special case of an arithmetic expression).

Example:

**Calling Program**

```
Z=A+B-JOE(3.*P,Q-1)
.
.
.
R=S+JAM(Q,2.5*P)
.
.
.
```

**Function Subprogram**

```
   FUNCTION JOE(X,Y)
10 JOE=X+Y
   RETURN
   ENTRY JAM
   IF(X.GT.Y)10,20
20 JOE=X-Y
   RETURN
   END
```

Function subprogram JOE is executed as a result of its name appearing in another program unit.

## CALLING A SUBROUTINE SUBPROGRAM

```
CALL name

CALL name (p1,..., pn)

CALL name (p1,...,pn), RETURNS (b1,...,bm)

CALL name, RETURNS (b1,...,bm)
```

7

| | |
|---|---|
| name | Name of subroutine called. |
| $p_1, \ldots, p_n$ | Actual arguments which must correspond in order, number, type, and level with those specified in the SUBROUTINE statement. |
| $b_1, \ldots, b_m$ | Actual statement labels in the calling program unit that correspond in order and number with the dummy statement label arguments in the SUBROUTINE statement. This specification can be omitted if control returns to the statement immediately following the CALL statement. |

The total number of arguments of both kinds must not exceed 63.

A subroutine subprogram is executed when a CALL statement is encountered in a program unit. A subroutine must not directly or indirectly call itself. The CALL statement transfers control to the subroutine and either a RETURN or a RETURN i in the subroutine returns control to the calling program unit. If a RETURN is encountered, control is transferred to the first executable statement following the CALL statement. If RETURN i is encountered, control is transferred to the statement corresponding to i in the RETURNS list. (The RETURN statement is section 4 and Subroutine Subprogram in this section contain further details on the RETURNS list.)

The CALL statement can contain actual arguments and statement labels. They must correspond in order, number, type, and memory level to those in the subroutine subprogram definition.

The name in the CALL statement can be an alternate entry point in a subroutine subprogram, as specified in an ENTRY statement, or a subroutine name. The subroutine name must not appear in any specification statement in the calling program except an EXTERNAL statement.

Actual arguments in a subroutine subprogram call can be any of the following: expression, constant, variable, array name, array element name, subroutine subprogram name, basic external function name (not an intrinsic or statement function name), function reference (the function reference is a special case of an arithmetic expression).

Example 1:

**Calling Program**

```
      DO 5 I = 1,20
      .
      .
    5 CALL GRATER (STACK(I),TEMP(I))
      .
      .
```

**Subprogram**

```
      SUBROUTINE GRATER (A,B)
      IF (A.LE.B) GO TO 2
    1 B = A - B
      RETURN
    2 B = A + B
      RETURN
      END
```

The subroutine subprogram GRATER is called 20 times since the CALL statement as the last statement in a DO loop causes looping to continue until the DO loop terminal parameter, 20, is satisfied.

Example 2:

**Calling Program**

```
      PROGRAM MAIN(INPUT,OUTPUT)
      .
      .
      .
   10 CALL XCOMP(A,B,C),
     XRETURNS(101,102,103,104)
      .
      .
      .
  101 CONTINUE
      .
      .
      .
      GO TO 10
  102 CONTINUE
      .
      .
      .
      GO TO 10
  103 CONTINUE
      .
      .
      .
      GO TO 10
  104 CONTINUE
      END
```

**Subroutine Subprogram**

```
      SUBROUTINE XCOMP (B1,B2,G),
     XRETURNS(A1,A2,A3,A4)
      IF(B1*B2-4.159)10,20,30
   10 CONTINUE
      .
      .
      .
      RETURN A1             Return to 101
   20 CONTINUE
      .
      .
      .
      RETURN A2             Return to 102
   30 CONTINUE
      .
      .
      .
      IF (B1)40,50
   40 RETURN A3             Return to 103
   50 RETURN A4             Return to 104
      END
```

The values of A, B, and C in the CALL statement replace B1, B2, and G in the SUBROUTINE statement for use in the subprogram XCOMP. Statement numbers 101, 102, 103, and 104 replace A1, A2, A3, and A4 in the SUBROUTINE and RETURN i statements.

## USING THE ENTRY STATEMENT

```
      7
      ENTRY name
```

name is an entry point in a procedure subprogram.

The ENTRY statement defines an alternate entry point, which is other than the first executable statement, in a procedure subprogram. The ENTRY statement can appear anywhere an executable statement can appear in the subprogram except as the object of a logical IF or within the range of a DO where it is ignored and a warning diagnostic is issued. A procedure subprogram can contain any number of ENTRY statements. The first executable statement following ENTRY becomes the alternate entry point to the subprogram. ENTRY statements cannot be labeled and cause a fatal-to-execution error in a main program unit.

In the subprogram, the entry name can appear only in the ENTRY statement and each name must appear in a separate statement. A function entry name must be the same type as the name in the FUNCTION statement. The entry name must be unique within the program.

In the calling program, the reference to the entry name is made just as if reference were being made to the function subprogram or subroutine subprogram in which the entry name is contained. The name can appear in an EXTERNAL statement, and if it is a function subprogram entry name, in a type statement.

The dummy arguments, if any, appearing with the FUNCTION statement or SUBROUTINE statement do not appear with the ENTRY statement, but are assumed to be the same as for the main entry point.

In a function subprogram, the value of the function is the last value assigned to the name of the function, regardless of which ENTRY statement was used to enter the subprogram. The function name is used to return results to the calling program even though the reference was through an entry name.

Example 1:

**Calling Program** | **Subroutine Subprogram**

```
COMMON SET1 (25)           SUBROUTINE CLEAR (ARAY)
.                          DIMENSION ARAY (25)
.                          DO 100 I = 1,25 <---------- Main entry point
CALL CLEAR (SET1)      100 ARAY (I) = 0.0
.                          ENTRY FILL
.                        3 READ 2, VALUE, IPLACE <---- Alternate entry point
.                        2 FORMAT (10X, F7.2, I4)
CALL FILL (SET1)           ARAY (IPLACE) = VALUE
.                          IF (IPLACE .GT. 24) RETURN
.                          GO TO 3
.                          END
```

At some point in the calling program, a call is made to the subroutine: CALL CLEAR (SET1). The array SET1 is set to zero and values are read into the array. Later in the program, a call is made again to the subroutine CLEAR; but this time it is entered at the entry point FILL. When FILL is called, further values are read into the array SET1 without first setting the array to zero.

Example 2:

| **Calling Program** | **Subprogram** |
|---|---|
| `RESULT=FSHUN(X,Y,Z)` | `  FUNCTION FSHUN(A,B,C)` |
| `RES2=FRED(R,S,T)` | `3 FSHUN=A*B/C**2` |
| | `  RETURN` |
| | `  ENTRY FRED` |
| | `  IF(A .LE. 702.) GO TO 3` |
| | `  FSHUN=(C+A)/B` |
| | `  RETURN` |
| | `  END` |

When the FUNCTION FSHUN is entered at the beginning of the function, or through the ENTRY FRED, the result will be returned to the calling program through the function name FSHUN.

Example 3:

```
FUNCTION CAT(A,B)
.
.
.
DOG=10.+3.2
ENTRY DOG
```

The entry point name DOG is not valid because it has been used as a variable.

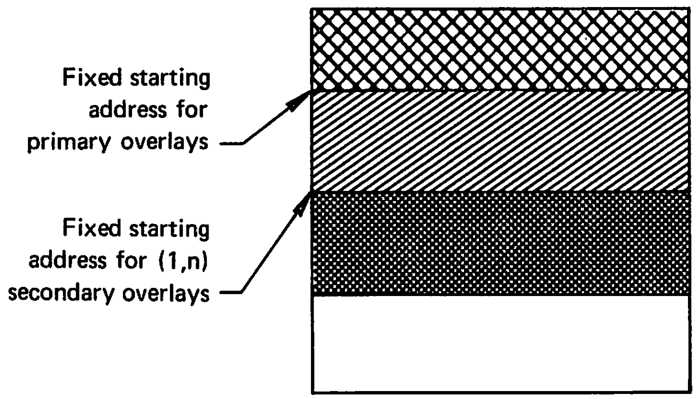
## OVERLAYS

To reduce the amount of storage required, and to make efficient use of field length, a user can divide a program into overlays. Prior to execution, the object modules of an overlay program are linked by the loader and placed on a mass storage device in their absolute form; no time is required for linking at execution time. (See Loader Reference Manual for more details.)

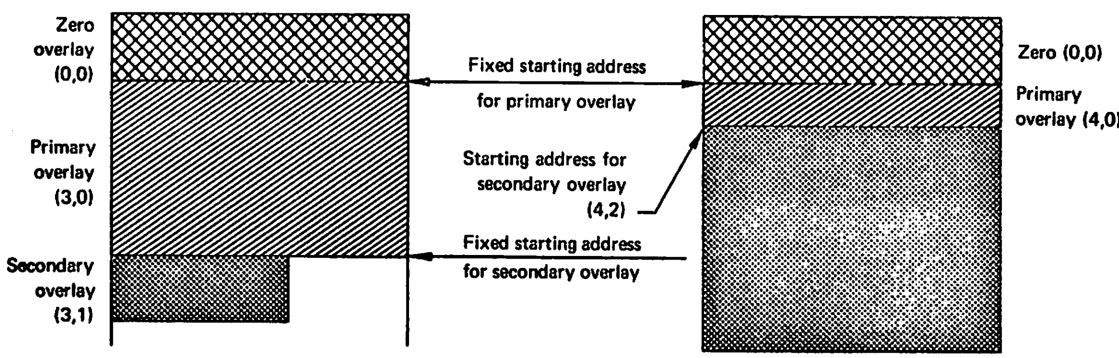
An overlay is a portion of a program written on a file in absolute form and loaded at execution time without relocation. As a result, the size of the resident loader for overlays can be reduced substantially. Overlays can be used when the organization of core can be defined prior to execution.

When each overlay is generated, the loading operation is completed by loading library and user subprograms and linking them together. The resultant overlay is in fixed format, in that internal references are fixed in their relationship to one another. The entire overlay has a fixed origin address within the field length and, therefore, is not relocatable. The overlay loader simply reads the required overlay from the overlay file and loads it starting at its pre-established origin in the user's field length.

Overlays are loaded into memory at three levels: zero, primary, and secondary.




The zero or main overlay is loaded first and remains in core at all times. A primary overlay may be loaded immediately following the zero overlay, and a secondary overlay immediately following the primary overlay. Overlays may be replaced by other overlays. For example, if a different secondary overlay is required, the overlay loader simply reads it from the overlay file and places it in memory at the same starting address as the previously loaded overlay.




When a primary overlay is loaded, the previously loaded primary overlay and any of its associated secondary overlays are destroyed. For this reason, no primary overlay may load other primary overlays. Loading a secondary overlay destroys a previously loaded secondary overlay.

Overlays are identified by a pair of integers:

zero or main overlay (0,0)

primary overlay (n,0)

secondary overlay (n,k)

n and k are positive integers in the range 1-77 octal. For any given program execution, all overlay identifiers must be unique.

For example, (1,0) (2,0) (3,0) (4,0) are primary overlays. (3,1) (3,2) (3,5) (3,7) are secondary overlays associated with primary overlay (3,0). Secondary overlays are denoted by the primary number and a non-zero secondary number. For example, (1,3) denotes that secondary overlay number 3 is related to primary overlay (1,0). (2,5) denotes secondary overlay 5 is related to primary overlay (2,0).

A secondary overlay can be called into core only by its primary overlay. Overlay (1,0) can call (1,2); but overlay (2,0) should not call (1,2), nor should overlay (0,0) call overlay (1,1).

Overlay numbers (0,n) are not valid ($n > 0$). For example, (0,3) is an illegal overlay number.

Execution is faster if the more commonly used subprograms are placed in the zero overlay, which remains in main memory at all times, and the less commonly used subprograms are placed in primary and secondary overlays which 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; it need not be the first program unit in the overlay. The program name in the PROGRAM statement becomes the primary entry point for the overlay when the overlay is called.

## OVERLAY COMMUNICATION

Data is passed between overlays through labeled or blank common. Any element of a labeled or blank common block in the main overlay (0,0) may be referenced by any higher level overlay. Any labeled or blank common declared in a primary overlay may be referenced only by the primary overlay and its associated secondary overlays – not by the zero overlay. If blank common is used for communicating between overlays, the user must ensure that sufficient field length is reserved to accommodate the largest loaded overlay in addition to blank common.

Blank common is located at the top (highest address) of the first overlay in which blank common is declared. For example, if blank common is declared in the (0,0) overlay, it is located at the top of the (0,0) overlay and is accessible to all higher level overlays. If blank common is declared in the (1,0) overlay, it is allocated at the top of the (1,0) overlay and is accessible only to the associated (1,k) overlays. Labeled common blocks are generated in the overlay in which they are first encountered; data may only be preset in labeled common blocks in this overlay.

## CREATING AN OVERLAY

An overlay is established by an OVERLAY directive preceding the program units for that overlay. An overlay consists of all program units appearing between its OVERLAY directive and the next OVERLAY directive or the end of source input. The directive must be punched starting in column 7 or later.

The PROGRAM statement for the zero or main overlay (0,0) must specify all file names such as INPUT, OUTPUT, TAPE1, etc., required for all overlay levels. File names should not appear in PROGRAM statements for other than the (0,0) overlay. The compile-time warning or informative message I/O FILE NOT DEFINED should be ignored for programs outside the (0,0) overlay.

Loading overlays from a file requires an end-around search of the file for the specified overlay; this can be time-consuming in large files. Under NOS and NOS/BE, a Fast Overlay Loading facility (FOL) is available. When FOL is enabled, an FOL directory is created in space allocated in the (0,0) overlay. When a higher level overlay is loaded, the directory is used to locate the overlay, and the overlay is loaded with a single disk access. This is the fastest method available for overlay loading and is recommended for applications where speed is essential. The FOL facility requires that all overlays in the overlay structure reside on the same file in the same order in which they were generated. FOL mode is specified by the presence of the OV parameter on the OVERLAY directive.

If the FOL facility is not used, and speed is essential, each overlay should be written on a separate file, or the overlays should be called in the same order in which they were generated.

The group of relocatable decks to be processed by the loader to create an overlay-structured program must be presented to the loader in the following order. The main overlay must be loaded first. Any primary group followed by its associated secondary group can follow, then any other primary group followed by its associated secondary group, and so forth.

The OVERLAY directive format is:

```
      7
      OVERLAY (fname,i,j,origin,OV=m)
```

fname — Name of the file on which the generated overlay is to be written.

ij — Overlay level numbers in octal without the B suffix. The numbers specified are not checked or converted by FORTRAN.

origin — Optional parameter specifying the origin of the overlay; not allowed for (0,0) overlay. The CYBER loader (NOS/BE, NOS) accepts any of the following forms; the SCOPE 2 loader allows only Cnnnnnn. If the origin parameter is omitted, the preceding comma must also be omitted.

| | |
|---|---|
| Cnnnnnn | The overlay is loaded nnnnnn words from the start of blank common. nnnnnn must be an octal number of up to six digits. |
| 0=nnnnnn | The overlay is loaded at the address specified; nnnnnn must be an octal number $\geq 110_8$. |
| 0=eptname | The overlay is loaded at the address of the entry point specified, which must have been declared in a lower level overlay. |
| 0=eptname ± nnnnnn | The overlay is loaded at the address of the entry point specified but the address is biased by the amount of the offset. |

OV=m — Optional parameter specifying the total decimal number, m, of higher level overlays in the overlay structure. Valid only on a (0,0) overlay directive and only under NOS and NOS/BE. Signals the overlay generator and overlay loader to go into Fast Overlay Loading (FOL) mode. The value of m must not exceed 20000 (decimal). If the OV parameter is omitted, the preceding comma must also be omitted.

The first overlay directive must have a file name and ij must be 0,0. Subsequent directives can omit file name indicating that the overlays are to be written on the same file. All overlays need not reside on the same file, unless in fast overlay loading mode. The second overlay directive must be of a primary overlay such as 3,0.

If the origin parameter is omitted, the overlay is loaded in the normal way directly after the zero overlay. The origin parameter cannot be included on the zero overlay directive. It is used on primary and secondary overlay directives to allow the programmer to change the size of blank common at overlay generation time.

Example:

```
OVERLAY(FNAME,0,0 ,OV=4)
PROGRAM CAT(INPUT,OUTPUT,TAPE5=INPUT)
.
.
.
OVERLAY(1,0)
PROGRAM A
.
.
.
OVERLAY(1,1)
PROGRAM B
.
.
.
OVERLAY(1,2)
PROGRAM C
.
.
.
OVERLAY(1,3)
PROGRAM D
.
.
.
```

All the above overlays are written on the file FNAME. The OV parameter signals FOL mode and indicates that there are 4 higher level overlays.

## CALLING AN OVERLAY

Overlays are called with a CALL OVERLAY statement; only the zero overlay (0,0) can be loaded when a program call control statement is encountered. The format of the CALL OVERLAY statement is:

```
      7
      CALL OVERLAY (fname,i,j,recall,k)
```

| | |
|---|---|
| fname | The name of the file or overlay in H format. |
| i,j | Overlay level numbers in octal with the B suffix, or the decimal equivalent. |
| recall | Optional recall parameter. |
| k | Optional parameter specifying where the overlay is located; can be zero, non-zero, or L format Hollerith constant. |

If the k parameter is zero or not specified, the overlay is included in the file named by fname. If a non-zero k parameter is specified, fname is the name of the overlay to be loaded. If k is an L format Hollerith constant, the overlay is loaded from the library named in the constant (not applicable under NOS 1). If k is any other non-zero value, the overlay is loaded from the global library set (refer to the appropriate operating system reference manual or the Loader Reference Manual).

If 6HRECALL is specified as the recall parameter, 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.

The three parameters fname, i, and j must be specified or the results are unpredictable.

When a RETURN or END statement is encountered in the main program of a zero overlay, execution of the program terminates and control returns to the operating system. When either of the statements is encountered in a primary or secondary overlay, control returns to the next executable statement after the CALL OVERLAY statement that invoked the current overlay.

Example 1:

```
CALL OVERLAY(1HA,1,0)
```

This statement causes a primary overlay to be loaded from the file named A.

Example 2:

```
CALL OVERLAY(3HBJR,0,0,0,1)
```

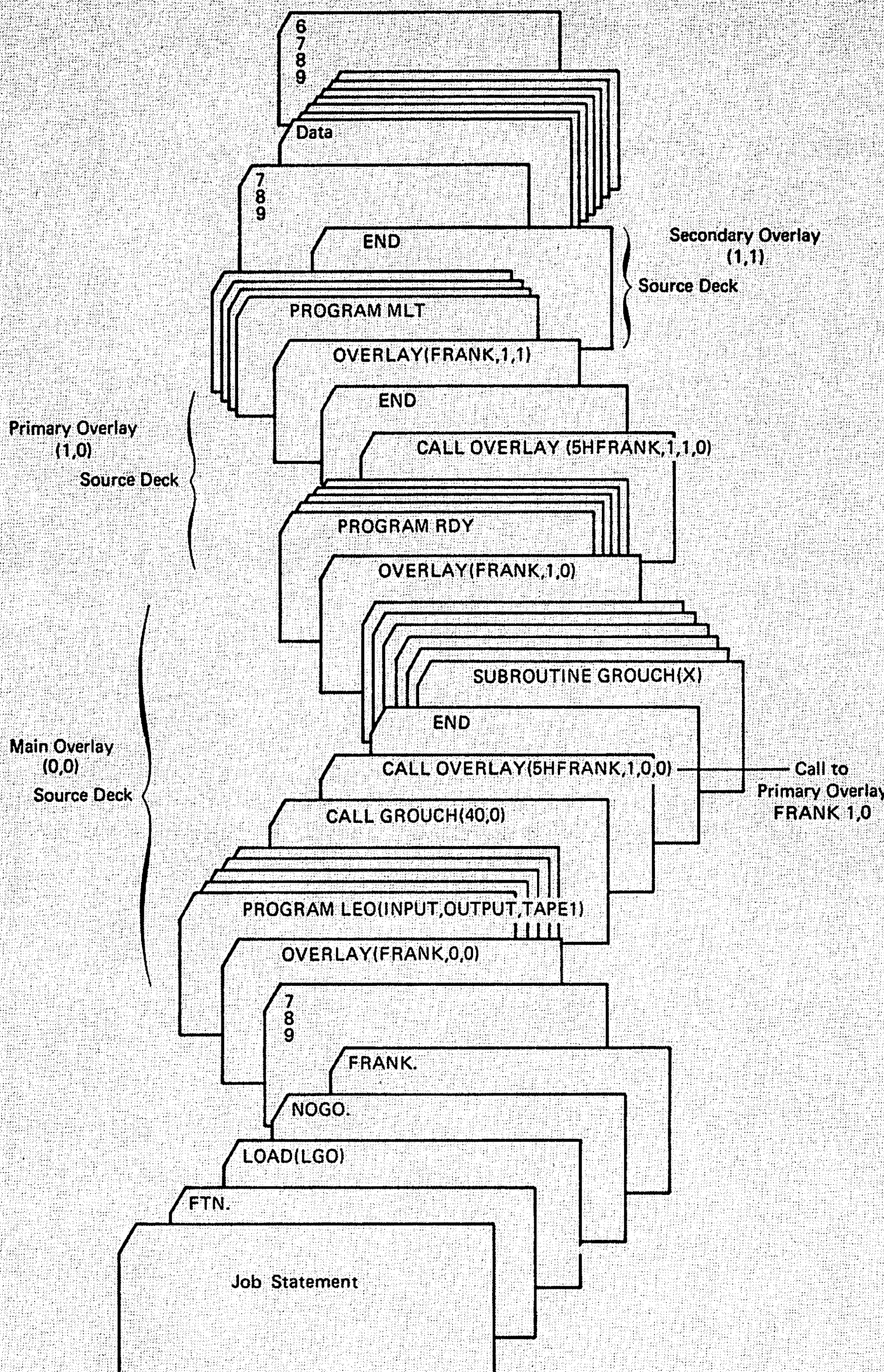
This statement, which specifies the k parameter as a non-zero value, causes a main overlay with the name BJR to be loaded from the global library set.

Example 3:

```
OVERLAY(XFILE,0,0)
PROGRAM ONE(INPUT,OUTPUT,PUNCH)
.
.
.
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
```

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.

Example:




Preparation of Overlay 0,0; 1,0; and 1,1

The above example illustrates the preparation of zero, primary and secondary overlays. The zero overlay FRANK 0,0 consists of a main program LEO and a subroutine GROUCH. The primary overlay FRANK 1,0 consists of a main program RDY. The secondary overlay FRANK 1,1 is a main program MLT. All three overlays reside on the file FRANK.

The LOAD(LGO) and NOGO control statements request the loader to load the program from the file LGO. As the loader reads file LGO, it encounters the overlay directive OVERLAY (FRANK,0,0) which instructs it to create a main overlay from the program and write it on file FRANK. When the absolute form of all the overlays has been generated, execution begins when the control statement FRANK. is encountered. FRANK. causes the main overlay to be loaded from file FRANK and executed.

During execution of the main overlay, the CALL OVERLAY (5HFRANK,1,0,0) statement is encountered and the primary overlay 1,0 is loaded into central memory. The CALL OVERLAY (5HFRANK,1,1) statement in the primary overlay causes the secondary overlay to be loaded into memory.

The primary and secondary overlays can reside on files other than FRANK. For example, the primary overlay could be on file JIM and the secondary overlay on file JOHN.

```
FTN.
LOAD(LGO)
NOGO.
FRANK.
7/8/9
OVERLAY (FRANK,0,0)
PROGRAM LEO (INPUT,OUTPUT,TAPE1)
.
.
.
CALL OVERLAY (3HJIM,1,0,0)
.
.
.
OVERLAY (JIM,1,0)
PROGRAM RDY
.
.
.
CALL OVERLAY (4HJOHN,1,1,0)
END
OVERLAY (JOHN,1,1)
PROGRAM MLT
.
.
.
END
```

Example:

The following program, for execution under NOS/BE 1 or SCOPE 2, contains several subroutines and functions and is to be used repeatedly. The entire program can be generated, therefore, as a main overlay and placed on the file in the absolute form. The control statement CATALOG creates a permanent file OVRLY where the absolute form of the program will be kept. When the program is required again, the permanent file OVRLY is called by an ATTACH control statement.

Control Cards

```
FTN.
LOAD(LGO)
NOGO.
CATALOG(REPEAT,OVRLY,ID=IBB)
7/8/9
```

Main Overlay

```
OVERLAY (REPEAT,0,0)
PROGRAM A (INPUT,OUTPUT,TAPE1)
.
.
.
END
SUBROUTINE B
.
.
.
END
FUNCTION C
.
.
.
END
SUBROUTINE D
.
.
.
END
REAL FUNCTION E
.
.
.
END
6/7/8/9
```

Main program A and the subroutines and functions B-E reside on the file REPEAT in absolute form. They can be called and executed without recompilation by the control statements:

```
job card
ATTACH(REPEAT,OVRLY,ID=IBB)
REPEAT.
7/8/9
  data
6/7/8/9
```

The operating system and Loader reference manuals give full details of the control statements which appear in the program above.

# FORTRAN EXTENDED SUPPLIED PROCEDURES 8

FORTRAN Extended provides certain procedures that are of general utility or difficult to express in FORTRAN; they are referenced in the same way as user-written procedures. The three classes of FORTRAN Extended supplied procedures are: intrinsic functions, basic external functions, and utility subprograms. Utility subprograms include operating system interface routines and subroutine packages such as the mass storage input/output routines, the CYBER Record Manager interface routines, and the Sort/Merge interface routines.

## INTRINSIC FUNCTIONS

An intrinsic function is a compiler-defined procedure that returns a single value. It is inserted in the referencing program at compile time. The form of the intrinsic function reference is the same as the statement function reference outlined in section 7.

When a variable, array, or statement function is defined with the same name as that of an intrinsic function, the user-supplied definition prevails.

When a function subprogram is defined with the same name as that of an intrinsic function, the user definition prevails only if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that changes the type associated with the intrinsic function.

Table 8-1 lists the intrinsic functions provided by FORTRAN Extended. The results of functions with the type listed as no mode assume the type of the expression in which they are used; such functions cannot be used as operands in logical expressions.

In tables 8-1 and 8-2, a single argument is denoted by A and multiple arguments by A1, A2, . . . . The symbol $0^+$ denotes a very small positive quantity; $0^-$ denotes a very small negative quantity.

## BASIC EXTERNAL FUNCTIONS

A basic external function is a predefined procedure included with the FORTRAN Common Library. These procedures are used to evaluate standard mathematical functions such as sine, cosine, square root, etc. A basic external function is referenced by the appearance of the function name with appropriate arguments in an expression.

A basic external function ordinarily is called by value; however, it is called by name if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the function, or if option T, D, or OPT=0 is specified on the FTN control statement. (Section 17 contains a description of call by value and call by name.)

When a variable, array, or statement function is defined with the same name as that of a basic external function, the user definition overrides the system definition.

TABLE 8-1. INTRINSIC FUNCTIONS

| Intrinsic Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Absolute Value | $\lvert A \rvert$ | 1 | ABS<br>IABS<br>DABS | Real<br>Integer<br>Double | Real<br>Integer<br>Double | Y=ABS(X)<br>J=IABS(I)<br>DOUBLE A,B<br>B=DABS(A) |
| Truncation | Sign of A times largest integer $\leq \lvert A \rvert$ for $\lvert A \rvert \leq 2^{48}-1$ | 1 | AINT<br>INT<br>IDINT | Real<br>Real<br>Double | Real<br>Integer<br>Integer | Y=AINT(X)<br>I=INT(X)<br>DOUBLE Z<br>J=IDINT(Z) |
| Remaindering † | A1 (mod A2) | 2 | AMOD††<br>MOD†† | Real<br>Integer | Real<br>Integer | B=AMOD(A1,A2)<br>J=MOD(I1,I2) |
| Choosing largest value | Max(A1, A2,...) | 2 - 63 | AMAX0<br>AMAX1<br>MAX0<br>MAX1<br>DMAX1 | Integer<br>Real<br>Integer<br>Real<br>Double | Real<br>Real<br>Integer<br>Integer<br>Double | X=AMAX0(I,J,K)<br>A=AMAX1(X,Y,Z)<br>L=MAX0(I,J,K,N)<br>I=MAX1(A,B)<br>DOUBLE W,X.Y,Z<br>W=DMAX1(X,Y,Z) |
| Choosing smallest value | Min(A1, A2,...) | 2 - 63 | AMIN0<br>AMIN1<br>MIN0<br>MIN1<br>DMIN1 | Integer<br>Real<br>Integer<br>Real<br>Double | Real<br>Real<br>Integer<br>Integer<br>Double | Y=AMIN0(I,J)<br>Z=AMIN1(X,Y)<br>L=MIN0(I,J)<br>J=MIN1(X,Y)<br>DOUBLE A,B,C<br>C=DMIN1(A,B) |
| Float | Conversion from integer to real | 1 | FLOAT ††† | Integer | Real | X1=FLOAT(I) |

† MOD or AMOD (a,b) is defined as a−[a/b] b, where [X] is the largest integer that does not exceed the magnitude of X with sign the same as X. The results are not defined when the second argument is zero.

†† The arguments of MOD must each be less than or equal to $2^{48}-1$; for AMOD (a,b), $\lvert a,b \rvert$ must be less than $2^{48}$.

††† The argument of FLOAT must be less than or equal to $2^{48}-1$.

TABLE 8-1. INTRINSIC FUNCTIONS (Contd)

| Intrinsic Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Fix | Conversion from real to integer. Same as INT | 1 | IFIX | Real | Integer | IY=IFIX(Y) |
| Transfer of Sign | \|A1\| with sign of A2†† | 2 | SIGN<br>ISIGN<br>DSIGN | Real<br>Integer<br>Double | Real<br>Integer<br>Double | Z=SIGN(X,Y)<br>J=ISIGN(I1,I2)<br>DOUBLE X,Y,Z<br>Z=DSIGN(X,Y) |
| Positive Difference | If A1>A2 then A1-A2. If A1 ⩽A2 then 0. | 2 | DIM<br>IDIM | Real<br>Integer | Real<br>Integer | A=DIM(C,D)<br>J=IDIM(I1,I2) |
| Logical Product | Bit-by-bit logical AND of $A_1$ through $A_n$ | 2 - 63 | AND | any type † | no mode | A=AND(X,Y,Z) |
| Logical Sum | Bit-by-bit logical OR of $A_1$ through $A_n$ | 2 - 63 | OR | any type † | no mode | A=OR(X,Y,Z) |
| Exclusive OR | Bit-by-bit Exclusive OR of $A_1$ through $A_n$ | 2 - 63 | XOR | any type † | no mode | A=XOR(X,Y,Z) |
| Complement | Bit-by-bit Boolean complement of A | 1 | COMPL | any type † | no mode | B=COMPL(A) |

† For a double precision or complex argument, only the first word is used. The result occupies one word.

†† For functions SIGN, ISIGN, and DSIGN, the sign of the second argument is defined as positive when the value of that argument is +0 and negative when the value is -0.

TABLE 8-1. INTRINSIC FUNCTIONS (Contd)

| Intrinsic Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Shift | Shift A1 by A2 bit positions: left circular if A2 is positive; right with sign extension, and end off if A2 is negative. 0≤\|A2\|≤60† | 2 | SHIFT | A1: any type †† <br> A2: integer | no mode | B=SHIFT(A,I) |
| Mask | Form mask of A1 bits set to 1 starting at the left of the word. 0≤A1≤60 † | 1 | MASK | Integer | no mode | A=MASK(I) |
| Obtain Most Significant Part of Double Precision Argument | | 1 | SNGL | Double | Real | DOUBLE Y <br> X=SNGL(Y) |
| Obtain Real Part of Complex Argument | | 1 | REAL | Complex | Real | COMPLEX A <br> B=REAL(A) |

† The results of MASK and SHIFT are undefined for arguments outside the bounds.

†† For a double precision or complex argument, only the first word is used. The result occupies one word.

TABLE 8-1. INTRINSIC FUNCTIONS (Contd)

| Intrinsic Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Obtain Imaginary Part of Complex Argument | | 1 | AIMAG | Complex | Real | COMPLEX A<br>D=AIMAG(A) |
| Express Single Precision Argument in Double Precision Form | | 1 | DBLE | Real | Double | DOUBLE Y<br>Y=DBLE(X) |
| Express Two Real Arguments In Complex Form | A1+A2i<br>(where $i^2 = -1$) | 2 | CMPLX | Real | Complex | COMPLEX C<br>C=CMPLX(A1,A2) |
| Obtain Conjugate of a Complex Argument | a-bi<br>(where A=a+bi) | 1 | CONJG | Complex | Complex | COMPLEX X,Y<br>Y=CONJG(X) |
| Random Number Generator | Returns values uniformly distributed over the range (0,1); dummy argument is ignored. | 1 | RANF | any type | Real | Y=RANF(A) |
| Obtain address of a variable, array element, or entry point of external subprogram | Argument is the name of a variable, array element, or external subprogram | 1 | LOCF | any type | Integer | J=LOCF(Q) |

TABLE 8-2. BASIC EXTERNAL FUNCTIONS

| Basic External Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Exponential | $e^A$<br>$-675.82<A<741.67$ | 1<br>1 | EXP<br>DEXP | Real<br>Double | Real<br>Double | Z=EXP(Y)<br>DOUBLE X,Y<br>Y=DEXP(X) |
| | $e^{(X+iY)}$<br>$-675.82<X\leqslant 741.67$<br>$\|Y\|\leqslant\pi \times 2^{46}$ | 1 | CEXP | Complex | Complex | COMPLEX A,B<br>B=CEXP(A) |
| Natural Logarithm | $\log_e(A)$<br>$A>0$ | 1<br>1 | ALOG<br>DLOG | Real<br>Double | Real<br>Double | Z=ALOG(Y)<br>DOUBLE X,Y<br>Y=DLOG(X) |
| | $\log_e(X+iY)$<br>$X^2+Y^2\neq 0$ | 1 | CLOG† | Complex | Complex | COMPLEX A,B<br>B=CLOG(A) |
| Common Logarithm | $\log_{10}(A)$<br>$A>0$ | 1 | ALOG10<br>DLOG10 | Real<br>Double | Real<br>Double | B=ALOG10(A)<br>DOUBLE D,E<br>E=DLOG10(D) |
| Trigonometric Sine in radians | $\sin(A)$<br>$\|A\|\leqslant\pi \times 2^{46}$ | 1<br>1 | SIN<br>DSIN | Real<br>Double | Real<br>Double | Y=SIN(X)<br>DOUBLE D,E<br>E=DSIN(D) |
| | $\sin(X+iY)$<br>$\|X\|\leqslant\pi \times 2^{46}$<br>$\|Y\|<741.67$ | 1 | CSIN | Complex | Complex | COMPLEX CC,F<br>CC=CSIN(F) |
| Trigonometric Cosine in radians | $\cos(A)$<br>$\|A\|\leqslant\pi \times 2^{46}$ | 1<br>1 | COS<br>DCOS | Real<br>Double | Real<br>Double | X=COS(Y)<br>DOUBLE D,E<br>E=DCOS(D) |
| | $\cos(X+iY)$<br>$\|X\|\leqslant\pi \times 2^{46}$<br>$\|Y\|<741.67$ | 1 | CCOS | Complex | Complex | COMPLEX CC,F<br>CC=CCOS(F) |
| Hyperbolic Tangent | $\tanh(A)$ | 1<br>1 | TANH<br>DTANH | Real<br>Double | Real<br>Double | B=TANH(A)<br>DOUBLE D,E<br>D=DTANH(E) |
| Hyperbolic Sine | $\sinh(A)$<br>$\|A\|\leqslant 742.36$ | 1<br>1 | SINH<br>DSINH | Real<br>Double | Real<br>Double | B=SINH(A)<br>DOUBLE D,E<br>D=DSINH(E) |
| Hyperbolic Cosine | $\cosh(A)$<br>$\|A\|\leqslant 742.36$ | 1<br>1 | COSH<br>DCOSH | Real<br>Double | Real<br>Double | B=COSH(A)<br>DOUBLE D,E<br>D=DCOSH(E) |

†CLOG returns values with imaginary parts in the range $(-\pi,\pi]$. For $x<0$, therefore, CLOG(x+i0) returns an imaginary part with a value $=+\pi$; CLOG(x+i0⁺) returns an imaginary part with a value $\approx+\pi$; and CLOG(x−i0⁺) returns an imaginary part with a value $\approx-\pi$.

TABLE 8-2. BASIC EXTERNAL FUNCTIONS (Contd)

| Basic External Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Error Function | $\frac{2}{\sqrt{\pi}}\int_{0}^{A} e^{-t^2}dt$ | 1 | ERF | Real | Real | Y=ERF(X) |
| Complementary Error Function | $\frac{2}{\sqrt{\pi}}\int_{A}^{\infty} e^{-t^2}dt$ A<25.923 | 1 | ERFC | Real | Real | Y=ERFC(X) |
| Hyperbolic Arctangent | arctanh(A) $\lvert A\rvert < 1$ | 1 | ATANH | Real | Real | Y=ATANH(X) |
| Trigonometric Sine in Degrees | sin(A) $\lvert A\rvert < 2^{47}$ | 1 | SIND | Real | Real | Y=SIND(X) |
| Trigonometric Cosine in Degrees | cos(A) $\lvert A\rvert < 2^{47}$ | 1 | COSD | Real | Real | Y=COSD(X) |
| Trigonometric Tangent in Degrees | tan(A) $\lvert A\rvert < 2^{47}$ | 1 | TAND† | Real | Real | Y=TAND(X) |

†The argument for TAND must not be an odd multiple of 90.

TABLE 8-2. BASIC EXTERNAL FUNCTIONS (Contd)

| Basic External Function | Definition | Number of Arguments | Symbolic Name | Type of Argument | Type of Function | Example |
|---|---|---|---|---|---|---|
| Square Root | $(A)^{1/2}$ (non-negative root) $A \geq 0$ | 1<br>1 | SQRT<br>DSQRT | Real<br>Double | Real<br>Double | Y=SQRT(X)<br>DOUBLE D,E<br>E=DSQRT(D) |
| | $A^{1/2}$ (principal value) | 1 | CSQRT† | Complex | Complex | COMPLEX CC,F<br>CC=CSQRT(F) |
| Arctangent | arctan (A) | 1<br>1 | ATAN††<br>DATAN†† | Real<br>Double | Real<br>Double | Y=ATAN(X)<br>DOUBLE D,E<br>E=DATAN(D) |
| | arctan (A1/A2) $A1^2+A2^2 \neq 0$ | 2<br>2 | ATAN2†††<br>DATAN2††† | Real<br>Double | Real<br>Double | B=ATAN2(A1,A2)<br>DOUBLE D,D1,D2<br>D=DATAN2(D1,D2) |
| Remaindering | A1 (mod A2) | 2 | DMOD ξ | Double | Double | DOUBLE DM,D1,D2<br>DM=DMOD(D1,D2) |
| Modulus | $\sqrt{a^2+b^2}$ A=a+bi | 1 | CABS | Complex | Real | COMPLEX C<br>CM=CABS(C) |
| Arccosine | arccos (A) $\|A\| \leq 1$ | 1 | ACOSξξ<br>DACOSξξ | Real<br>Double | Real<br>Double | X=ACOS(Y)<br>DOUBLE D,E<br>E=DACOS(D) |
| Arcsine | arcsin (A) $\|A\| \leq 1$ | 1 | ASINξξξ<br>DASINξξξ | Real<br>Double | Real<br>Double | X = ASIN(Y)<br>DOUBLE D,E<br>E = DASIN(D) |
| Trigonometric Tangent in radians | tan (A) $\|A\| \leq 2^{47}$ | 1 | TAN<br>DTAN | Real<br>Double | Real<br>Double | X=TAN(Y)<br>DOUBLE D,E<br>E = DTAN(D) |

†CSQRT returns values in the right half plane.

††ATAN and DATAN return values in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$.

†††ATAN2 and DATAN2 return values in the range $(-\pi,\pi]$. For x < 0, therefore, ATAN2(0,x) returns a value = $+\pi$; ATAN2($0^+$,x) returns a value ≈ $+\pi$; and ATAN2($0^-$,x) returns a value ≈ $-\pi$.

ξ The function DMOD (a,b) is defined as a-[a/b]b,where[X] is the largest integer that does not exceed the magnitude of X with sign the same as X; the result is not defined when the second argument is zero.

ξξ ACOS and DACOS return values in the range $[0,\pi]$. ξξξ ASIN and DASIN return values in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

When a function subprogram is defined with the same name as that of a basic external function, the user definition overrides the library definition only if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the library function, or if option T, D, or OPT=0 is specified on the FTN control statement.

Table 8-2 lists the basic external functions.

Arguments for which a result is not mathematically defined, or those of a type other than that specified, should not be used. Arguments of the trigonometric functions SIN, COS, and TAN are in radians; those of SIND, COSD, and TAND are in degrees. The inverse trigonometric functions return principal values in radians.

If the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the library function, or if option T, D, or OPT=0 is specified on the FTN control statement, the arguments of all external functions are checked to ensure that they are neither indefinite nor infinite and fall within the limits listed in the Definition column of table 8-1. Argument checking is provided unconditionally for all single and double precision math functions except DSIN, DCOS, DLOG, and DLOG10. An informative diagnostic is provided when an argument is found to be invalid.

## MISCELLANEOUS UTILITY SUBPROGRAMS

The utility subprograms described below are supplied by the system and are always called by name (section 17 defines call by name). A user-supplied subprogram with the same name as a library subprogram overrides the library subprogram. Other utility routines, such as the mass storage routines, CYBER Record Manager interface routines, Sort/Merge interface routines and Post Mortem Dump routines are described later in this section.

In the definitions listed under the routines:

i and n are integer variables, constants, or expressions.

j is an integer variable.

a and b are variable or array names of any type.

u is a unit designator (as defined in section 5).

H is a Hollerith specification.

### RANDOM NUMBER GENERATOR

#### RANF (n)†

Random number generator. Returns values uniformly distributed over the range (0,1); the value 0 and 1 are excluded. n is a dummy argument which is ignored. Result is type real.

†RANF is an intrinsic function.

**CALL RANSET(n)**

Initializes seed of RANF. n is a one-word bit pattern. Bit 0 will be set to 1 (forced odd), and bits 59 through 48 will be set to 1717 octal.

**CALL RANGET(n)**

Obtains current seed of RANF between 0 and 1. n is a symbolic name to receive the seed. It is not necessarily normalized. The value returned may be passed to RANSET at a later time to regenerate the same sequence of random numbers.

## OPERATING SYSTEM INTERFACE ROUTINES

**DATE(a) or CALL DATE(a)†**

The current date is returned as the value of argument a or of the function in the form 10Hƀmm/dd/yyƀ (under NOS/BE 1, SCOPE 2) or 10Hƀyy/mm/dd. (under NOS 1), where ƀ denotes a blank, mm is the number of the month, dd is the number of the day within the month, and yy is the year. The value returned is Hollerith data and can be output using an A format specification.††

The default type of the function DATE is real; thus if J and K are integer variables as in:

```
J = DATE(K)
```

J will not be useful because the value returned will have been converted from real to integer.

**JDATE(a) or CALL JDATE(a)† ‡**

The current date is returned as the value of argument a or of the function in the form 5Ryyddd, where yy is the year and ddd is the number of the day within the year. The value returned is Hollerith data and can be output using an R format specification. The type of the function JDATE is integer.

**SECOND(t) or CALL SECOND(t)†**

The central processor time is returned from start-of-job in seconds as a real number, usually accurate to two decimal places. t is a real variable.

Example:

```
DPTIM = SECOND (CP)
```

†These routines can be used as functions or subroutines. The value is returned via the argument and the normal function return.

‡ Not available under SCOPE 2.

††The date format can be changed by the installation.

**TIME(a) or CALL TIME(a)†**

**CLOCK(a) or CALL CLOCK(a)†**

The current reading of the system clock is returned as the value of argument a or of the function in the form 10Hƀhh.mm.ss., where ƀ denotes a blank, and hh, mm, and ss are the number of hours, minutes, and seconds, respectively. The value returned is Hollerith data and can be output using an A format specification.

The default type of the functions TIME and CLOCK is real; thus if J and K are integer variables in the following statement, J is not useful because the value returned will have been converted from real to integer.

Example:

```
J = TIME(K)
```

**CALL DISPLA (H,k)**

A name and a value are placed in the dayfile. H is a Hollerith specification of not 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 code greater than 57 octal are replaced by blanks when displayed at the operator's console. If the first character is $, the message will flash at the console except under NOS 1, which allows flashing messages only for system origin jobs.

Example:

```
CALL DISPLA (7H TIME = , STOP-START)
```

**CALL REMARK (H)**

Places a message in the dayfile. Under SCOPE 2, the maximum message length is 90 characters displayed on one line. Under NOS/BE 1, the maximum message length is 80 characters displayed 40 characters per line. Under NOS 1, the message length is one line of 30 characters. A message exceeding the maximum length is truncated. A message shorter than the maximum must have all zeros in the lower 12 bits of the last word. These zeros are automatically supplied when a Hollerith constant is used as the parameter. Characters with display code greater than 57 octal are listed in the dayfile, but they are replaced by blanks when displayed at the operator's console. If the first character is $, the message will flash at the console, except under NOS 1, which allows flashing messages only for system origin jobs.

Example:

```
CALL REMARK (9HLAST DECK)
```

**CALL SLITE(i)**

Sense light i is turned on. If i = 0, all sense lights are turned off. If i is other than 0 through 6, an informative diagnostic is printed and sense lights are not changed.

**CALL SLITET(i,j)**

Sense light i is tested. If sense light i is on, j = 1; if sense light i is off, j = 2. If i is other than 1-6, an informative diagnostic is printed, all sense lights remain unchanged, and j = 2. Execution turns off sense light i if it is on.

---

†These routines can be used as functions or subroutines. The value is returned via the argument and the normal function return.

(Note: Logical variables generally provide a more efficient method of testing a condition than do calls to SLITE or SLITET.)

**CALL SSWTCH(i,j)**

If sense switch i is on, j is set to 1; if sense switch i is off, j is set to 2. 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 and NOS/BE), ONSW (NOS only), and OFFSW (NOS only).

**CALL OVERLAY(fname, primary, secondary, recall,k)**

See section 7.

**CALL EXIT**

Program execution is terminated and control is returned to the operating system. (Note: use of the STOP statement is preferable to CALL EXIT.)

**CALL CHEKPTX(filelist,SP)**

A checkpoint dump of the files specified is taken. The arguments are:

filelist Array in the following format:

| | 59 | 17 | 11 0 |
|---|---|---|---|
| Word 1 | | n | 0000 |
| Word 2 | $lfn_1$ | $f_1$‡ | |
| Word 3 | $lfn_2$ | $f_2$‡ | |
| | | | |
| Word n + 1 | $lfn_n$ | $f_n$‡ | |

‡Does not apply to SCOPE 2.

SP — Flag indicating whether or not all files assigned to the job are to be checkpointed. If SP is zero, all local files are checkpointed. If SP is nonzero, the files specified by filelist are checkpointed.

n — Number of files in following list, to a maximum of 42.

$lfn_i$ — Name (in left-justified display code) of user mass storage files to be processed.

$f_n$[‡] — Number indicating specific manner in which lfn is to be processed.

- 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 checkpoint 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.

Example:

```
      .
      .
      .
      DIMENSION IFILES(4)
      IFILES(1) = 30000B
      IFILES(2) = 5LTAPE1 .OR. 10000B
      IFILES(3) = 5LTAPE2 .OR. 30000B
      IFILES(4) = 5LTAPE3
      .
      .
      .
      CALL CHEKPTX(IFILES,1)
```

The names defined in the array passed to CHEKPTX must be the actual file names used at run time.

For more information, refer to the operating system reference manual checkpoint/restart discussions.

**CALL RECOVR(name,flags,checksum,addr1,addr2)**[‡]

name — Name of subroutine to be executed if flagged conditions occur (must be specified in an EXTERNAL statement).

flags — 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 377 allowed.

- 001 — Arithmetic mode error.
- 002 — PP call or auto-recall error.

[‡]Not available under SCOPE 2.

| | |
|---|---|
| 004 | Time or storage limit exceeded. |
| 010 | Operator drop, kill, or rerun. |
| 020 | System abort. |
| 040 | CP abort. |
| 100 | Normal termination. |
| 200 | Terminal interrupt. |

checksum
: Last word address of recovery code to be checksummed; 0 if no checksum is desired; a negative number if the address parameters follow, specifying the area where checksumming is desired.

addr1
: First word address of recovery code to be checksummed (used only when checksum is a negative number).

addr2
: Last word address of recovery code to be checksummed (used only when checksum is a negative number).

The RECOVR subroutine allows a user program to gain control at the time that normal or abnormal job termination procedures would otherwise occur. 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.

No more than ten 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 26-word integer array. The first 16 words are an image of the exchange package; the seventeenth word is the contents of RA+1. The first word of the exchange package contains the value of B0; bits 0 through 17 of B0 contain the error flag. The 18th word contains a zero value unless it is changed by user's recovery subroutine (discussed later in this section). The 19th through 26th words contain a copy of the 1st through 9th words of the reprieve parameter block. Refer to the appropriate operating system reference manual for detailed information.

2. A flag that, upon return, determines the type of program termination. This flag is initially set to zero. The possible types of program termination are discussed later in this section.

3. An array, starting at RA+1, that allows a FORTRAN 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 or REWIND statement.

Upon return from a user's recovery subroutine, the 18th word of the reprieve package and the argument flag contain values set by the user's recovery subroutine. If the 18th word of the reprieve package is set to 4, the program immediately resumes normal execution, and the argument flag is ignored. If the 18th word is not set to 4, the argument flag, which is set by the user's recovery subroutine, is checked. If the flag is set to nonzero, the job terminates normally, as if no errors had occurred. This happens after all the recovery subroutines have been processed. If the flag remains zero for all the recovery subroutines, the job continues as if RECOVR had not been called; that is, the original system error code is reset and processed.

If the flags are not set by any of the recovery subroutines, the job continues as if RECOVR had not been called; that is, the original system error code is reset and processed.

In an overlay structured program, calls to RECOVR as well as the user recovery subprograms should be in the (0,0) overlay.

For further information about RECOVR, refer to the appropriate operating system reference manual.

Example:

```
      PROGRAM MAIN(INPUT,OUTPUT)
      EXTERNAL REPREV,CHKSUM
      .
      .
      .
      CALL RECOVR(REPREV,72B,LOCF(CHKSUM))
      .
      .
      .
      STOP
      END
      SUBROUTINE REPREV(IXCHNG,IFLAG,IFLDLN)
      DIMENSION IXCHNG(17), IFLDLN(40000B)
      IFLAG = 1
      PRINT 10, IXCHNG, (IFLDLN(I), I=1,64)
10    FORMAT (3(6X, O20))
      RETURN
      ENTRY CHKSUM <-------------------- determines end of code to be checksummed
      END
```

## DEBUGGING AIDS

A number of calls and functions useful in debugging are described here. Many users find CYBER Interactive Debug and/or Post Mortem Dump more useful. They are described near the end of this section.

**CALL DUMP ($a_1,b_1,f_1, \ldots, a_n,b_n,f_n$)**

**CALL PDUMP ($a_1,b_1,f_1, \ldots, a_n,b_n,f_n$)**

Dumps central memory on the OUTPUT file in the indicated format. PDUMP returns control to the calling program; DUMP terminates program execution. $a_i$ and $b_i$ specify the beginning and the end of the storage area to be dumped. $1 \leqslant n \leqslant 20$. f is a format indicator, as follows:

| | |
|---|---|
| f = 0 or 3 | octal dump |
| f = 1 | real dump |
| f = 2 | integer dump |

For f values 0 through 3, $a_i$ and $b_i$ are the first and last words dumped. If 4 is added to any f value, the values of $a_i$ and $b_i$ are used as the addresses of the first and last words dumped within the job's field length. An ASSIGN statement or the LOCF function can be used to get addresses for the $a_i$ and $b_i$ parameters.

Examples:

| | |
|---|---|
| CALL PDUMP(A(1),A(100), 1) | Dumps from A(1) to A(100) as real numbers |
| CALL PDUMP (0, 1000B, 4) | Dumps from location 0 to 1000B in octal |

**CALL STRACE**

Provides traceback information from the subroutine calling STRACE back to the main program. Traceback information is written to the file DEBUG. To obtain traceback information interspersed with the source program, DEBUG should be equivalenced to OUTPUT in the PROGRAM statement. (Refer to STRACE, section 9).

**LEGVAR(a)**

Checks the value of variable a. Returns the result -1 if variable is indefinite, +1 if out of range, and 0 otherwise. Variable a is type real; result is type integer.

**CALL SYSTEM(errnum,mesg)**

errnum — Error number. An integer value from 0 to 9999 decimal. Error numbers used by the compiler (listed in appendix B) retain the severity associated with them. Error numbers 51 (non-fatal) 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. If errnum is negative, SYSTEM returns immediately with no action taken.

mesg — Error message; entered as a Hollerith constant with the first character used as a carriage control character and not printed.

The subroutine SYSTEM enables the user to issue an execution-time error message.

If error number zero is entered, the message is ignored, the output buffers are flushed, and control is returned to the calling program.

The file OUTPUT should be declared before SYSTEM is called. Otherwise, no errors are printed; and a message to this effect is entered in the dayfile.

Each line is printed unless the line limit of the OUTPUT buffer is exceeded, in which case the job is terminated.

Example:

```
CALL SYSTEM (3, ≠ CHECK DATA ≠ )
```

**CALL SYSTEMC (errnum,speclist)**

errnum — Error number for which non-standard recovery is to be implemented. Error numbers are listed in table B-5 of appendix B.

speclist — Integer array containing error processing specifications in consecutive locations:

| | |
|---|---|
| word 1 | F/NF (1 = fatal, 0 = non-fatal) |
| 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 all errors; this limit is 20 unless changed by a call to SYSTEMC |

SYSTEMC enables the user to replace the standard fatal/non-fatal error table with an enhanced error table, which contains specifications that regulate error processing. The error table, whose format is described below, is ignored for erroneous data input from a connected (terminal) file.

In an overlay program, if SYSTEMC is not called in the (0,0) overlay, the routine might not be available to higher level overlays.

In the error table, the first entry corresponds to error number 1, the second to error number 2, and so on. Each entry has the following format:

| 59 | 51 | 43 | 31 | 20 | | 17 … 0 |
|---|---|---|---|---|---|---|
| print frequency | frequency increment | print limit | detection total | F/NF | A/NA | user-specified recovery address |

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, non-fatal. 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 non-standard 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. |

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.

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 FORTRAN function detecting the error. For any other error, a user-supplied error recovery should be a subroutine subprogram.

The error recovery subprogram must not invoke any code during recovery that was also involved with the error being handled. In particular, if an input/output related error is being processed, no input/output operations of any type should be done, unless the recovery routine intends to terminate execution.

When SYSTEMC is called from an overlay or segment, it must reside in the (0,0) overlay or the root segment.

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 non-fatal, 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 non-standard 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 error for errors detected by a math library routine.<br><br>Address of the FIT for error 103.<br><br>Undefined for all other errors. |
| X1 | Address of the first argument in the list for errors detected by a math library routine.<br><br>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:<br><br>Error number associated with this error.<br><br>Address of message associated with this error.<br><br>Address within auxiliary table if A/NA bit set; otherwise 0.<br><br>In upper 30 bits, instruction consisting of RJ to SYSERR.j; in lower 30 bits, address of traceback information for routine detecting the error.<br><br>Information in the secondary argument list is not available to user supplied error recovery routines coded in FORTRAN. |
| A2 | Address of error table entry for this error. |
| X2 | Contents of error table entry for this error. |

**Example 1:**

```
      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
```

```
ARGUMENT TOO LARGE, FLOATING OVERFLOW
ERROR NUMBER      30      DETECTED BY EXP

ARGUMENT TOO SMALL
ERROR NUMBER     115   DETECTED BY EXP

*****SYSTEMC IS CALLED TO SUPPRESS PRINTING OF ERROR 115

ARGUMENT TOO LARGE, FLOATING OVERFLOW
ERROR NUMBER      30    DETECTED BY EXP

*****ERROR 115 DETECTED BUT NOT PRINTED

ERROR SUMMARY
     ERROR          TIMES
     0030           0001
     0115           0001
```

Program EXPECT illustrates a standard error recovery in a math library routine and how to suppress the printing of error message 115.

Example 2:

```
      PROGRAM FXAMPL(TAPE1,OUTPUT)
      EXTERNAL ITSOK
      DIMENSION NARRAY(6)
      DATA NARRAY/6*(-1)/
      NARRAY(1) = 0
      NARRAY(5) = LOCF(ITSOK)
      NARRAY(6) = 1
      CALL SYSTEMC(66,NARRAY)
      NAMELIST/DATA1/A,B
      READ (1, DATA1)
      REWIND 1
      NAMELIST/DATA2/A,B
      READ (1, DATA2)
      NAMELIST/DATAOUT/A,B
      PRINT DATAOUT
      STOP
      END
      SUBROUTINE ITSOK
      PRINT 10
   10 FORMAT (*0DATA SET NAMED ABOVE NOT USED*)
      RETURN
      END
```

Input:

```
 $DATA2
 A = 3.,
 B = 4.,
 $
```

Output:

```
NAMELIST NAME NOT FOUND -     DATA1
ERROR NUMBER   0066 DETECTED BY NAMIN=  AT ADDRESS 000435

DATA SET NAMED ABOVE NOT USED


$DATAOUT

A         = .3E+01,

B         = .4E+01,

$END

ERROR SUMMARY
    ERROR     TIMES
     0066     0001
```

**CALL LIMERR(lim)**

lim — Integer value; the program does not terminate when data errors are encountered until the number of errors occurring after the call exceeds the value of lim.

NUMERR(n) — A function that returns the number of errors since the last LIMERR call. Result type is integer. n is a dummy argument which is ignored.

The subroutine LIMERR and function NUMERR enable the user to input data without the risk of termination when improper data is encountered.

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.

Example:

The following example illustrates the use of LIMERR and NUMERR to suppress normal fatal termination when large sets of data are being processed.

```
          .
          .
          .
      CALL LIMERR (200)
      READ(1,125)(ARAY(I),I=1,1500)
  125 FORMAT (3F10.5,E10.1)
      IF (NUMERR(0).GT.0) GO TO 500
          .
          .
          .
  500 CALL LIMERR(200)
      READ(1,125)(BRAY(I),I=1,1500)
      IF (NUMERR(0).GT.0) GO TO 600
          .
          .
          .
  600 CALL LIMERR(100)
      READ(1,230)(LRAY(I),I=1,500)
      PRINT 99, NUMERR(0)
      READ(4,127)(MRAY(I),I=1,500)
      PRINT 99, NUMERR(0)
      READ(4,225)(NRAY(I),I=1,50)
          .
          .
          .
      IF (NUMERR(0).GT.0) GO TO 700
          .
          .
          .
  700 STOP
      END
```

When LIMERR is called, a limit of 200 errors is established. The number of errors is reset to zero. After ARAY is read, NUMERR(0) 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.

Example:

```
      PROGRAM EXAMPL(TAPE1,OUTPUT)
      DIMENSION ACARD(5)
      DATA ACARD /-1.,-2.,-3.,-4.,-5./
      CALL LIMERR(2)
      READ(1,10) (ACARD(I),I=1,5)
   10 FORMAT (F4.1)
      PRINT 20, NUMERR(0)
   20 FORMAT (1H0, I1, * DATA ERRORS FOUND*//)
      PRINT 30, (ACARD(I),I=1,5)
   30 FORMAT (1X, F4.1)
      STOP
      END
```

Input:

```
47.1
25./
48.3
24,6
91.2
```

Output:

```
 2        25./

          ...↑......1234567890123456789012345678901234567890

* ERROR DATA INPUT * ILLEGAL DATA IN FIELD *↑*
ERROR NUMBER    78    DETECTED BY INCOM=   AT ADDRESS 000215
CALLED FROM KRAKER= AT ADDRESS 000345
CALLED FROM INPC=   AT ADDRESS 000075
CALLED FROM EXAMPL  AT  LINE 5

 4        24,6

          ..↑.......1234567890123456789012345678901234567890

* ERROR DATA INPUT * ILLEGAL DATA IN FIELD *↑*
ERROR NUMBER    78    DETECTED BY INCOM=   AT ADDRESS 000215
CALLED FROM KRAKER= AT ADDRESS 000345
CALLED FROM INPC=   AT ADDRESS 000075
CALLED FROM EXAMPL  AT  LINE 5

2 DATA ERRORS FOUND


47.1
-2.0
48.3
-4.0
91.2
```

## INPUT/OUTPUT STATUS CHECKING

FORTRAN Extended provides the capability of checking for an end-of-file or a parity error condition following read operations via the functions UNIT, EOF, and IOCHEC.

Any of the following conditions encountered during a read returns an end-of-file status via the functions UNIT or EOF:

- End of section (in the case of file INPUT only)
- End of partition
- End of information
- Non-deleted 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; however, such an indication does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the Record Manager input/output routines.

§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 (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(u,a,b)**

The UNIT function 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. Parameters a and b are the first and last variable or array elements, respectively, of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement. 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 |

Example:

```
BUFFER IN(5,1)(B(1),B(100))

IF (UNIT(5,B(1),B(100))) 12,14,16
```

§Applies only to SCOPE 2.

Control transfers to the statement labeled 12, 14 or 16 if the value returned was -1., 0., or +1., respectively.

Parameters a and b should always be included. If these parameters are omitted (as in most older programs), there is a slight chance that at OPT=2, execution errors will result or that the elements between a and b will not contain the expected values.

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.

**EOF(u)**

The EOF function is used to test for an end-of-file condition on unit u following a formatted, list-directed, NAMELIST, or unformatted read. Zero is returned if no end-of-file is encountered, or a non-zero value if end-of-file is encountered.

Example:

```
IF (EOF(5)) 10,20
```

returns control to the statement labeled 10 if the previous read encountered an end-of-file; otherwise, control goes to statement 20.

If an end-of-file is encountered, EOF clears the indicator before returning control.

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 user should test for an end-of-file after each READ statement to avoid input errors. If an attempt is made to read on unit u and an EOF was encountered on the previous read operation on file, execution terminates and an error message is printed.

**IOCHEC(u)**

The IOCHEC function tests for 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.

Example:

```
J = IOCHEC(6)

IF (J) 15,25
```

Zero value would be returned to J if no parity error occurred and non-zero if an error had occurred; control would transfer to the statement labeled 25 or 15 respectively.

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.

## OTHER INPUT/OUTPUT SUBPROGRAMS

**LENGTH(u) or CALL LENGTHX(u,nw,ubc)**

Returns information regarding the previous BUFFER IN or READMS call of the file designated by u. nw or the value of LENGTH is set to the number of 60-bit words read. ubc is set to the number of unused bits in the last word of the transfer. nw, ubc, and value returned are type integer.

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 record of 23 8-bit frames 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.

Example:

```
NW = LENGTH(5)

or

CALL LENGTHX(5,NW,NUBC)
```

**CALL LABEL(u,labinfo)** ‡

u — Logical unit number.

labinfo — Name of 4-word array containing label information in the format shown in table 8-3.

This subroutine passes label information to the operating system.

The control statement that requests the tape for the job must have specified that the tape has labels before the CALL LABEL statement can be used.

CALL LABEL should not be used with files accessed with CYBER Record Manager interface routines.

‡Recognized but ignored under SCOPE 2.

TABLE 8-3. LABINFO BLOCK CONTENT

| Word | Bit Positions | Character Length | Contents | Default for Output |
|---|---|---|---|---|
| 1 | 59 thru 0 | 10 | File identifier | None. Entire field must be supplied by the user. |
| 2 | 59 thru 18 | 7 | File identifier continued | None. Entire field must be supplied by the user. |
| | 17 thru 0 | 3 | File sequence number | 001† |
| 3 | 59 thru 48 | 2 | Generation version number | 00† |
| | 47 thru 30 | 3 | Retention cycle | 000† |
| | 29 thru 0 | 5 | Creation date (YYDDD) | Today's Julian date. |
| 4 | 59 thru 24 | 6 | Set identifier (VSN) | Blanks (field can be blank or zero filled. |
| | 23 thru 0 | 4 | File section number | 0001† |

†Field must be display code zero filled.

Default output values are specified for a field by filling it with binary zeros. If any of the fields are improperly initialized, NOS generates an LX BUFFER/FET PARAMETER ERROR and aborts the job on output. For further information concerning the fields, consult the appropriate operating system reference manuals concerning words 9-12 of the file environment table (FET).

The actions that occur at the time of the LABEL call are as follows:

Passing the address and length of the user label block to CYBER Record Manager

Forcing a rewind on any succeeding open of the logical unit

The LABEL reference occurs before any access is made to the logical unit. If a WRITE subsequently occurs on the logical unit, the Record Manager routines write a label formatted from the LABINFO area onto the tape as an HDR1 label. Or, when a close or endfile operation is performed at job termination, the routines write an EOF1 label formatted from LABINFO. If a READ subsequently occurs on the logical unit, and a label is encountered on the opening read or an EOF condition, the label information is reformatted and placed into LABINFO.

## ECS/LCM/UEM SUBPROGRAMS

### CALL MOVLEV (a,b,n)

Transfers n consecutive words of data between a and b. a and b are variables or array elements; n is an integer value. a is the starting address of the data to be moved and b is the starting address of the receiving location.

Example:

```
CALL MOVLEV(A,B,1000)
```

No conversion is done by MOVLEV. If data from a real variable is moved to an integer type 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.

### CALL READEC(a,b,n)

Transfers data from extended core storage to central memory.

a is a simple variable or array element located in central memory. b is a simple variable or array element located in an extended core storage block or LCM block. n is an integer constant or expression. n consecutive words of data are transferred beginning with a in central memory and b in extended core storage.

**CALL WRITEC(a,b,n)**

Transfers data from central memory to extended core storage or LCM.

No type conversion is done.

Example:

```
LEVEL 3,B

CALL READEC(A,B,10)

CALL WRITEC(A,B,10)
```

## TERMINAL INTERFACE SUBPROGRAMS‡

**CALL CONNEC (u,cs)**

u unit designator.

cs optional character set designator (applicable to NOS/BE only): cs is an integer with a value from 0 to 2, in accordance with the character set to be used for the data entered or displayed at the terminal:

0 display code (default)
1 ASCII-128
2 ASCII-256 code

If a FORTRAN program to be run under INTERCOM for NOS/BE, under the NOS Time-Sharing System, under the NOS Interactive Facility, or under HELLO7 for SCOPE 2, 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.

In particular, the file INPUT must be connected to the terminal if data is to be entered there and an alternate 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 an alternate unit is not designated in the WRITE or PRINT statement. These files are automatically connected to the terminal when the program is executed under the NOS FTNTS subsystem or under the RUN command of the EDITOR utility of INTERCOM.

Under HELLO7, any file can be connected by providing a FILE control statement specifying CNF = YES.

Under INTERCOM, any file can be connected to the terminal by the CONNECT command.

Under all operating systems, the user can connect any file from within the program by using the CALL CONNEC statement.

A file n is considered still connected if a CALL CONNEC(n) has been made by a program running at a terminal and if the program terminates under normal or abnormal circumstances without a CALL DISCON(n). Any subsequent input/output on n will still be through the terminal unless the file is returned.

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 pre-existing file is not reassociated with the file name.

‡More information about INTERCOM is in the INTERCOM reference manual and the INTERCOM Interactive Guide for Users of FORTRAN Extended. More information about NOS is in the Interactive Facility reference manual. More information about HELLO7 is in the SCOPE 2 reference manual.

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 buffer input/output is used, either a FILE control statement (section 16) specifying RT=S must be provided, or blanks cannot terminate a line.

When a CALL 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 64-character 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 FORTRAN program that data represented in an ASCII 128-character set (providing the capability for recognizing lowercase letters and control codes) 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 128-character and 256-character sets, characters are stored in five 12-bit bytes in each central memory word. Characters in the ASCII 128-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. 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 (see section 7). Allowance should also be made in input/output operations for the fact that internal characters require twice as much space as external characters.

If ASCII information needs to be read from or sent to a terminal a formatted READ or WRITE statement should be used. This will cause trailing blanks to be truncated from the line.

If the exact length of the data input at the terminal is required, BUFFER IN and BUFFER OUT should be used. LENGTHX can be used after a BUFFER IN to determine how many characters were entered. BUFFER IN and BUFFER OUT ensure that trailing blanks are not added or deleted from the line. The file used for BUFFER IN and BUFFER OUT must be of record type S.

For a FORTRAN 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 the statement:

```
CALL CONNEC (u)
```

Data input or output through a terminal under NOS in NORMAL mode is represented ordinarily in 6-bit display code. 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 to input in ASCII mode. 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 6/12-bit display code if the ASCII command has been entered.

Under any system, if a file specified in a CALL 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; under NOS, the file is returned.

**CALL DISCON (u)**

This subroutine disconnects a file from within a FORTRAN program.

This request is ignored if the specified file is not connected. After execution of this statement under NOS/BE 1, the specified file remains local to the terminal. In addition, if the file existed prior to connection, the file name is re-associated 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 1, a CALL DISCON causes the connected file to be returned; the disconnected file name is not re-associated with the pre-existing information.

All files to be connected or disconnected during program execution must be declared in the PROGRAM statement. An attempt to connect or disconnect an undeclared file results in a fatal diagnostic.

Calls to CONNEC and DISCON are recognized and ignored when programs are not executed under INTERCOM or interactively under NOS 1.

Examples:

```
CALL CONNEC (6)

K = 4LAGES
CALL CONNEC (K)

CALL CONNEC (6,2)

CALL CONNEC (4LDATA,1)

CALL DISCON (6)
```

# 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. Record Manager word addressable file organization is used to implement MSIO files. The 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

Random 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 differ 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 randomly accessible file capability is provided by the mass storage input/output subroutines.

In a random file, any record may 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 concept of file position does not apply to a random file. The notion of rewinding a random file is, for instance, without meaning.

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 may result in destruction of the file contents. (In the case of a sub-index, the user must clear the array before using it as a sub-index, and read the sub-index 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 1). Under NOS/BE 1 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 clear the assigned index array. 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 may continue.

## MASS STORAGE SUBROUTINES

Object time input/output subroutines control the transfer of records between central memory and mass storage.

### OPENING A FILE

OPENMS opens the mass storage file and informs the system that it is a random (word addressable) file.

**CALL OPENMS (u,ix,lngth,t)**

u Unit designator.

ix Name of the array containing the master index.

lngth Length of master index

for a number index: lngth ⩾ (number of entries in master index) + 1

f for a name index: lngth ⩾ 2 * (number of entries in master index) + 1 t Type of index.

| | |
|---|---|
| t = 0 | file has a number master index |
| t = 1 | file has a name master index |

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 master index of the number type. If the file already exists, the master index is read into memory starting at address I.

## WRITING RECORDS

WRITMS transmits data from central memory to the file.

**CALL WRITMS (u,fwa,n,k,r,s)**

u Unit designator.

fwa Name of the array in central memory (address of first word).

n Number of 60-bit words to be transferred.

k Record key.

| | |
|---|---|
| for number index: | $1 \leqslant k \leqslant$ lngth - 1 |
| for name index | k = any 60-bit quantity except $\pm 0$ |

r Rewrite.

| | |
|---|---|
| r = 1 | Rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length. |
| 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). |

s Sub-index flag.

| | |
|---|---|
| s = 1 | Write sub-index marker flag in index control word for this record. |
| s = 0 | Do not write sub-index marker flag in index control word (default value). |

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.

Except under SCOPE 2, 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 is for future file editing routines. Current routines do not test the flag, but the user should include this parameter in new programs (when appropriate) to facilitate transition to a future edit capability.

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.

## READING RECORDS

READMS transmits data from the file to central memory.

**CALL READMS (u,fwa,n,k)**

u Unit designator

fwa Name of the array in central memory (address of first word)

n Number of 60-bit words to be transferred. If n is less than the record length, n words are transferred without diagnostic.

k Record key

for number index: $k = 1 \leqslant k \leqslant \text{lngth} - 1$

for name index: k = any 60-bit quantity except $\pm 0$

Except under SCOPE 2, Record Manager operates more efficiently if n is always a multiple of 64.

Example:

```
CALL READMS (3,DATAMOR,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 DATAMOR.

## CLOSING A FILE

CLOSMS writes the master index from central memory to the file and closes the file. CLOSMS is provided to close a file so that it can be returned to the operating system before the end of a FORTRAN run, to preserve a file created by an experimental job that might subsequently abort, or for other special purposes. In an overlay program that is STATICly loaded, a mass storage file must be closed explicitly by CLOSMS.

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/BE1 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 (i.e. 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.

**CALL CLOSMS (u)**

u Unit designator

Example:

```
CALL CLOSMS (2)
```

This statement closes the file TAPE2.

## SPECIFYING A DIFFERENT INDEX

STINDX selects a different array to be used as the current index to the file. The call permits a file to be manipulated with more than one index. For example, when the user wishes to use a sub-index instead of the master index, STINDX is called to select the sub-index as the current index. The STINDX call does not cause the sub-index 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 — Unit designator.

ix — Name of the array in central memory containing the sub-index (first word address).

lngth — Length of sub-index

- for a number index: lngth ⩾ (number of entries in sub-index) + 1
- for a name index: lngth ⩾ 2 * (number of entries in sub-index) + 1

t — Type of index. If omitted, t is the same as the current index.

- t = 0 File has a number sub-index
- t = 1 File has a name sub-index

Example 1:

```
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. The records referenced via this sub-index 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 may 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 sub-indexes 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.

## SUB-INDEX

The sub-index type can be specified independently for each sub-index. A different sub-index 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 sub-index type specification is the responsibility of the user; no error message is issued.

## MULTI-LEVEL 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 (sub-indexes) 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 sub-index array. This process may be chained as many times as required, limited only by the amount of central memory space available. (Each active sub-index requires an index array area.) The mass storage routine uses the sub-index just as it uses the master index; no distinction is made.

A separate array space must be declared for each sub-index that will be in active use. Inactive sub-indexes may, of course, be stored in the random file as additional data records.

The sub-index 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 sub-index array area. The user must clear the sub-index array to zeros. If an existing file is being manipulated and the sub-index already exists on the file, the user must read the sub-index from the file into the sub-index array by a call to READMS before STINDX is called. STINDX then informs the mass storage routine to use this sub-index as the current index. The first WRITMS to an existing file using a sub-index 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 sub-index, it must be written to the file after the current index designation has been changed back to the master index, or a higher level sub-index by a call to STINDX.

Example 1 creates and modifies a random file using a number index:

```
      PROGRAM MS1 (TAPE3)

C  CREATE RANDOM FILE WITH NUMBER INDEX.

      DIMENSION INDEX(11), DATA(25)
      CALL OPENMS (3,INDEX,11,0)

      DO 99 NRKEY=1,10
C                .
C                .
C  (GENERATE RECORD IN ARRAY NAMED DATA.)
C                .
C                .
 99   CALL WRITMS (3,DATA,25,NRKEY)

      STOP
      END


      PROGRAM MS2 (TAPE3)

C  MODIFY RANDOM FILE CREATED BY PROGRAM MS1.
C  NOTE LARGER INDEX BUFFER TO ACCOMMODATE TWO NEW
C  RECORDS.

      DIMENSION INDEX(13), DATA(25), DATAMOR(40)
      CALL OPENMS (3,INDEX,13,0)
```

```
C  CLEAR THE SUBINDEX AREA.
      DO 77 I=1,10
 77   SUBIX(I)=0

C  CHANGE THE INDEX IN CURRENT USE TO SUBIX.
      CALL STINDX (2,SUBIX,10)

C  GENERATE AND WRITE NINE RECORDS.
      DO 88 MINOR=1,9
C                 .
C                 .

C  WRITE A RECORD.
 88   CALL WRITMS (2,RECORD,50,MINOR)

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)

 99   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)
C                 .
C                 .
C  (MANIPULATE THE SELECTED RECORD AS DESIRED.)
C                 .
C                 .

      STOP
      END
```

Example 4:

```
      PROGRAM MS5 (INPUT,OUTPUT,TAPE9)

C  CREATE FILE WITH NAME INDEX AND TWO LEVELS OF SUBINDEX.

      DIMENSION STATE(101), COUNTY(501), CITY(501), ZIP(100)
      INTEGER STATE, COUNTY, CITY, ZIP
 10   FORMAT (A10,I10)
 11   FORMAT (I10)
 12   FORMAT (5X,8I15)

      CALL OPENMS (9,STATE,101,1)
```

```
C  READ MASTER DECK CONTAINING STATES, COUNTIES, CITIES

C  AND ZIP CODES.
      DO 99 NRSTATE=1,50
      READ 10,STATNAM, NRCNTYS

C  CLEAR THE COUNTY SUBINDEX.
      DO 21 I=1,501
 21   COUNTY(I)=0

      DO 98 NRCN=1,NRCNTYS
      READ 10, CNTYNAM, NRCITYS

C  CLEAR THE CITY SUBINDEX.
      DO 31 I=1,501
 31   CITY(I)=0

      CALL STINDX (9,CITY,501)

      DO 97 NRCY=1,NRCITYS
      READ 10, CITYNAM, NRZIP
C  CLEAR THE ZIP CODE LIST
      DO 41, J=1,100
41    ZIP(J)=0
      DO 96 NRZ=1,NRZIP
 96   READ 11,ZIP(NRZ)

 97   CALL WRITMS (9,ZIP,NRZIP,CITYNAM)

      CALL STINDX (9,COUNTY,501)
 98   CALL WRITMS (9,CITY,501,CNTYNAM)

      CALL STINDX (9,STATE,101)
 99   CALL WRITMS (9,COUNTY,501,STATNAM)

C  FILE IS GENERATED.  NOW PRINT OUT LOCAL ZIP CODES.

      CALL STINDX (9,STATE,101)
      CALL READMS (9,COUNTY,501,≠CALIFORNIA≠)
      CALL STINDX (9,COUNTY,501)
      CALL READMS (9,CITY,501,≠SANTACLARA≠)
      CALL STINDX (9,CITY,501)
      CALL READMS (9,ZIP,100,≠SUNNYVALE≠)
      PRINT 12, ZIP

      CALL STINDX (9,STATE,101)

      STOP
      END
```

```
C  READ 8TH RECORD FROM FILE TAPE3.
      CALL READMS (3,DATA,25,8)
C                .
C                .
C  (MODIFY ARRAY NAMED DATA.)
C                .
C                .

C  WRITE MODIFIED ARRAY AS RECORD 8 AT END OF
C  INFORMATION IN THE FILE
      CALL WRITMS (3,DATA,25,8)

C  READ 6TH RECORD.
      CALL READMS (3,DATA,25,6)
C                .
C                .
C  (MODIFY ARRAY.)
C                .

C                .

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,DATAMOR,25,2)
C                .
C                .
C  (ADD 15 NEW WORDS TO THE ARRAY NAMED DATAMOR.)
C                .
C                .

C  CALL FOR IN-PLACE REWRITE OF RECORD 2.  IT WILL
C  DEFAULT TO A NORMAL WRITE AT END-OF-INFORMATION
C  SINCE THE NEW RECORD IS LONGER THAN THE OLD ONE,
C  AND FILE SPACE IS THEREFORE UNAVAILABLE.
      CALL WRITMS (3,DATAMOR,40,2,-1)

C  READ THE 4TH AND 5TH RECORDS.
      CALL READMS (3,DATA,25,4)
      CALL READMS (3,DATAMOR,25,5)
C                .
C                .
C  (MODIFY THE ARRAYS NAMED DATA AND DATAMOR.)
C                .
C                .
```

```
C  WRITE THE ARRAYS TO THE FILE AS TWO NEW RECORDS.
      CALL WRITMS (3,DATA,25,11)
      CALL WRITMS (3,DATAMOR,25,12)

      STOP
      END
```

Example 2 uses a name index for a random file:

```
      PROGRAM MS3 (TAPE7)

C  CREATE A RANDOM FILE WITH NAME INDEX.

      DIMENSION INDEX(9), ARRAY(15,4)
      DATA REC1,REC2/7HRECORD1,≠RECORD2≠/
      CALL OPENMS (7,INDEX,4,1)
C                .
C                .
C  (GENERATE DATA IN ARRAY AREA.)
C                .
C                .

C  WRITE FOUR RECORDS TO THE FILE.  NOTE THAT
C  KEY NAMES ARE RECORD(N).
      CALL WRITMS (7,ARRAY(1,1),15,REC1)
      CALL WRITMS (7,ARRAY(1,2),15,REC2)
      CALL WRITMS (7,ARRAY(1,3),15,7HRECORD3)
      CALL WRITMS (7,ARRAY(1,4),15,≠RECORD4≠)

C  CLOSE THE FILE.

      CALL CLOSMS (7)

      STOP
      END
```

Example 3:

```
      PROGRAM MS4 (TAPE2)

C  GENERATE SUBINDEXED FILE WITH NUMBER INDEX.  FOUR
C  SUBINDEXES WILL BE USED, WITH NINE DATA RECORDS
C  PER SUBINDEX, FOR A TOTAL OF 36 RECORDS.

      DIMENSION MASTER(5), SUBIX(10), RECORD(50)
      CALL OPENMS (2,MASTER,5,0)

      DO 99 MAJOR=1,4
```

## COMPATIBILITY WITH PREVIOUS MASS STORAGE ROUTINES

FORTRAN Extended mass storage routines and the files they create are not compatible with mass storage routines and files created under versions of FORTRAN Extended before version 4. Major internal differences in the file structure were necessitated by adding the Record Manager interface. However, source programs are fully compatible. Any source program that compiled and executed successfully under earlier versions will do so under this version, provided that all file manipulated by mass storage routines are manipulated only by these routines.

# FORTRAN—CYBER RECORD MANAGER INTERFACE

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 five file organizations are not discussed. In order to use these routines, it is necessary to refer to the CYBER Record Manager publications listed in the preface.

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 1 and NOS 1, but not under SCOPE 2.

## 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, 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.

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 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 below. 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.

| | |
|---|---|
| fit | Name of array containing file information table. Linked to the actual file by means of the LFN field. |
| afit | Name of an array that contains a list of addresses of FITs terminated by a word of zeros. |
| 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. |
| pd | Processing direction established when file is opened: |

| | |
|---|---|
| 5LINPUT | Read only |
| 6LOUTPUT | Write only |
| 3LI-O | Read and write |
| 3LNEW | File creation (indexed sequential, direct access, actual key only) |

of File positioning at open time:

| | |
|---|---|
| 1LR | Rewind |
| 1LN | No file positioning |
| 1LE | Extend; file is positioned immediately before end of information |

cf File positioning after close:

| | |
|---|---|
| 1LR | Rewind |
| 1LN | No positioning |
| 1LU | Unload |
| 3LRET | Return |
| 3LDIS | Disconnect (terminal files only) |
| 3LDET | No positioning; release buffer space and remove from active file list |

| | | | |
|---|---|---|---|
| type | Type of close (not a file information table field): | | |
| | | 4LFILE | File close |
| | | 6LVOLUME | 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. | | |
| 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: | | |
| | | 1LP | Write record preceding current record |
| | | 1LN | Write record as next record |
| | | 1LC | 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 |
| | | 4LSKIP | Skip to beginning of next record before reading |
| lev | Level number for end of section; 0 to 17. | | |
| id | FIT identifier. | | |

†Applies only to Initial Indexed Sequential files.

## SUBROUTINES

In the subroutine formats below, braces are used to indicate that more than one parameter occupies the same position. In all cases, these parameters are applicable to mutually exclusive file organizations.

**CALL FILExx (fit, keyword$_1$, value$_1$, . . . ,keyword$_n$, value$_n$)**

xx is SQ (for sequential files), IS (for indexed sequential files), DA (for direct access files), AK (for actual key files) or WA (for word addressable files).

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. CALL FILExx must be executed before the file is opened. CALL FILExx ensures that the object libraries BAMLIB and AAMLIB are made available to the job.

**CALL STOREF (fit, keyword, value)**

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.

**IFETCH(fit,field) or CALL IFETCH(fit,field,value)**

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; in which case, the value is returned in the integer variables specified as the third parameter.

**CALL OPENM(fit,pd,of)**

OPENM opens a file and prepares it for further processing. Only FILExx, STOREF, and IFETCH can precede execution of CALL OPENM.

**CALL CLOSEM (fit,cf,type)**

CLOSEM closes the file after all processing has been completed. Only STOREF and IFETCH can follow execution of CLOSEM.

**CALL GET(fit,wsa,** $\left\{\begin{matrix}\textbf{ka}\\\textbf{wa}\end{matrix}\right\}$**,kp,mkl,rl,** $\left\{\begin{matrix}\textbf{ex}\\\textbf{dx}\end{matrix}\right\}$ **)**

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.

**CALL PUT(fit,wsa,rl,** $\left\{\begin{matrix}\textbf{ka}\\\textbf{wa}\end{matrix}\right\}$**,kp,pos,ex)**

PUT writes a record to the file from the working-storage area (wsa).

**CALL GETP(fit,wsa,ptl,skip,dx)**

GETP reads a partial record. The number of characters to be read is indicated by ptl.

**CALL PUTP(fit,wsa,ptl,rl,ex)**

PUTP writes a partial record. The number of characters to be written by this write 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).

**CALL GETN(fit,wsa,ka,ex)**

GETN reads the next record in sequential order from an indexed sequential, direct access, or actual key file. The key of the record read is placed in ka after the read.

**CALL DLTE(fit,ka,kp,pos,ex)**

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.

**CALL REPLC(fit,wsa,rl,ka,kp,pos,ex)**

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.

**CALL WEOR(fit,lev)**

WEOR terminates a section or partition, or S type record.

**CALL WTMK(fit)**

Writes a tape-mark (equivalent to end of partition).

**CALL ENDFILE(fit)**

Writes an end of partition.

**CALL REWND(fit)**

REWND positions a tape file to the beginning of the current volume. It positions a mass storage file to the beginning of information.

**CALL GETNR(fit,wsa,ex,ka)**

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).

CALL FLUSHM(afit)

FLUSHM performs all file close operations (such as buffer flushing), but the file remains open.

CALL FLUSH1(fit)

FLUSH1 performs the same function as FLUSHM, but for a single file instead of a list of files.

CALL FITDMP(fit,id)

FITDMP dumps the contents of the file information table to the error file ZZZZZEG. The CRMEP control statement (see the CYBER Record Manager AAM reference manual) can then be used to print file ZZZZZEG.

CALL SEEKF(fit,ka,kp,inkl,ex)

SEEKF initiates block transfer to the file buffer. The program can continue processing while the transfer occurs. This overlapping of central memory processing and input/output activity can shorten program execution time.

CALL SKIP(fit,count)

SKIP repositions an indexed sequential or actual key file in a forward or backward direction a specified number of records. It does not return a record to the working storage area. A positive value for count indicates a forward move; a negative value indicates a backward move.

CALL STARTM(fit,ka,kp,mkl,ex)

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. A new key address value is stored in ka.

## 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 <u>of</u> parameter in OPENM was not specified as R, N or E. |

## MULTIPLE INDEX PROCESSING

FORTRAN Extended provides the capability of multiple indexing for IS, DA, and AK files via CYBER Record Manager.

Each multiple-indexed 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 following statement is used:

**CALL RMOPNX(fit,pd,of)**

The parameters are the same as those of CALL OPENM. The file may 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.

The following subroutine 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) |

To position a multiple index file, the following subroutine is used:

**CALL STARTM(fit,ka,kp,mkl,ex)**

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

FORTRAN Extended 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. For this reason the STATIC control statement parameter must not be used for programs using SORT/MERGE.

The FORTRAN subroutines interfacing with Sort/Merge are listed below. 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. The Sort/Merge subroutines are on the library SRTLIB.

In an overlay structured program using blank common, the Sort/Merge interface routines must not be called from the (0,0) overlay.

**CALL SMSORT(mrl,ba)**

| | |
|---|---|
| mrl | Maximum length in characters of records to be sorted. |
| ba§ | LCM buffer area in decimal for intermediate scratch files constructed by Sort/Merge. |
| ba‡ | Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. |

SMSORT calls for a sort on rotating mass storage.

**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 calls for a balanced tape sort. SMTAPE (see below) must also be called.

**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. |

---

§Applies only to SCOPE 2.

‡Applies only to NOS 1 and NOS/BE 1.

SMSORTP calls for a polyphase tape sort. SMTAPE must also be called.

**CALL SMMERGE(mrl,ba)**

| | |
|---|---|
| mrl | Maximum length in characters of records to be merged. |
| ba§ | LCM buffer area in decimal for intermediate scratch files constructed by Sort/Merge. |
| ba‡ | Number of words of central memory to be used by Sort/Merge for working storage. If omitted, amount is computed by Sort/Merge. |

SMMERGE calls for merge-only processing.

The MRL option on the SMSORT, SMSORTB, SMSORTP, and SMMERGE calls is used only when one or more files are FORTRAN files. If all files are processed through direct calls to CYBER Record Manager (access = 0 in SMFILE calls), the MRL option is required only as a dummy parameter.

**CALL SMFILE(dis,i/o,lfn,action)**

dis — File disposition:

| | |
|---|---|
| ≠SORT≠ | File to be sorted. |
| ≠MERGE≠ | File to be merged. |
| ≠OUTPUT≠ | File to receive output. |

i/o — Mode of file input/output:

| | |
|---|---|
| ≠FORMATTED≠<br>≠CODED≠ | File accessed with formatted input/output. |
| ≠BINARY≠ | File accessed with unformatted input/output. |
| 0‡ | File accessed with interfacing CYBER Record Manager subroutines (see this section above). |

lfn — File name indicator:

| | |
|---|---|
| u | Logical unit number, 0 to 99. |
| nLfilename | File name left justified with zero fill. |
| fit‡ | When i/o is specified as 0, an array containing the file information table. |

action — File disposition following sort or merge:

≠REWIND≠
≠UNLOAD≠
≠NONE≠ (default)

§Applies only to SCOPE 2.

‡Applies only to NOS 1 and NOS/BE 1.

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). If a file is to be accessed with formatted or unformatted FORTRAN input/output, its name must be declared in the PROGRAM statement. Files should be properly positioned before they are sorted or merged.

**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:

| | |
|---|---|
| ≠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 — 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. |
| sequence | Name of a collating sequence specified in a call to SMSEQ (see below). |

If a code identifier other than DISPLAY is used, this field must be omitted; otherwise, run time error 165 is issued.

order — Order of sort processing.

| | |
|---|---|
| ≠A≠ | Ascending (default). |
| ≠D≠ | Descending. |

One SMKEY call is required to describe each sort key to be used. The first SMKEY call indicates the major key; subsequent calls indicate additional or minor keys in the order encountered.

**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.

SMSEQ 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.

The characters in seqspec can be specified as their octal equivalents in the form ijB or as Hollerith constants in the form 1Rx. Characters to collate equal are specified in a call to SMEQU (see below). Unspecified characters collate high (following the last character specified in seqspec) and equal.

**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.

SMEQU specifies that two or more characters in the collating sequence are equal for comparison purposes.

**CALL SMOPT ($opt_1$, . . . , $opt_n$)**

opt — Non-ordered 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 (decimal) n 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.

SMOPT specifies special record handling options. If SMOPT is called more than once, the last call will override all previous calls. If SMOPT is called, it must be done immediately after the call to SMSORT or SMMERGE.

‡Applies only to NOS 1 and NOS/BE 1.

**CALL SMTAPE (taplist)‡**

taplist — List of logical file names, each in the form nLfilename, to be used in balanced or polyphase tape merge.

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, a minimum of three tapes.

**CALL SMOWN ($exitnum_1$,$subname_1$, . . . , $exitnum_n$,$subname_n$)**

exitnum — Number of the owncode exit.

subname — Name of the user-supplied owncode exit subroutine

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 follows:

| exitnum | entry | exit |
|---|---|---|
| 1 or 3 | SUBROUTINE subname (a,rl) | CALL SMRTN (retaddr), for retaddr = 1 or 3<br>CALL SMRTN (retaddr,b,rl), for retaddr = 0 or 2 |
| 2 or 4 | SUBROUTINE subname | CALL SMRTN (retaddr), for retaddr = 0<br>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<br>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. Storing into a causes indeterminate results.

b — Integer array of length (rl + 9)/10 in which the user stores a record when subname is called. b should not be the same as a.

rl — Record length in characters.

No parameters are needed on SUBROUTINE subname for exit number 1 if there are no input files.

**CALL SMEND**

Required as the last in a series of Sort/Merge interfacing subroutines, SMEND initiates execution of the sort or merge.

**CALL SMABT**

Terminates a sequence of SORT/MERGE interface calls without calling Sort/Merge. The state of the interface is the same as if no calls had been made.

---

‡Applies only to NOS 1 and NOS/BE 1.

# FORTRAN-CYBER INTERACTIVE DEBUG INTERFACE

CYBER Interactive Debug (CID) is a debugging facility, available under NOS 1 and NOS/BE 1, which allows the user to monitor and control the execution of programs from an interactive terminal. CID is on the library DBUGLIB.

A brief discussion of CID is presented here. For more information, refer to the CYBER Interactive Debug reference manual.

FORTRAN Extended provides the capability of interfacing with CID. The CID features allow the user to:

Suspend program execution at specified locations called breakpoints.

Set traps which cause program execution to be suspended on specific events, such as the loading of an overlay.

Display values stored into variables and arrays while program execution is suspended.

Enter data into the program.

Interrupt and restart the program from the terminal.

Define and save sequences of CID commands to be executed automatically when a breakpoint or trap is encountered during program execution.

## CONTROL STATEMENT

In order to make use of all the CID facilities, a FORTRAN program must be compiled, loaded and executed in debug mode. Debug mode is activated by the control statement

DEBUG or DEBUG(ON)

When a source program is compiled in debug mode, the compiler produces a line number table and a symbol table along with the binary object code. The CID package is loaded along with the compiled code and becomes part of the user's field length.

CID is deactivated by the control statement

DEBUG(OFF)

As an alternative to compiling with DEBUG(ON), the necessary compiler tables can be produced by specifying DB or DB=ID on the FTN control statement. Subsequent executions with DEBUG(ON) can make use of CID.

If debug mode has been activated with DEBUG(ON), it can be subsequently turned off for the duration of a compilation by specifying DB=0 on the FTN control statement. The default is DB=0.

A program that has been compiled with DEBUG(ON) or DB=ID can subsequently be executed with DEBUG(OFF), but CID cannot be used.

## USER-CID INTERACTION

In debug mode, after the user's program has been loaded, but before execution is initiated, CID requests input of commands. Typically, the user initially sets breakpoints and traps which specify debugging options to be performed during program execution.

When a breakpoint or trap is encountered during execution, execution is suspended while CID performs the sequence of commands specified in the body of the breakpoint or trap definition. With certain breakpoints or traps, the user has the option of entering debug commands at the terminal before execution is resumed.

## CID OUTPUT

Output from CID consists of informative messages, diagnostics, and the results of commands. Certain informative messages always appear at the terminal; other messages are arranged into classes, and the user can specify which message classes are to be sent to the terminal.

## BATCH DEBUGGING

CID is primarily intended to be used interactively, but can be used in batch mode. In this case, the user must place CID commands as the first record in the file DBUGIN.

Output from CID is written to a file called DBUGOUT. The type of output written to this file is controlled in the same manner in which output is sent to the terminal when CID is used interactively.

# INTERFACE TO COMMON MEMORY MANAGER

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.

Interface to CMM can be done to assign blocks of memory for arrays. This assignment is completely dynamic, and for efficient use, the blocks should be returned to the system when finished.

The Common Memory Manager 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 dynamic 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 on the array name, and subtracting this value from the first word address of the block returned by CMM. The array element n can then be referenced by array name (offset + n). 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))
CMMAR(IOFF +5)=1.0
  .
  .
  .
```

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 addresses.
- 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).
- >0 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 should include a CALL SYMLIB subroutine call in the main program. SCOPE 2 users must specify SYMIO in the LDSET statement instead of SYMLIB.

## POST MORTEM DUMP

Post Mortem Dump (PMD) analyzes the execution time errors in FORTRAN Extended Version 4 programs. PMD provides interpreted output in a form which is more easily understood than the octal dump normally output following a fatal error; PMD prints a 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. The names and values of the variables in the routine in which the error was detected are printed; this process is repeated, tracing back through the calling sequence of routines until the main program is reached.

Use of PMD does not affect the use of FORTRAN Extended DEBUG or CYBER Interactive Debug. PMD is activated by a hardware or software fatal error and can also intentionally be invoked by the user. PMD overrides any user-supplied load map directive or MAP(ON) control statement. For example, the following statements do not produce a load map if PMD was specified:

```
LDSET(MAP=SBEX)
LOAD(LGO)
EXECUTE.
```

However, the loader always writes a block and statistics map to file ZZZZZMP for PMD's use. It is the user's responsibility to rewind and copy this file to output. If nonfatal loader errors occur, a summary of the errors is included in the PMD output.

When PMD is used, the FORTRAN Extended 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.

PMD reloads the user field length before it aborts to allow a subsequent octal dump of the user's program if one has been specified.

To use PMD, the PMD parameter must be specified on the FTN 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; an attempt is made to associate each address register with a variable or array referenced within the routine in which the error occurred.

An alphabetical 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 alphabetically. The column labeled RELOCATION is left blank for local variables. It contains the block name for COMMON variables and F.P. nn for formal parameters, where nn indicates the parameter number.

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 in 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, integer.

For batch jobs, the dump is written to 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.

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.

F A full dump is written to the file PMDUMP when the job is executed with the file OUTPUT connected. This option is valid for interactive jobs only and is the default if the *OP parameter is omitted.

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 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. 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 PMD might differ for successive executions of the same program on certain computer systems. This can occur on systems with parallel functional units such as the 6600, 6700, CYBER 70 models 74 and 76, and the CYBER 170 models 175, 176, 750, and 760.

In addition to the file ZZZZZMP created by the loader, PMD requires a file called ZZZZZSY, which is created by the compiler. If a program that uses PMD is to be compiled, loaded, and executed in separate jobs, those files must be saved and must be available to PMD during execution.

PMD can, in most cases, trace back from the point at which an error occurs to the main program. Traceback occurs repeatedly through the calling sequence of routines until the main program is reached. In PMD traceback, PMD prints the names and values of the variables in the routine in which an error is detected.

Traceback is abandoned under the following conditions:

PMD discovers a routine name that occurred earlier in the traceback. For example, the descendent of an ancestor routine has called the ancestor again.

Traceback flows through a routine that was not invoked by standard FORTRAN calling conventions, such as those used in a function or subroutine reference. For example, if an abort occurs while an input/output operation is being performed in a CRM routine, the traceback is abandoned because CRM uses the sequence:

```
SB6 address
JP  B6
```

to perform the equivalent of the standard calling sequence:

```
RJ  address
```

that a function or subroutine reference generates.

Multiple entry points exist in a system resident routine. If multiple entry points are in a system resident routine, traceback might be abandoned because PMD continues traceback from the first entry point (which is assumed to be the last active entry point for the routine). If traceback continues at an entry point other than the last active entry point for the routine, traceback continues up a calling chain other than the one that produced the error. CPU.SYS is an example of a routine with multiple entry points that is subject to this condition.

When traceback is abandoned the following error message is issued:

```
***TRACEBACK ABANDONED
```

The formats of the optional PMD subroutine calls are as follows:

```
CALL  PMDARRY(i)
CALL  PMDARRY(i,j)
CALL  PMDARRY(i,j,k)
```

The last subroutine call listed causes dump of arrays to be limited to elements whose subscripts do not exceed i, j, and k for their respective dimensions; i, j, and k represent the first, second, and third dimensions, respectively.

If k is omitted, three-dimensional arrays are not printed. If j and k are omitted, two- and three-dimensional arrays are not printed; only one-dimensional arrays are printed.

Array dumping parameters can also be specified on the LGO call card. The three formats are:

| | | |
|---|---|---|
| LGO,*DA=I | Corresponds to call | PMDARRY(I). |
| LGO,*DA=I+J | Corresponds to call | PMDARRY(I,J). |
| LGO,*DA=I+J+K | Corresponds to call | PMDARRY(I,J,K). |

where I, J, and K represent the first, second, and third dimensions, respectively.

If neither CALL PMDARRY nor LGO,*DA= is used, the default array dimensions of I, J, and K are assumed to be 20, 2, and 1, respectively.

Once PMDARRY has been called, the established conditions apply to all program units in the user program. Any number of PMDARRY 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)(2,3,2)(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 subblock have the same value, PMD will print the message:

ALL THREE ELEMENTS WERE . . . .

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. These active routines have been executed but are not necessarily in the traceback chain. Following an abort or call to PMDSTOP, all routines in the traceback chain are dumped. A limit of ten successive calls to PMDDUMP is imposed. The tenth call to PMDDUMP is converted to a PMDSTOP call.

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 more than 10 times, in which case a nonreturnable call to PMDSTOP occurs.

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.

An example of Post Mortem Dump output follows:

```
PROGRAM EXAMPL      76/6     OPT=1 TRACE   PMDMP                  FTN 4.3+80080*T*    80/06/05. 11.55.37        PAGE     1

  1               PROGRAM EXAMPL(INPUT,OUTPUT,TAPE1,TAPE6=OUTPUT)                          EXAMPL
           C                                                                               EXAMPL
           C      THIS PROGRAM ILLUSTRATES SOME OF THE FEATURES OF                         EXAMPL
           C      THE PMDMP FEATURE                                                        TNAME
  5        C                                                                               EXAMPL
                  COMPLEX CVAR1                                                            EXAMPL
                  LOGICAL LVAR1                                                            TESTJ1
                  DIMENSION IARRAY(10,2),ARRAY2(30)                                        EXAMPL
                  COMMON /BLOCKA/ ARRAY1(7,6,6)                                            EXAMPL
 10               EQUIVALENCE (MASK1,RMASK)                                                EXAMPL
                  DATA ((IARRAY(I,J),I=1,10),J=1,2) /10*3,10*-4/                           EXAMPL
                  ARRAY2(17)=1717.1717                                                     EXAMPL
                  CVAR1=(1.0,1.5)                                                          EXAMPL
                  RVAR1=*ANTELOPE*                                                         TESTJ1
 15               LVAR1=.FALSE.                                                            TESTJ1
                  IVAR1=10HANT                                                             EXAMPL
                  IVAR2=3LANT                                                              EXAMPL
                  IVAR3=3RANT                                                              EXAMPL
                  IVAR4=INT(10000.1)                                                       EXAMPL
 20               MASK1=MASK(48)                                                           EXAMPL
                  CALL EXTRAS(0,VAR1,IARRAY(1,2),10,IARRY2)                                EXAMPL
                  STOP                                                                     EXAMPL
                  END                                                                      EXAMPL


      SYMBOLIC REFERENCE MAP (R=2)

ENTRY POINTS      DEF LINE      REFERENCES
 6215  EXAMPL         1

VARIABLES      SN  TYPE              RELOCATION
    0  ARRAY1      REAL          ARRAY     BLOCKA          REFS        9
 6313  ARRAY2      REAL          ARRAY                     REFS        8    DEFINED       12
 6254  CVAR1       COMPLEX                                 REFS        6    DEFINED       13
 6267  IARRAY      INTEGER       ARRAY                     REFS        8       21   DEFINED       11
 6265  IARRY2   *  INTEGER                                 REFS       21
 6260  IVAR1    *  INTEGER                              DEFINED       16
 6261  IVAR2    *  INTEGER                              DEFINED       17
 6262  IVAR3    *  INTEGER                              DEFINED       18
 6263  IVAR4    *  INTEGER                              DEFINED       19
 6256  LVAR1       LOGICAL                                 REFS        7    DEFINED       15
 6266  MASK1       INTEGER                                 REFS       10    DEFINED       20
 6266  RMASK       REAL                                    REFS       10
 6257  RVAR1    *  REAL                                 DEFINED       14
 6264  VAR1     *  REAL                                    REFS       21

FILE NAMES          MODE
    0  INPUT
 2054  OUTPUT
 4130  TAPE1
 2054  TAPE6

EXTERNALS           TYPE    ARGS        REFERENCES
       EXTRAS                 5             21

INLINE FUNCTIONS    TYPE    ARGS        DEF LINE   REFERENCES
       INT        INTEGER     1   INTRIN               19
       MASK       NO TYPE     1   INTRIN               20

COMMON BLOCKS   LENGTH
       BLOCKA      252

STATISTICS
  SYMTAB+DIMTAB                  1128      74
  PROGRAM LENGTH                 411B     265
  BUFFER LENGTH                 5740B    3040
  SCM LABELED COMMON LENGTH      374B     252
         65200B SCM USED
```

```
SUBROUTINE EXTRAS     76/6    OPT=1 TRACE  PMDMP               FTN 4.3+80080*T*   80/06/05. 11.55.37        PAGE     2

  1                SUBROUTINE EXTRAS(I,J,IARRAY,N,IARRY2,UNUSED)                            EXAMPL
                   DOUBLE PRECISION DVAR1                                                   TESTJ1
                   INTEGER IARRAY(N),IARRY2(1)                                              EXAMPL
  5          C     COMMON BLOCK LENGTHS DISAGREE -- ERROR TRIGGERS CALL TO PMDMP
                   COMMON /BLOCKA/ ARRAY1(6,6,6)                                            EXAMPL
                   READ (1) J                                                               EXAMPL
                   REWIND 1                                                                 EXAMPL
 10                WRITE (6,1000)                                                           EXAMPL
          1000     FORMAT(///* THIS IS AN FTN POST MORTEM DUMP EXAMPLE *)                   TNAME
                   I=3                                                                      EXAMPL
                   X=3.0                                                                    TESTJ1
                   IVAR5=IVAR6                                                              EXAMPL
 15                DVAR1=DSQRT(2.0D00)                                                      TESTJ1
                   PI=3.1415926                                                             TESTJ1
           100     J = 0
                   J = SIN(FLOAT(J))
             C     INFINITE OPERAND HERE, TRIGGERING A CALL TO PMDMP...WITH X RECEIVING
 20          C     A VALUE OF EITHER *3.00000000000* OR *POSITIVE INDEFINITE* IN THE
             C     VARIABLE LISTING... BECAUSE THIS VALUE IS MACHINE DEPENDENT, THIS WILL
             C     NOT BE CHECKED FOR IN THE COMPARISON ROUTINE
                   X=X/J                                                                    TESTJ1
                   X=X+X                                                                    TESTJ1
 25                CALL ABSENT                                                              EXAMPL
                   IF(J.EQ.0) GO TO 100                                                     TESTJ1
                   RETURN                                                                   EXAMPL
                   END                                                                      EXAMPL


          SYMBOLIC REFERENCE MAP (R=2)

ENTRY POINTS      DEF LINE      REFERENCES
    3  EXTRAS         1            27

VARIABLES       SN   TYPE           RELOCATION
    0  ARRAY1        REAL        ARRAY    BLOCKA      REFS        7
   64  DVAR1         DOUBLE                           REFS        2   DEFINED      15
    0  I             INTEGER              F.P.     DEFINED        1
                                                                 12
    0  IARRAY        INTEGER     ARRAY    F.P.        REFS        3   DEFINED       1
    0  IARRY2        INTEGER     ARRAY    F.P.        REFS        3   DEFINED       1
   67  IVAR5      *  INTEGER                       DEFINED       14
   70  IVAR6      *  INTEGER     *UNDEF               REFS       14
    0  J             INTEGER              F.P.        REFS       18      23        26   DEFINED       1        8        17
                                                                 18
    0  N             INTEGER              F.P.        REFS        3   DEFINED       1
   71  PI         *  REAL                          DEFINED       16
    0  UNUSED        REAL        *UNUSED  F.P.     DEFINED        1
   66  X             REAL                             REFS       23     2*24   DEFINED       13       23       24

FILE NAMES          MODE
       TAPE1        UNFMT                    READS        8     MOTION        9
       TAPE6        FMT                      WRITES      10

EXTERNALS             TYPE    ARGS          REFERENCES
       ABSENT                  0                25
       DSQRT         DOUBLE    1 LIBRARY        15
       SIN           REAL      1 LIBRARY        18

INLINE FUNCTIONS     TYPE     ARGS          DEF LINE   REFERENCES
       FLOAT         REAL      1  INTRIN                   18

STATEMENT LABELS                   DEF LINE    REFERENCES
   24  100                            17           26
   52  1000      FMT                  11           10

COMMON BLOCKS     LENGTH
       BLOCKA       216

STATISTICS
  SYMTAB+DIMTAB                     1128B        74
  PROGRAM LENGTH                     724B        58
  SCM LABELED COMMON LENGTH         3330B       216
          652008 SCM USED


THIS IS AN FTN POST MORTEM DUMP EXAMPLE
```

```
FTN POST MORTEM DUMP          FTN 4                         ERROR  REPORT                          80/06/05. 11.55.46.

*** YOUR JOB HAS THE FOLLOWING NON-FATAL LOAD ERROR(S):
                    UNSATISFIED EXTERNAL REF -- ABSENT

*** YOUR JOB HAS THE FOLLOWING /COMMON/ LENGTH CLASH(ES):
    /BLOCKA/ ,LOADED LENGTH=   252,LENGTH IN ROUTINE EXTRAS =   216

/// EXECUTION WAS TERMINATED BECAUSE YOUR PROGRAM FAILED WITH ERROR CONDITION OVERFLOW

... ARRAYS WILL BE PRINTED BY  DEFAULT  PARAMETERS (   20,    2,    1)

... YOUR PROGRAM REQUIRED  33400B WORDS TO LOAD, 16127B WORDS TO RUN

... FILE STATUS AT TIME OF ERROR

    FILE NAME  FORTRAN NAMES  LASTOP   STATUS FO BT RT RECORD COUNT
      -INPUT       INPUT     UNUSED            SQ     W          0
      -OUTPUT      OUTPUT    PUT/PUTP  E-O-R   SQ  C  Z          4
                   TAPE6
      -TAPE1       TAPE1     REWIND    B-O-I   SQ  I  W          0

/// THE ERROR OCCURRED IN SUBROUTINE EXTRAS ,ABOUT    4 WORDS AFTER LINE     18 (    2 WORDS BEFORE LINE     25)

... THE REGISTERS CONTAINED THE FOLLOWING AT THE TIME OF THE ERROR

    A-REGISTERS (CONTAIN ADDRESSES)                                     ASSOCIATED LOCATION
     REG OCT VAL  SYMBOL                                    OCTAL VALUE              ARITHMETIC VALUE        CHAR VALUE
      A0 0067428 PARAMETER LIST ADDRESS + 0        (A0)=00000000000000067578      ADDRESS OF PARAMETER     1
      A1 0071178 ADDRESS OF A TEMPORARY            (A1)=00000000000000000008      0.
      A2 0114648   WITHIN  SINCOS.                 (A2)=04000071050000000008       .505292518332-218      5LD  +E
      A3 0115258   WITHIN  SINCOS.                 (A3)=60641516711555623306B     -.495774235001E-01
      A4 0067438 PARAMETER LIST ADDRESS + 1        (A4)=00000000000000067718      ADDRESS OF PARAMETER     2
      A5 0071448    X                              (A5)=37770000000000000008       POSITIVE INFINITE
      A6 0071448    X                              (A6)=37770000000000000008       POSITIVE INFINITE
      A7 0067718 PARAMETER  2 - J                  (A7)=00000000000000000008                   0

    B-REGISTERS                                   X-REGISTERS (USED FOR COMPUTATION)
     REG OCT VAL   DEC VAL                         REG      OCTAL VALUE              ARITHMETIC VALUE        CHAR VALUE
      B0 0000008 =       0                          X0  37770000000000000008       POSITIVE INFINITE
      B1 0071058 =    3653                          X1  00000000000000000008       0.
      B2 7760008 =   -1023                          X2  00000000004000071058            67112517          5RD  +E
      B3 0000008 =       0                          X3  00000000000000000008       0.
      B4 0000008 =       0                          X4  00000000000000067718                3577          2R-+
      B5 0000128 =      10                          X5  17216000000000000008       3.00000000000         3LDQ=
      B6 0110048 =    4612                          X6  37770000000000000008       POSITIVE INFINITE
      B7 0000008 =       0                          X7  00000000000000000008       0.

P-REGISTER 0071078     CM FIELD LENGTH 021600B     ECS/LCM FIELD LENGTH 00000000B

RA+0 00020071100000000000B      RA+1 00000000000000000008      PSD (CY176 ONLY) 0003B

... VARIABLES IN SUBROUTINE EXTRAS
        NAME         TYPE        RELOCATION      CURRENT VALUE                                COMMENTS  NAME

        ARRAY1       REAL        /BLOCKA/        ARRAY   (6,6,6)                                        ARRAY1
        DVAR1        DOUBLE                      1.41421356237309504880169                              DVAR1
        I            INTEGER       F.P. 1                 3     = 1RC                                   I
        IARRAY       INTEGER       F.P. 3        ARRAY   (10)                                           IARRAY
        IARRY2       INTEGER       F.P. 5        ARRAY   (1)                                            IARRY2
*** THE NEXT ITEM WAS SET TO AN UNINITIALIZED VALUE -    IVAR6
        IVAR5        INTEGER                   ** NOT INITIALIZED**                                     IVAR5
```

```
*** THE NEXT ITEM IS NEVER DEFINED
     IVAR6       INTEGER                        NOT INITIALIZED                                    IVAR6
     J           INTEGER      F.P. 2                       0                                       J
     N           INTEGER      F.P. 4                      10       = 1RJ                           N
     PI          REAL                          3.14159260000                                       PI
     UNUSED      REAL         F.P. 6         ** OMITTED FROM THE CALL STATEMENT   **               UNUSED
 ERROR--X        REAL                         POSITIVE INFINITE                                    X

*** ROUTINE EXPECTED  6 ARGUMENTS BUT WAS CALLED WITH  5 ARGUMENTS

... ARRAYS IN SUBROUTINE EXTRAS

    REAL     ARRAY ARRAY1(6,6,6)
                 ALL REQUESTED ELEMENTS OF THIS ARRAY WERE  NOT INITIALIZED

    INTEGER ARRAY IARRAY(10)                    DECLARED AS IARRAY(N)
                 ALL REQUESTED ELEMENTS OF THIS ARRAY WERE            -4

    INTEGER ARRAY IARRY2(1)
                 ALL REQUESTED ELEMENTS OF THIS ARRAY WERE  NOT INITIALIZED

... CALLED FROM LINE NUMBER   21 OF   PROGRAM  EXAMPL
--------------------------------------------------------------------------------------------------------------------

FTN POST MORTEM DUMP          FTN 4                       PROGRAM  EXAMPL                     80/06/05. .11.55.46.


... SITUATION AT THE TIME SUBROUTINE EXTRAS  WAS CALLED AT LINE NUMBER   21 OF   PROGRAM  EXAMPL

... VARIABLES IN   PROGRAM  EXAMPL
     NAME        TYPE       RELOCATION      CURRENT VALUE                                COMMENTS  NAME

     ARRAY1      REAL        /BLOCKA/       ARRAY   (7,6,6)                                        ARRAY1
     ARRAY2      REAL                       ARRAY   (30)                                           ARRAY2
     CVAR1       COMPLEX                    ( 1.0000000000    , 1.5000000000    )                  CVAR1
     IARRAY      INTEGER                    ARRAY   (10,2)                               ARG.   3  IARRAY
*** THE NEXT ITEM IS AN INTEGER  ARRAY   IN THE CALLED ROUTINE
     IARRY2      INTEGER                    NOT INITIALIZED                              ARG.   5  IARRY2
     IVAR1       INTEGER                   10HANT             OR INTEGER > 2**48-1                 IVAR1
     IVAR2       INTEGER                   3LANT              OR INTEGER > 2**48-1                 IVAR2
     IVAR3       INTEGER                             5012     = 3RANT                              IVAR3
     IVAR4       INTEGER                            10000     = 3RB1P                              IVAR4
     LVAR1       LOGICAL                    .FALSE.                                                LVAR1
     MASK1       INTEGER                            -4095     =48-BIT MASK                         MASK1
*** THE NEXT VARIABLE CONTAINS AN INTEGER VALUE
     RMASK       REAL                               -4095     =48-BIT MASK                         RMASK
*** THE NEXT VARIABLE IS UNNORMALIZED
     RVAR1       REAL                       .296962555974-270  10HANTELOPE                         RVAR1
*** THE NEXT ITEM IS AN INTEGER VARIABLE IN THE CALLED ROUTINE
     VAR1        REAL                      0.                                            ARG.   2  VAR1
*** THE NEXT CONSTANT  MAY HAVE BEEN ALTERED IN THE CALLED ROUTINE
     CONSTANT    INTEGER                               3      = 1RC                      ARG.   1  CONSTANT
     CONSTANT    INTEGER                              10      = 1RJ                      ARG.   4  CONSTANT

... ARRAYS IN   PROGRAM  EXAMPL

    REAL     ARRAY ARRAY1(7,6,6)
                 ALL REQUESTED ELEMENTS OF THIS ARRAY WERE  NOT INITIALIZED

    REAL     ARRAY ARRAY2(30)
 (ARRAY2(N))
    N=1,16                  ALL THESE ELEMENTS WERE  NOT INITIALIZED
    N=17     1717.17170000       NOT INITIALIZED     NOT INITIALIZED     NOT INITIALIZED
```

```
    INTEGER ARRAY IARRAY(10,2)
 (IARRAY(N,0))
    N=1,10                ALL THESE ELEMENTS WERE              3     = 1RC
 (IARRAY(N,1))
    N=1,10                ALL THESE ELEMENTS WERE             -4

••• TRACEBACK SUCCESSFULLY COMPLETED

/// END OF ERROR REPORT
```

# DEBUGGING FACILITY 9

The debugging facility allows the programmer to debug programs within the context of the FORTRAN language. Using the statements described in this section, the programmer can check the following:

| | |
|---|---|
| Array bounds | Function references and the values returned |
| Assigned GO TO | Values stored into variables and arrays |
| Subroutine calls and returns | Program flow |

The debugging facility, together with the source cross reference map, is provided specifically to assist the programmer develop or convert programs.

The debugging mode is selected by specifying D or D = lfn on the FTN control statement. This option automatically selects fast compilation (OPT=0) and full error traceback (T option). If any other optimization level is specified, it will be ignored. Specification of both D and TS results in a fatal error. The following examples are equivalent:

```
FTN (D)
FTN (D=INPUT,OPT=0,T)
FTN  (D,OPT=2)
```

OPT=2 is ignored, OPT=0 and T are automatically selected.

Debug output is written on the file DEBUG. The DEBUG file, which must be on a queue device, is given print disposition and printed separately from the output file upon job termination. To obtain debugging information on the same file as the source program, or any other queue device resident††† file, DEBUG must be equivalenced to that file in the PROGRAM statement.

Examples:

```
PROGRAM EX (INPUT,OUTPUT,DEBUG=OUTPUT)
```

Debug output is interspersed with program output on the file OUTPUT.

```
PROGRAM EX(INPUT,OUTPUT,TAPEX,DEBUG=TAPEX)
```

Debug output is written on the file TAPEX.

The following control statement sequence causes the debug output to be printed on the output file at termination of the job. It is not interspersed with the results of program execution.

```
FTN(D)
LGO.
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
EXIT(S)† or EXIT.††          Abnormal termination
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
```

†NOS/BE 1 and SCOPE 2
††NOS 1
†††NOS/BE 1

In debug mode, programs execute regardless of most compilation errors. Execution, however, terminates when a fatal error is detected, and the following message is printed:

```
FATAL ERROR ENCOUNTERED DURING PROGRAM EXECUTION
DUE TO COMPILATION ERROR
```

Partial execution of programs containing fatal errors allows the programmer to insert debugging statements to assist in locating fatal and non-fatal errors. Partial execution is prohibited for only four classes of errors:

Any declarative error (any error encountered before at least one valid executable statement is found)

Any fatal compilation error (defined in Appendix B)

Any missing (undefined) DO termination

Any illegal transfer into an innermost DO loop that is not an extended range loop

When a program is compiled in debug mode, at least 15000 (octal) words are required beyond the minimum field length for normal compilation. To execute, at least 2500 (octal) words beyond the minimum are required. The CPU time required for compilation is also greater than for normal OPT=0 compilation.

If the D option is not specified on the FTN control statement, all debugging statements are treated as comments; therefore, it is not necessary to remove debugging statements from a program.

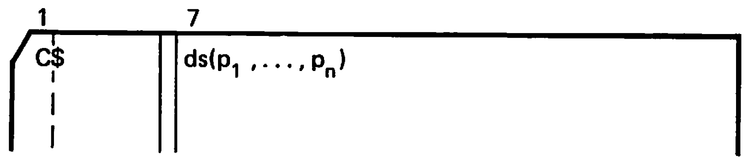
All debugging options are activated and deactivated at compile time only. This compile time processing is not to be confused with program flow at execution time.

Example:

```
      PROGRAM TEST (OUTPUT,DEBUG=OUTPUT)
      .
      .
      .
      GO TO  4
      .
      .
      .
C$ (DEBUGGING OPTION)
C$ (DEBUGGING OPTION)
    4 CONTINUE
      .
      .
      .
      END
```

Even though a section of code may never be executed, the debugging options are processed at compile time and are effective for the remainder of the program. In the above example, the code between the GO TO statement and the CONTINUE statement may never be executed. However, debugging statements between these statements are processed at compile time and are effective for the remainder of the program, or until deactivated by a C$ OFF statement.

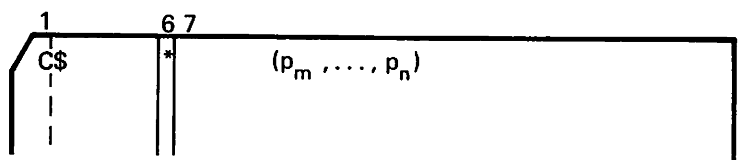
## DEBUGGING STATEMENTS




ds — Type of option, beginning after column 6: DEBUG, AREA, ARRAYS, CALLS, FUNCS, GOTOS, NOGO, OFF, STORES, TRACE

$p_i$ — Argument list; details extent of the option, ds (not used with NOGO, GOTOS; required for AREA, STORES; optional for other options)

## CONTINUATION LINE



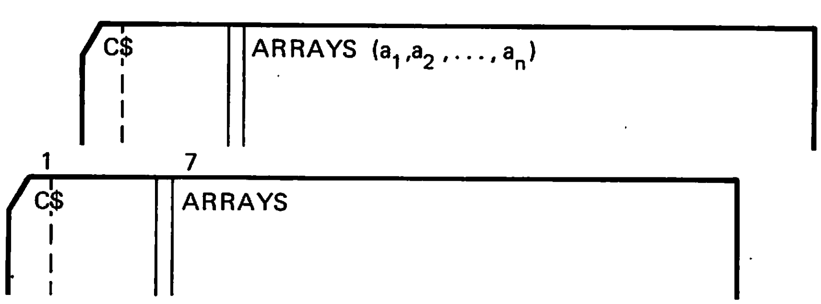

Debugging statements are written in columns 7-72, as in a normal FORTRAN statement, but columns 1 and 2 of each statement must contain the characters C$. Any character, other than a blank or zero, in column 6 denotes a continuation line. Columns 3, 4, and 5 of any debugging statement must be blank. A maximum of 19 continuation lines is allowed.

Comment lines may be interspersed with debugging statements. The statement separator ($) cannot be used with debugging statements. When the debug mode is not selected, all debugging statements are treated as comments.

Example:

```
C$    ARRAYS (A, BNUMB,Z10, C, DLIST, MATRIX,
C$   *NSUM, GTEXT,
C$   *TOTAL)
```

## ARRAYS STATEMENT




$a_1,...,a_n$ — array names

The ARRAYS statement initiates subscript checking on specified arrays. If no argument list is specified, all arrays in the program unit are checked. Each time a specified or implied element of an array is referenced, the calculated subscript is checked against the dimensioned bounds. The address is calculated according to the method described in section 1. Subscripts are not checked individually. If the address is found to be greater than the storage allocated for the array or less than one, a diagnostic is issued. The reference then is allowed to occur. Bounds checking is not performed for array references in input/output statements, or in ENCODE/DECODE statements. In a subprogram, the bounds that are checked are those in effect in the subprogram, including variable dimensions.

Example:

```
      PROGRAM ARRAYS (OUTPUT,DEBUG=OUTPUT)
      INTEGER  A(2), B(4), C(6), D(2,3,4)
      PRINT 1
    1 FORMAT(*0     ARRAYS  EXAMPLE*///)
*
*     TURN ON ARRAYS FOR ARRAYS  A  AND  D
*
C$    ARRAYS (A, D)
*
*     A(3) IS OUT OF BOUNDS  AND  ARRAYS IS ON FOR  A, SO A DIAGNOSTIC
*          IS PRINTED.
*
      A(3) = 1
*
*     B(5) IS OUT OF BOUNDS  BUT  ARRAYS IS NOT ON FOR  B, SO NO
*          DIAGNOSTIC IS PRINTED.
*
      B(5) = 1
*
      C(2) = A(A(3))
*
*     EVEN THOUGH A(3) WAS OUT OF BOUNDS, THE ASSIGNMENT TOOK PLACE.
*          A(A(3)) IS EQUIVALENT TO  A(1). THIS SUBSCRIPT IS IN BOUNDS,
*          HOWEVER THE REFERENCE TO A(3) WILL CAUSE A DIAGNOSTIC.
*
      D(-5,0,6) = 99
*
*     FOR THE ARRAY D(L,M,N) THE STORAGE ALLOCATED IS L * M * N.
*          THE SUBSCRIPT FOR THE ELEMENT D(I,J,K) IS COMPUTED AS FOLLOWS
*                              (I + L*(J-1 + M*(K-1)))
*          FOR THE ELEMENT D(-5,0,6) THE SUBSCRIPT APPEARS TO
*          BE OUT OF BOUNDS BECAUSE THE INDIVIDUAL SUBSCRIPTS ARE OUT
*          OF BOUNDS.  HOWEVER, 22, THE COMPUTED ADDRESS, IS LESS THAN
*          24, THE STORAGE ALLOCATED, AND NO DIAGNOSTIC IS ISSUED.
*
*
*     TURN ON ARRAYS FOR ALL ARRAYS
*
C$    ARRAYS
*
*     WITH THIS FORM ALL ARRAY REFERENCES WILL BE CHECKED. THERE WILL
*          BE DIAGNOSTICS FOR B(5), C(-1), AND D(0,0,0).  BECAUSE A(2)
*          IS IN BOUNDS AND A(4) IS IN AN I/O STATEMENT, THERE WILL BE
*          NO DIAGNOSTICS FOR EITHER OF THESE REFERENCES.
*
      A(2) = 1
      B(5) = 2 + C(-1)
      D(0,0,0) = 1
      PRINT 2, A(4)
    2 FORMAT(1X, A10)
      END
```

```
ARRAYS  EXAMPLE

/DEBUG/  ARRAYS  AT LINE   13- THE SUBSCRIPT VALUE OF         3 IN ARRAY A      EXCEEDS DIMENSIONED BOUND OF     2
/DEBUG/          AT LINE   20- THE SUBSCRIPT VALUE OF         7 IN ARRAY A      EXCEEDS DIMENSIONED BOUND OF     2
/DEBUG/          AT LINE   47- THE SUBSCRIPT VALUE OF         5 IN ARRAY B      EXCEEDS DIMENSIONED BOUND OF     4
/DEBUG/          AT LINE   47- THE SUBSCRIPT VALUE OF        -1 IN ARRAY C      EXCEEDS DIMENSIONED BOUND OF     6
/DEBUG/          AT LINE   68- THE SUBSCRIPT VALUE OF        -8 IN ARRAY D      EXCEEDS DIMENSIONED BOUND OF    24
```

## CALLS STATEMENT

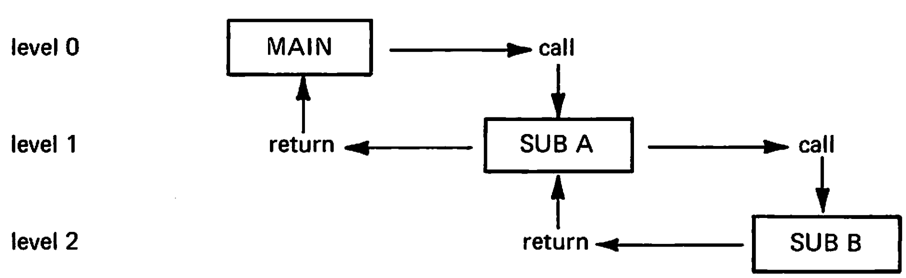
```
C$    CALLS (a1,...,an)
```

```
1     7
C$    CALLS
```

$a_1,...,a_n$ subroutine names

The CALLS statement initiates tracing of calls to and returns from specified subroutines. If there is no argument list all subroutines will be traced. Non-standard returns, specified in a RETURNS list, are included. To trace alternate entry points to a subroutine, either the entry points must be explicitly named in the argument list, or the form with no argument list must be used (all external calls traced). The message printed contains the names of the calling and called routines, as well as the line and level number of the call and return.

A main program is at level zero; a subroutine or a function called by the main program is at level 1, another subprogram called by the subprogram at level 1, is at level 2, and so forth. Calls are shown in order of ascending level number, returns in order of descending level number.




For example, subroutine SUB A is called at level 1 and a return is made to level 0. SUB B is called at level 2 and a return is made to level 1.

Example:

```
      PROGRAM CALLS(OUTPUT,DEBUG=OUTPUT)
      PRINT 1
    1 FORMAT(*0      CALLS  TRACING*)
*
*     TURN ON CALLS FOR SUBROUTINES  CALLS1  AND  CALLS2
*
C$    CALLS(CALLS1, CALLS2)
      X = 1.
      CALL CALLS1 (X,Y), RETURNS (10)
   10 IF (X .EQ. 1.) CALL CALLS2(X)
      CALL SUBNOT
      CALL CALLS1E (X,Y)
*
*     DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*         CALLS1 AND CALLS2.  SINCE THE CALLS ARE FROM THE MAIN PROGRAM,
*         THEY ARE AT LEVEL 0.  THE CALLS TO SUBNOT AND THE ALTERNATE
*         ENTRY POINT CALLS1E ARE NOT TRACED BECAUSE THEY DO NOT APPEAR
*         IN THE ARGUMENT LIST OF THE C$ CALLS STATEMENT.
*
*
*     TURN ON  CALLS FOR ALL SUBROUTINES
*
C$    CALLS
      CALL SUBNOT
      CALL CALLS2(X)
      CALL CALLS1E (X,Y)
*     DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*         SUBNOT, CALLS2, AND CALLS1E, SINCE ALL CALLS ARE TO BE
*         TRACED.
      END


      SUBROUTINE CALLS1(X,Y), RETURNS(A)
      Y = -X
      IF (Y .NE. X) RETURN A
      RETURN
      ENTRY CALLS1E
      RETURN
      END


      SUBROUTINE CALLS2(X)
      CALL CALLS1(X,Y), RETURNS(5)
    5 RETURN
      END


      SUBROUTINE SUBNOT
      X = -1.
      CALL CALLS1(X,Y), RETURNS(5)
    5 RETURN
      END
```

```
     CALLS  TRACING
/DEBUG/  CALLS    AT LINE     9- ROUTINE CALLS1  CALLED  AT LEVEL   0
/DEBUG/           AT LINE    10- ROUTINE CALLS1  RETURNS TO LEVEL   0 AT STATEMENT 10
/DEBUG/           AT LINE    10- ROUTINE CALLS2  CALLED  AT LEVEL   0
/DEBUG/           AT LINE    11- ROUTINE CALLS2  RETURNS TO LEVEL   0
/DEBUG/           AT LINE    24- ROUTINE SUBNOT  CALLED  AT LEVEL   0
/DEBUG/           AT LINE    25- ROUTINE SUBNOT  RETURNS TO LEVEL   0
/DEBUG/           AT LINE    25- ROUTINE CALLS2  CALLED  AT LEVEL   0
/DEBUG/           AT LINE    26- ROUTINE CALLS2  RETURNS TO LEVEL   0
/DEBUG/           AT LINE    26- ROUTINE CALLS1E CALLED  AT LEVEL   0
/DEBUG/           AT LINE    27- ROUTINE CALLS1E RETURNS TO LEVEL   0
```

In this example, only calls from the main program are traced. To trace calls from subprograms, a C$ CALLS statement must appear in the subprograms.

## FUNCS STATEMENT

```
1     7
C$    FUNCS (a1,...,an)
```

```
1     7
C$    FUNCS
```

If no function names ($a_1,...,a_n$) are listed, all external functions referenced in the program unit are traced. Alternate entry points must be named explicitly in the argument list, or implicitly in the C$ FUNCS statement with no paramenters.

Function tracing is similar to call tracing, but the value returned by the function is included in the debug message. Each time a specified external function is referenced, a message is printed which contains the routine name and line number containing the reference, function name and type, value returned, and level number. The level concept is the same as for the CALLS statement.

Statement function references and intrinsic function references are not traced, nor are function references in input/output statements.

Example:

The following program, VARDIM2, illustrates both the C$ FUNCS and C$ CALLS statements. All function references in the main program are traced because C$ FUNCS appears without an argument list; references to functions PVAL, AVG and MULT and the values returned to the main program (level 0) are traced. All subroutine calls in the main program are traced also because a C$ CALLS statement without an argument list appears.

Function references within the FUNCTION subprograms PVAL, AVG and MULT are traced since C$ FUNCS statements appear within these subprograms. If no C$ FUNCS statements appear in the subprograms, only main program function references will be traced.

```
      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
C     THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBPROGRAM CONCEPTS
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      PVALSF(X,Y) = PVAL(X,Y)
C$    CALLS
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
C
C     ALL EXTERNAL CALLS ARE DIAGNOSED.
C
C$    FUNCS
      AA = PVALSF(12,AVG)
      AM = PVALSF(12,MULT)
C
C     PVALSF IS A STATEMENT FUNCTION, SO THE FUNCS STATEMENT DOES NOT
C         APPLY TO IT AND NO MESSAGE IS PRINTED.  HOWEVER, THE EXTERNAL
C         FUNCTION PVAL IS REFERENCED WITHIN THE CODE FOR PVALSF,
C         AND THOSE REFERENCES ARE DIAGNOSED.
C     MULT AND AVG ARE NAMES AS ARGUMENTS TO PVALSF, HOWEVER, THE
C         FUNCTIONS ARE NOT ACTUALLY REFERENCED AND MESSAGES ARE NOT
C         PRINTED.
C
      STOP
      END
```

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
1     A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO2I=1,M
2     A(I)=A(I)+V
      RETURN
      END
```

```
      SUBROUTINE IOTA (A,M)
C     IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C          THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
1     A(I)=I
      RETURN
      END
```

```
      FUNCTION PVAL(SIZE,WAY)
C  PVAL COMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALUE IS RETURNED
C     BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED.  SIZE IS AN INTEGER
C     VALUE PASSED ON TO THE FUNCTION.
      INTEGER SIZE
C$    FUNCS(ABS)
      PVAL=ABS(WAY(SIZE))
C
C     WAY DOES NOT APPEAR IN THE ARGUMENT LIST FOR THE FUNCS STATEMENT,
C         SO ONLY THE REFERENCE TO ABS IS DIAGNOSED.
C
      RETURN
      END
```

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
      COMMON A(100)
      AVG=0.
      DO1I=1,J
1     AVG=AVG+A(I)
C$    FUNCS
C
C     ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
C
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
      REAL FUNCTION MULT(J)
C   MULT COMPUTES A STRANGE AVERAGE.  IT MULTIPLIES THE FIRST AND 12TH
C     ELEMENTS OF COMMON AND SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C     BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
C
      COMMON ARRAY(12)
C$    FUNCS
C
C     ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
C
      MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
      RETURN
      E N D
```

```
/DEBUG/  VARDIM2 AT LINE    8- ROUTINE SET      CALLED  AT LEVEL   0
/DEBUG/          AT LINE    9- ROUTINE SET      RETURNS TO LEVEL   0
/DEBUG/          AT LINE    9- ROUTINE IOTA     CALLED  AT LEVEL   0
/DEBUG/          AT LINE   10- ROUTINE IOTA     RETURNS TO LEVEL   0
/DEBUG/          AT LINE   10- ROUTINE INC      CALLED  AT LEVEL   0
/DEBUG/          AT LINE   11- ROUTINE INC      RETURNS TO LEVEL   0
/DEBUG/          AT LINE   15- REAL     FUNCTION PVAL   CALLED AT LEVEL   0
/DEBUG/          AT LINE   15- REAL     FUNCTION PVAL   RETURNS A VALUE OF   1.500000000     AT LEVEL   0
/DEBUG/          AT LINE   16- REAL     FUNCTION PVAL   CALLED AT LEVEL   0
/DEBUG/  MULT    AT LINE   11- REAL     FUNCTION AVG    CALLED AT LEVEL   2
/DEBUG/          AT LINE   11- REAL     FUNCTION AVG    RETURNS A VALUE OF  -1.500000000     AT LEVEL   2
/DEBUG/  VARDIM2 AT LINE   16- REAL     FUNCTION PVAL   RETURNS A VALUE OF  26.500000000     AT LEVEL   0
```

## STORES STATEMENT

```
 1     7
 C$    STORES (c1,c2,...,cn)
```

An argument list must be specified for the STORES statement.

($c_1$,...,$c_n$) are variable names or expressions in the forms:

variable name

variable name .relational operator. constant

variable name .relational operator. variable name

variable name .checking operator.

Relational operators are .EQ., .NE., .GT., .GE., .LT., .LE.

Checking operators are .RANGE., .INDEF., .VALID.

Example:

```
C$      STORES(SUM,DGAMP,AX,NET.LT.4,ROWSUM.RANGE.)

C$      STORES(A1,AGAIN,I,A2.EQ.5.0,IAGAIN.LE.IVAR)

C$      STORES(C.EQ.(1.,1.),L.VALID.,D.NE.10.004)

C$      STORES(G.RANGE.,TR.EQ..FALSE.)
```

The STORES statement is used to record changes in value of specified variables or arrays. The STORES statement applies only to assignment statements. Values changed as a result of input/output, or use in DATA, ASSIGN, and COMMON statements, or argument lists to subroutines and functions are not detected. The STORES statement does not apply to the index variable in a DO loop.

If the value of a variable in an EQUIVALENCE group is changed, the STORES statement will not detect changes to the value of other variables in the group.

## VARIABLE NAMES

In the first form of the STORES statement, a message is printed each time the value of a variable or an array element changes. The variable and name of the array must appear as arguments in the C$ STORES statement.

Example:

```
          PROGRAM STORES (INPUT,OUTPUT,DEBUG = OUTPUT)
          LOGICAL L1,L2
C$        STORES (NSUM,DGAMP,AX)
          NSUM = 20
          DGAMP = .5
          AX = 7.2 + DGAMP
          L1 = .TRUE.
          L2 = .FALSE.
          PLANT = 2.5
          A = 7.5
          PRINT 3
        3 FORMAT (1H0)
          STOP
          END
```

Each time the value of the variables NSUM, DGAMP and AX changes, a message is printed. The values of PLANT, A, L1 and L2 are not printed, since they do not appear in the argument list.

```
/DEBUG/  STORES  AT LINE    4- THE NEW VALUE OF THE VARIABLE NSUM    IS            20
/DEBUG/          AT LINE    5- THE NEW VALUE OF THE VARIABLE DGAMP   IS   .5000000000
/DEBUG/          AT LINE    6- THE NEW VALUE OF THE VARIABLE AX      IS   7.700000000
```

Array elements should not be specified in the parameter list of a STORES statement; the array name must be used. If an array element appears, an informative diagnostic is printed. Changes to any element of the array are noted; only the array name without subscript is listed.

Example:

```
      PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
      REAL A(10), B(4,2)
C$    STORES (A,B)
      B(1,2) = 5.5
      B(4,2) = 0.
      DO 4 N = 1,3
    4 A(N) = N+1
      PRINT 5
    5 FORMAT (1H0)
      STOP
      END
```

```
/DEBUG/  STORAR  AT LINE    4- THE NEW VALUE OF THE VARIABLE B       IS   5.500000000
/DEBUG/          AT LINE    5- THE NEW VALUE OF THE VARIABLE B       IS  0.
/DEBUG/          AT LINE    7- THE NEW VALUE OF THE VARIABLE A       IS   2.000000000
/DEBUG/          AT LINE    7- THE NEW VALUE OF THE VARIABLE A       IS   3.000000000
/DEBUG/          AT LINE    7- THE NEW VALUE OF THE VARIABLE A       IS   4.000000000
```

The values stored into array elements B(1,2) and B(4,2) appear in the debug output under the array name B in both cases, and array elements A(1), A(2), and A(3) appear under the array name A.

## RELATIONAL OPERATORS

In the second form of the C$ STORES statement, a message is printed only when the stored value satisfies the relation specified in the argument list. The two components of the relational expression must be of the same type.

```
      PROGRAM ST3 (INPUT,OUTPUT,DEBUG=OUTPUT)
    5 FORMAT (1H0)
      PRINT 5
      M = 5
C$    STORES (I.EQ.3,N.LE.M,ANT)
      I = 3
      I = 4
      N = 4
      N = 6
      J = 10
      ANT = 77.0
      END
```

```
/DEBUG/  ST3     AT LINE    6- THE NEW VALUE OF THE VARIABLE I     IS           3
/DEBUG/          AT LINE    8- THE NEW VALUE OF THE VARIABLE N     IS           4
/DEBUG/          AT LINE   11- THE NEW VALUE OF THE VARIABLE ANT   IS 77.00000000
```

I appears in the debug output when it is equal to 3; N appears when it is less than or equal to M. Since no relational operator is specified with ANT, it is printed whenever the value changes.

## CHECKING OPERATORS

In the third form of the STORES statement, a message is issued only when the stored value is out of range, indefinite, or invalid as specified by the checking operator.

| | |
|---|---|
| RANGE | Out of range |
| INDEF | Indefinite |
| VALID | Out of range or indefinite |

For example:

```
C$    STORES (ROWSUM .RANGE., COLSUM .VALID.)
```

Whenever the value to be stored into ROWSUM is out of range, a message is printed. Whenever the value to be stored into COLSUM is out of range or indefinite, a message is printed.

## HOLLERITH DATA

Hollerith data stored in a variable of type integer is interpreted by the STORES statement as an integer number. Hollerith data stored in a variable of type real or double precision is interpreted as a real or double precision number.

In the following example, the three integer variables IHOLL, IRIGHT and ILEFT contain the characters PA in display code (20 and 01).

```
IHOLL     20015555555555555555

          P A  blank fill

IRIGHT    00000000000000002001

          zero fill       P A

ILEFT     20010000000000000000

          P A zero fill
```

Example:

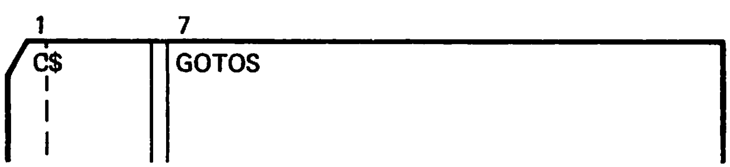
```
                PROGRAM DEHOL (INPUT,OUTPUT,DEBUG=OUTPUT)
          C$    DEBUG
          C$    STORES(IHOLL,IRIGHT,ILEFT,HOLL)
                IHOLL=2HPA
                IRIGHT=2RPA
                ILEFT=2LPA
                HOLL=2HPA
                PRINT 1
              1 FORMAT (1HO)
                STOP
                END
```

```
/DEBUG/  DEHOL   AT LINE     6- THE NEW VALUE OF THE VARIABLE IHOLL   IS **************
/DEBUG/          AT LINE     7- THE NEW VALUE OF THE VARIABLE IRIGHT  IS           1025
/DEBUG/          AT LINE     8- THE NEW VALUE OF THE VARIABLE ILEFT   IS **************
/DEBUG/          AT LINE     9- THE NEW VALUE OF THE VARIABLE HOLL    IS   .4021071096E+15
```

The variables IHOLL, IRIGHT, and ILEFT are interpreted as integer numbers. Since the field width allocated by the STORES option (14 digits) is insufficient to contain the converted quantities represented by IHOLL and ILEFT, these fields are filled with asterisks. The variable IRIGHT is converted and printed out by the STORES option as 1025.

The variable HOLL is interpreted as a real number, and its value is printed out.

## GOTOS STATEMENT




No argument list can be specified with the C$ GOTOS statement. The GOTOS statement initiates checking of all assigned GO TO statements to ensure that the statement label assigned to the integer variables is in the GO TO statement list. If no match is found, a message is printed and transfer of control continues.

```
          PROGRAM GO TOS (OUTPUT,DEBUG=OUTPUT)
          INTEGER A
    C$    GOTOS
    *         (GOTOS NEVER USES AN ARGUMENT LIST)
    *
          ASSIGN 1 TO A
          GO TO A (1, 2, 3)
    *
    *     IN THIS CASE NO MESSAGE IS PRINTED SINCE THE LABEL ASSIGNED TO
    *         A IS IN THE GOTO LIST.
    *
        4 PRINT 10
       10 FORMAT(*   --CONTROL TRANSFERED TO STATEMENT LABEL 4--*)
          STOP
        1 ASSIGN 4 TO A
          GO TO A (1, 2, 3)
    *
    *     IN THIS CASE A MESSAGE IS PRINTED SINCE THE LABEL 4 IS NOT IN
    *         THE GOTO LIST.  CONTROL THEN TRANSFERS TO LABEL 4.
    *
        2 CONTINUE
        3 CONTINUE
          END
```

```
/DEBUG/  GOTOS   AT LINE   16- ASSIGNED GOTO INDEX CONTAINS THE ADDRESS 002151. NO MATCH FOUND IN STATEMENT LABEL ADDRESS LIST
  --CONTROL TRANSFERED TO STATEMENT LABEL 4--
```

# TRACE STATEMENT

```
C$     TRACE (lv)

1      7
C$     TRACE
```

lv is a level number 0-49. If lv = 0, tracing occurs only outside DO loops. If lv = n, tracing occurs up to and including level n in a DO nest. If no level is specified, tracing occurs only outside DO loops.

The C$ TRACE statement traces the following transfers of control within a program unit:

GO TO

Computed GO TO

Assigned GO TO

Arithmetic IF

True side of logical IF

Transfers resulting from a return specified in a RETURNS list are not traced. (These can be checked by the C$ CALLS statement.)

If an out-of-bound computed GO TO is executed, the value of the incorrect index is printed before the job is terminated.

Messages are printed each time control transfers during execution. The message contains the routine name, the line where the transfer took place, and the number of the line to which the transfer was made, as well as the statement number of this line, if present.

A message is printed each time control transfers at a level less than or equal to the one specified by lv. For example, if a statement C$ TRACE(2) appears before a sequence of DO loops nested four deep, tracing takes place in the two outermost loops only.

TRACE messages are produced at execution time, but TRACE levels are assigned at compile time; therefore, the compile time environment determines the tracing status of any given statement. For example, a DO loop TRACE statement applies only to control transfers occurring between the DO statement and its terminal statement at compile time (physically between the two in the source listing).

Example:

```
                        PROGRAM P(OUTPUT,DEBUG=OUTPUT)
                        DATA J/0/
     level 0      C$    TRACE(1)
                        IF (J .EQ. 0)  GO TO 11
5    level 1         11 DO 1 I1  =  1, 3
                        IF ( (J+1) .EQ. I1 )  GO TO 12
                     12 J = 1
       level 2          DO 2 I2 = 1, 5
                        J = J + I2
10                      GO TO 2
                      2 CONTINUE
                  C$    TRACE(3)
       level 2          DO 20 I2 = 1, 3
                        IF ( I2 .EQ. 3 )  GO TO 20
15                      J = 2
         level 3        DO 3 I3 = 1, 4
                        IF ( J .GT. I3 )  GO TO 31
                     31 DO 4 I4 = 1, 2
           level 4      GO TO 4
20                    4 CONTINUE
                      3 CONTINUE
                     20 CONTINUE
                        J = 0
                      1 CONTINUE
25                      END
```

```
/DEBUG/  P   AT LINE   4- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE   4- CONTROL WILL BE TRANSFERRED TO STATEMENT 11    AT LINE    5
/DEBUG/      AT LINE   6- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE   6- CONTROL WILL BE TRANSFERRED TO STATEMENT 12    AT LINE    7
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31    AT LINE   18
/DEBUG/      AT LINE  14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/      AT LINE  14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20    AT LINE   22
```

In the first level 2 loop no debug messages are printed since the TRACE(1) statement is in effect. However, when the TRACE(3) statement becomes effective, flow is traced up to and including level 3. There are no messages for transfers within the level 4 loop. To trace only inner loops, for example levels 3 and 4 in the above example, a C$ TRACE(4) statement is placed immediately before the DO statement for the level 3 loop (line 16). A C$ OFF (TRACE) statement is placed after the terminal line for the level 3 loop, so that subsequent program flow in levels 0, 1, and 2 is not traced.

The level number applies to the entire program unit; it is not relative to the position of the C$ TRACE statement in the program. For example, to trace the level 4 DO loop in Program P:

```
     C$   TRACE(4)
```

must be specified. Positioning the statement C$ TRACE(1) before statement 31 would not achieve the same result.

Care must be taken with the use of debugging statements within DO loops. Since nested loops are executed more frequently, the quantity of debug output may quickly multiply.

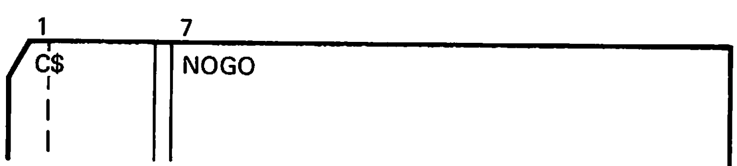
The C$ TRACE (lv) statement traces transfers of control within DO loops; however, transfers between the terminal statement and the DO statement are not traced.

Example:

```
          DO 100 I = 1,10
          .
          .
          .
     100 CONTINUE
```

Transfers from statement 100 to the DO statement are not traced.

## NOGO STATEMENT




No argument list is specified with this statement. The NOGO statement suppresses partial execution of a program containing compilation errors.

If a NOGO statement is present anywhere in the program, it applies to the entire program. It is therefore not affected by an OFF statement or by bounds in an AREA statement.

## DEBUG DECK STRUCTURE

Debugging statements may be interspersed with FORTRAN statements in a program unit (main program, subroutine, function). The debugging statements apply to the program unit in which they appear. Interspersed debugging statements (figure 9-1) change the FORTRAN generated line numbers for a program.

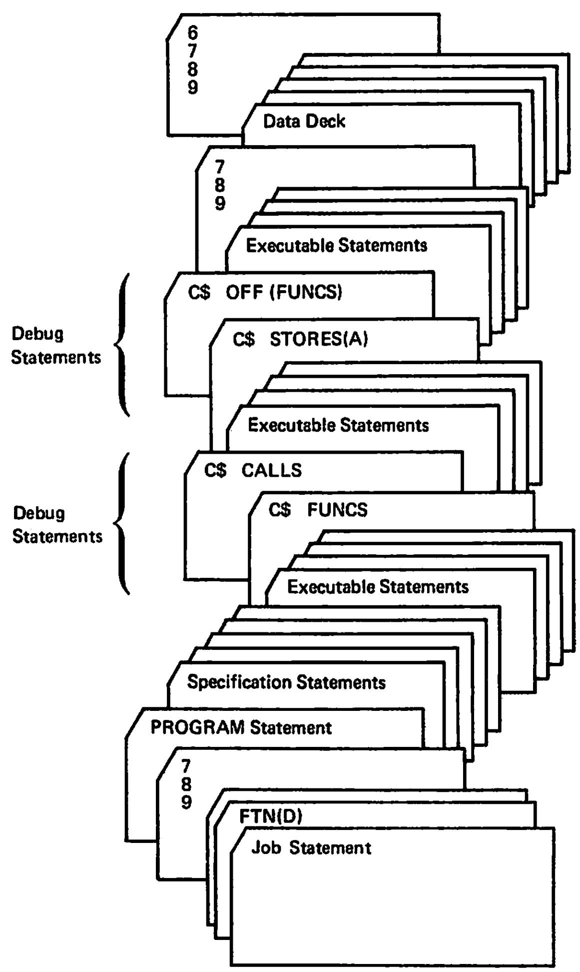
Debugging statements also may be grouped to form a debugging deck in one of the following ways:

As a deck placed immediately after the PROGRAM, SUBROUTINE or FUNCTION statement heading the routine to which the deck applies (internal debugging deck, figure 9-3). Any names specified in the DEBUG statement, other than the name of the enclosing routine, are ignored.

As a deck immediately preceding the first source deck in the source input file (external debugging deck, figure 9-2).

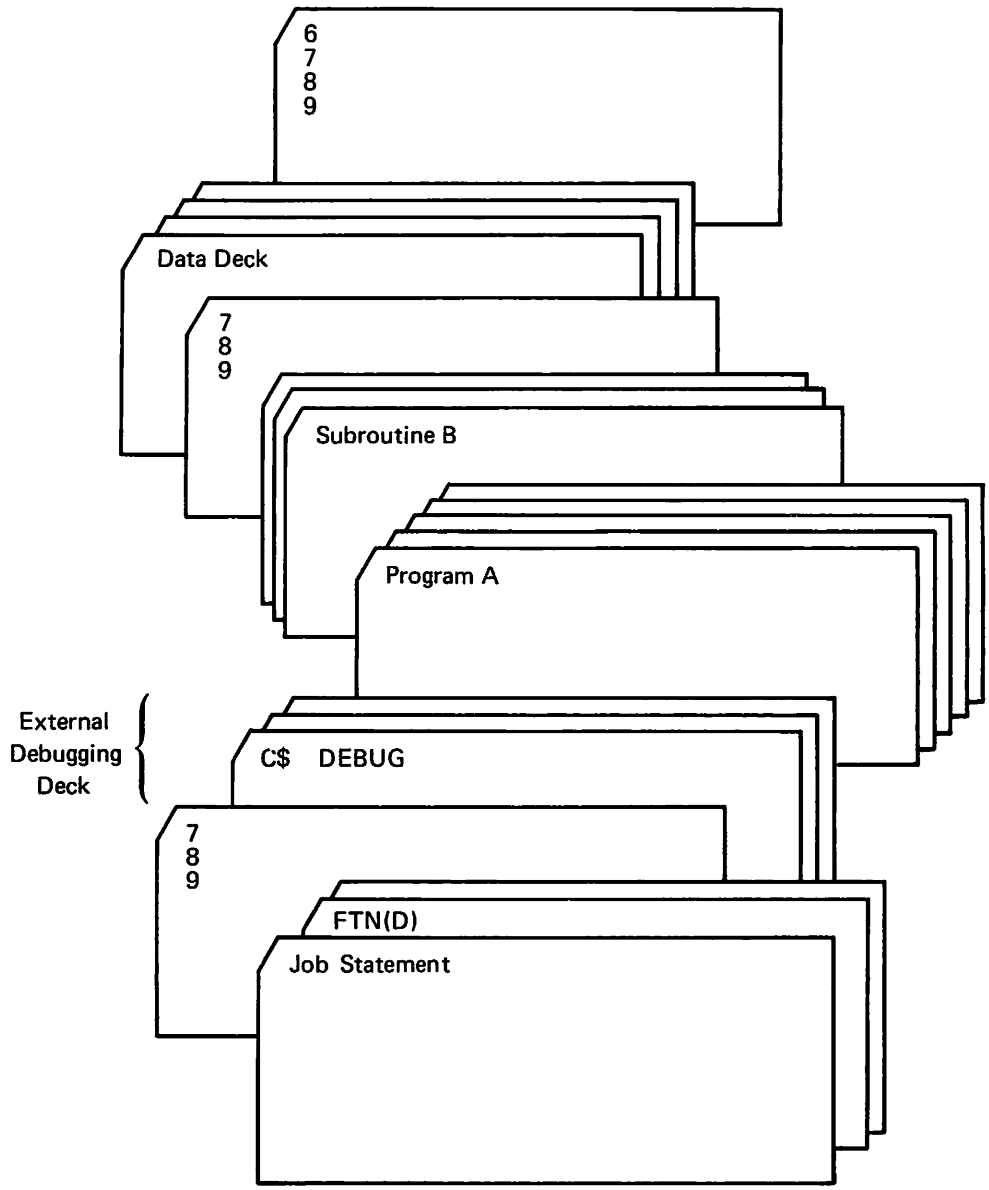
As one or more decks on the file specified by the D parameter on the FTN control statement (external debugging deck, figure 9-4). When no name is specified by the D parameter, the INPUT file is assumed.

All debugging decks must be headed by a C$ DEBUG statement. In an internal debugging deck, the C$ DEBUG statement is used without an argument list, since the deck can only appear to the routine in which it is inserted. In an external debugging deck, a C$ DEBUG may be used with or without an argument list. The statements in the external debugging deck apply to all program units in the compilation.



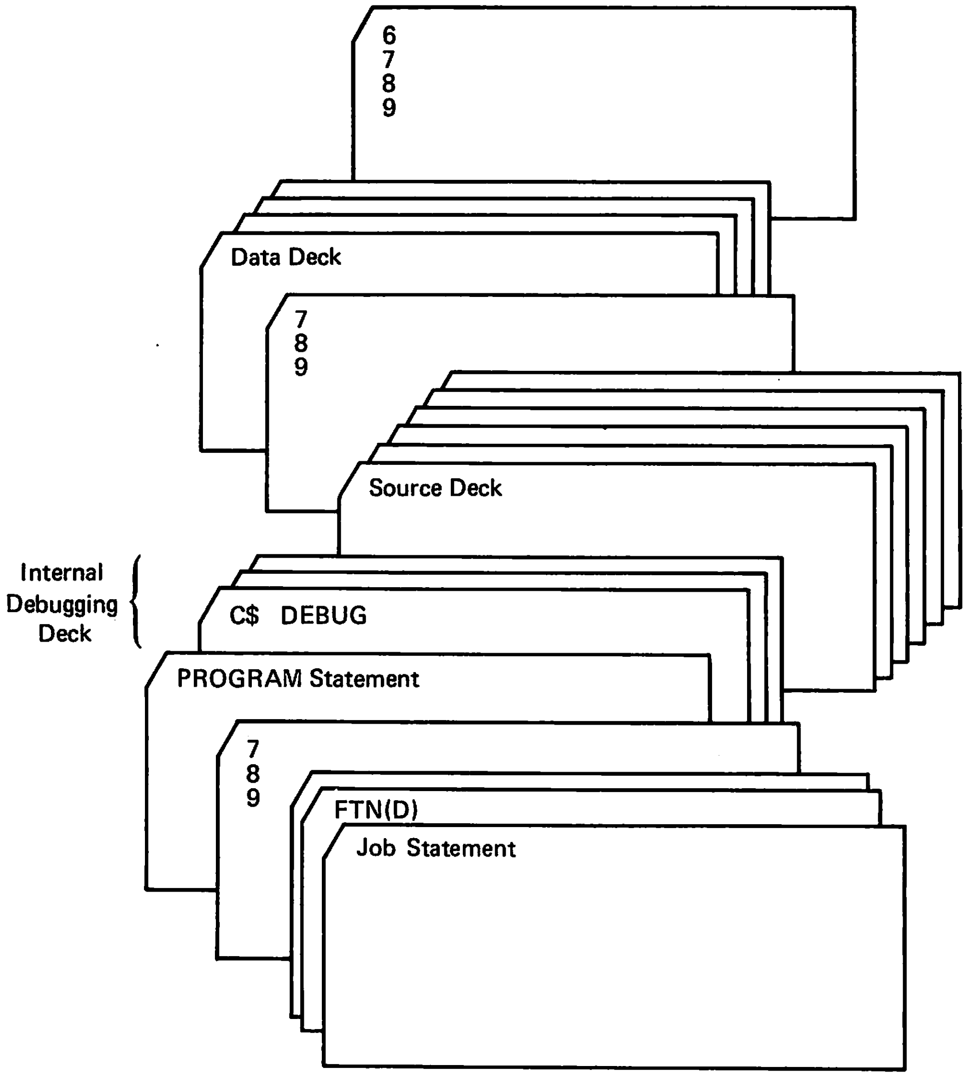

Debugging statements are interspersed; they are inserted at the point in the program where they will be activated.

Figure 9-1. Example of Interspersed Debugging Statements



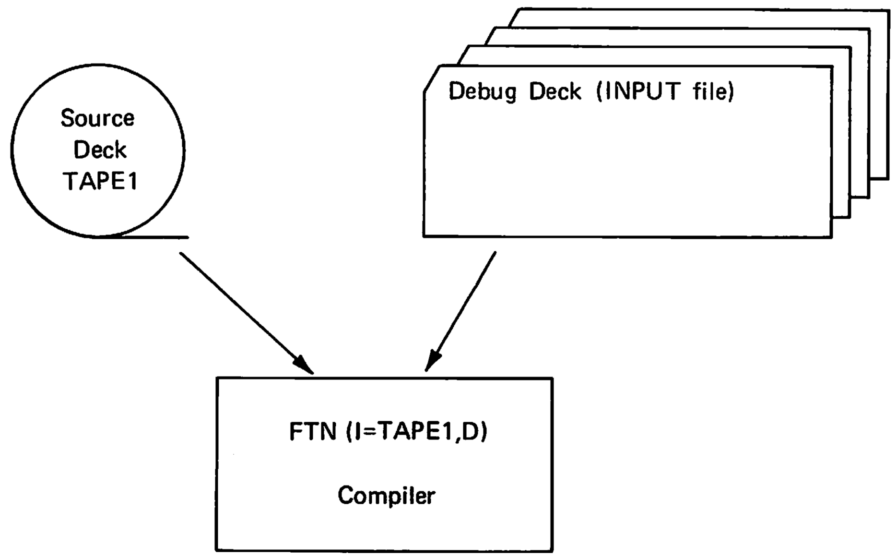

The external debugging deck is placed immediately in front of the first source line. All program units (here, Program A and Subroutine B) will be debugged (unless limiting bounds are specified in the deck). This positioning is particularly useful when a program is to be run for the first time, since it ensures that all program units will be debugged.

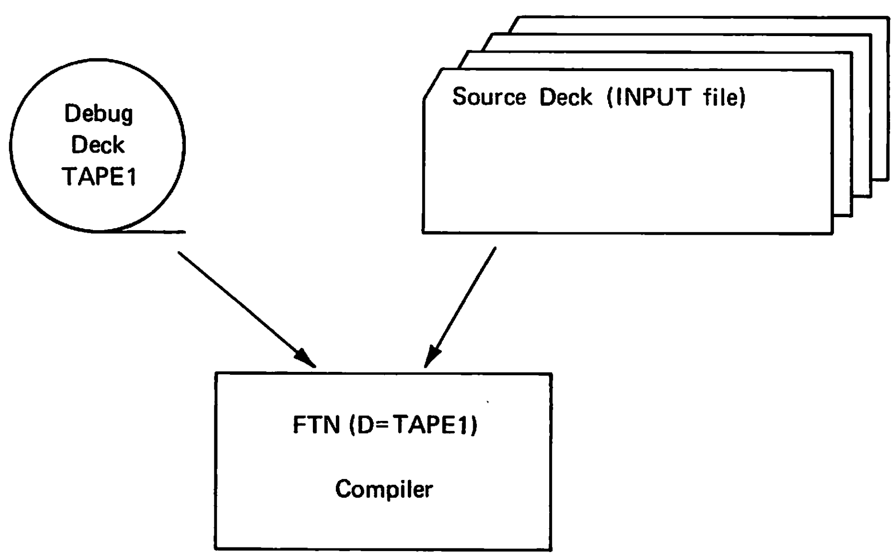
Figure 9-2. External Debugging Deck




When the debugging deck is placed immediately after the PROGRAM statement and before any specification statements, all statements in the program unit will be debugged (unless limiting bounds are specified in the deck); no statements in other program units will be debugged. This positioning is best when the job is composed of several program units known to be free of bugs and one unit that is new or known to have bugs.

Figure 9-3. Example of Internal Debugging Deck




The debugging deck is placed on a separate file (external debugging deck) named by the D parameter on the FTN control statement and called in during compilation. All program units will be debugged (unless the program units to be debugged are specified in the deck). This positioning is useful when several jobs can be processed using the same debugging deck.

Figure 9-4. Example of External Deck on Separate File

## DEBUG STATEMENT

```
1     7
C$    DEBUG
```

```
1     7
C$    DEBUG (name1 ,..., namen)
```

$name_1,...,name_n$ routines to which the debugging deck applies

Internal and external debugging decks start with a DEBUG statement and end with the first line other than a debugging statement or comment. Interspersed debugging statements do not require a DEBUG statement.

In an internal debugging deck, the first form of the statement (without an argument list) is generally used, since the deck can apply only to the program unit in which it appears. If a name is specified it must be the name of the routine containing the debugging deck; if any other name is specified, an informative diagnostic is printed.

In an external debugging deck, if no names are specified, the deck applies to all routines compiled. Otherwise, it will apply to only those program units specified by $name_1,...,name_n$; if any other name is specified, an informative diagnostic is printed.

Example:

In the following program, a DEBUG statement is not required since the debugging statement, C$ STORES (A,B), is interspersed.

```
      PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
      REAL A(10), B(4,2)
C$    STORES (A,B)
      B(1,2) = 5.5
      B(4,2) = 0.
      DO 4 N = 1,3
    4 A(N) = N+1
      PRINT 5
    5 FORMAT (1H0)
      STOP
      END
```

However, if the C$ STORES statement immediately follows the PROGRAM statement, this is an internal debugging deck, and a C$ DEBUG statement must appear.

```
      PROGRAM DEHOL (INPUT,OUTPUT,DEBUG=OUTPUT)
C$    DEBUG
C$    STORES(IHOL,IRIGHT,ILEFT,HOLL)
      IHOL=2HPA
      IRIGHT=2RPA
      ILEFT=2LPA
      HOLL=2HPA
      PRINT 1
    1 FORMAT (1H0)
      STOP
      END
```

There can be several DEBUG statements in an external deck, and a routine can be mentioned more than once.

```
C$    DEBUG
C$    STORES(I,J)
C$    DEBUG(MAIN,EXTRA,NAMES)
C$    ARRAYS(VECTAB,MLTAB)
C$    DEBUG(MAIN)
C$    TRACE
C$    CALLS(EXTRA,NAMES)
```

## AREA STATEMENT

| 1 | 7 |
|---|---|
| C$ | AREA bounds$_1$ ,. . . ,bounds$_n$ |

| 1 | 7 |
|---|---|
| C$ | AREA/name$_1$/bounds$_1$ ,. . . , bounds$_n$ ,. . . /name$_n$/bounds$_1$ ,. . . ,bounds$_n$ |

C$ AREA(bounds$_1$,...,bounds$_n$) is used in internal debugging decks only.

name$_1$,name$_2$,...,name$_n$ are the names of routines to which the bounds apply.

bounds are line positions defining the area to be debugged.

bounds can be written in one of the following forms:

| | | |
|---|---|---|
| $(n_1,n_2)$ | $n_1$ | Initial line position. |
| | $n_2$ | Terminal line position. |
| $(n_3)$ | $n_3$ | Single line position to be debugged. |
| $(n_1,*)$ | $n_1$ | Initial line position. |
| | * | Last line of program. |
| $(*,n_2)$ | * | First line of program. |
| | $n_2$ | Terminal line position. |
| (*,*) | * | First line of program. |
| | * | Last line of program. |

Line positions can be:

| | |
|---|---|
| nnnnn | Statement label. |
| Lnnnn | Source program line number as printed on the source listing by the FORTRAN Extended compiler (source listing line numbers change when debugging statements are interspersed in the program). |
| id.n | UPDATE line identifier (defined in the UPDATE Reference Manual); id must begin with an alphabetic character and contain no special characters. |

A comma must be used to separate the line positions, and embedded blanks are not permitted. Any of the line position forms can be combined and bounds can overlap.

The AREA statement is used to specify an area to be debugged within a program unit. All debugging statements applicable to the program areas designated by the AREA statement must follow that statement. Each AREA statement cancels the preceding program AREA statement. An AREA statement (or contiguous set of AREA statements) specifies bounds for all debugging statements that occur between it and the next C$ DEBUG, AREA statement, or FORTRAN source statement.

AREA statements may appear only in an external or an internal debugging deck (figures 9-2, 9-3, and 9-4). If they are interspersed in a FORTRAN source deck, they will be ignored.

In an external debugging deck, the following form, with /name$_i$/ specified, must be used. It can be used with both forms of the DEBUG statement.

```
1     7
C$    DEBUG
1     7
C$    AREA/name1/bounds1,...,boundsn,... /namen/bounds1 ,...,boundsn
```

or

```
1     7
C$    DEBUG (name1,...,namen)
1     7
C$    AREA/name1/bounds1 ,...,boundsn,... /namen/bounds1 ,...,boundsn
```

If /name$_i$/ is omitted, or names in the /name$_i$/ list do not appear in (name$_1$,...,name$_n$) in the DEBUG statement, the AREA statement is ignored.

In an internal debugging deck, the following form is used, and the bounds apply to the program unit that contains the deck.

```
1     7
C$    DEBUG
1     7
C$    AREA bounds1,...,boundsn
```

Example:

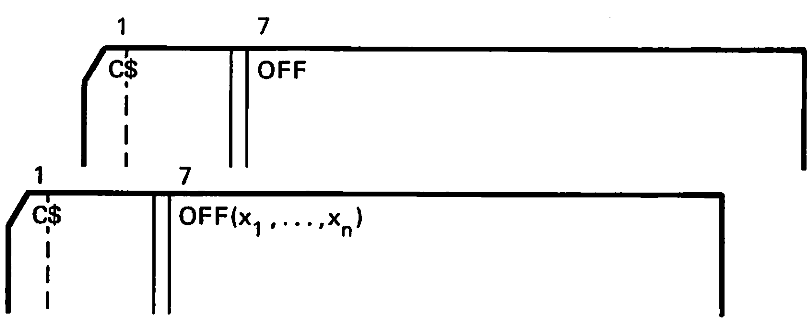
External deck

```
C$   DEBUG
C$   AREA/PROGA/(XNEW.10,XNEW.30)/SUB/*,L50)
C$   ARRAYS (TAB,TITLE,DAYS)
C$   AREA/SUB/(15,99)
C$   STORES (DAYS)
```

Internal deck

```
C$   DEBUG
C$   AREA (L10,*)
C$   FUNCS (ABS)
```

## OFF STATEMENT




$x_1,...,x_n$ debug options

The OFF statement deactivates the options specified by $x_i$ or all currently active options except NOGO, if no argument list exists. Only options activated by interspersed debugging statements are affected. Options activated in debug decks or by subsequent debugging statements are not affected.

The OFF statement is effective at compile time only. In a debugging deck, the OFF statement is ignored.

Example:

```
      PROGRAM OFF (OUTPUT,DEBUG=OUTPUT)
C$    DEBUG
C$    STORES(C)
      INTEGER A, B, C
C$    STORES(A, B)

      A = 1
      B = 2
      C = 3
*
*     MESSAGES WILL BE PRINTED FOR STORES INTO A, B, AND C.
*
C$    OFF
*
      A = 4
      B = 5
      C = 6
*     THE OFF STATEMENT WILL ONLY AFFECT THE INTERSPERSED DEBUGGING
*         STATEMENT, SO THERE WILL BE NO MESSAGES FOR STORES INTO
*         A OR B.  HOWEVER, C$  STORES(C) IN THE DEBUGGING DECK IS NOT
*         AFFECTED, AND A MESSAGE IS PRINTED FOR A STORE INTO C.
*
      END
```

```
/DEBUG/  OFF    AT LINE    7- THE NEW VALUE OF THE VARIABLE A      IS          1
/DEBUG/         AT LINE    8- THE NEW VALUE OF THE VARIABLE B      IS          2
/DEBUG/         AT LINE    9- THE NEW VALUE OF THE VARIABLE C      IS          3
/DEBUG/         AT LINE   17- THE NEW VALUE OF THE VARIABLE C      IS          6
```

## PRINTING DEBUG OUTPUT

Debug messages produced by the object routines are written to a file named DEBUG. The file is printed upon job termination, unless otherwise specified by the user, because it has a print disposition. To intersperse debugging information with output, the programmer should equate DEBUG to OUTPUT on the PROGRAM statement. An FET and buffer are supplied automatically at load time if the programmer does not declare the DEBUG file in the PROGRAM statement. For overlay jobs, the buffer and FET will be placed in the lowest level of overlay containing debugging. If this overlay level would be overwritten by a subsequent overlay load, the debug buffer will be cleared before it is overwritten.

At object time, printing is performed by seven debug routines. These routines are called by code generated at compile time when debugging is selected.

| Routine | Function |
|---|---|
| BUGARR | Checks array subscripts |
| BUGCLL | Prints messages when subroutines are called and when return to calling program occurs |
| BUGFUN | Prints messages when functions are called and when return to calling program occurs |
| BUGGTA | Prints a message if the target of an assigned GO TO is not in the list |
| BUGSTO | Performs stores checking |
| BUGTRC | Flow trace printing except for true sides of logical IF |
| BUGTRT | Flow trace printing for true sides of logical IF |

## STRACE ENTRY POINT

Traceback information from a current subroutine level back to the main level is available through a call to STRACE. STRACE is an entry point in the object routine BUGCLL. A program need not specify the D option on the FTN control statement to use the STRACE feature.

STRACE output is written on the file DEBUG; to obtain traceback information interspersed with the source program's output, DEBUG should be equivalenced to OUTPUT in the PROGRAM statement.

Examples:

```
PROGRAM MAIN (OUTPUT,DEBUG=OUTPUT)
CALL SUB1
END

SUBROUTINE SUB1
CALL SUB2
RETURN
END

SUBROUTINE SUB2
I = FUNC1(2)
RETURN
END

FUNCTION FUNC1 (K)
FUNC1 = K ** 10
CALL STRACE
RETURN
END
```

Output from STRACE:

```
/DEBUG/   FUNC1    AT LINE     3- TRACE ROUTINE CALLED
                                  FUNC1    CALLED BY SUB2    AT LINE    2, FROM    1 LEVELS BACK
                                  SUB2     CALLED BY SUB1    AT LINE    2, FROM    2 LEVELS BACK
                                  SUB1     CALLED BY MAIN    AT LINE    2, FROM    3 LEVELS BACK
```

A main program is at level 0; a subroutine or function called by the main program is at level 1; another subprogram called by a subprogram is at level 2, etc. Calls are shown in order of ascending level number, returns in order of descending level number.

For additional information regarding the debugging facility, refer to the FORTRAN Extended Debug User's Guide.

# FTN CONTROL STATEMENT 10

The FORTRAN Extended compiler is called from the library and executed by an FTN or FTN4† control statement. Either control statement calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. Either control statement may be used in any of the following forms:

```
FTN(p1,p2,...,pn) comments

FTN. comments

FTN,p1,...,pn. comments
```

Examples:

```
FTN (A,L,R,GO,S=0)
FTN4 (A,L,R,GO,S=0)
```

## PARAMETERS

The optional parameters, $p_1,...,p_n$ must be separated by commas and may be in any order. If no parameters are specified, FTN 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.

Default values are used for omitted parameters. These defaults are set when the system is installed; since installations can change default values, the user should determine what default values are in effect at the user's particular installation.

Unrecognizable parameters are ignored. 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 A, B, D, G, I, L, ML, P, S, and X parameters are passed to COMPASS when intermixed COMPASS subprograms are present.

†Not valid on SCOPE 2.

In the following description of the FTN control statement parameters, lfn indicates a file name consisting of one to seven letters and digits, the first a letter. Two or more options using the same file terminates compilation with a message to the dayfile.

## A EXIT PARAMETER (Default: A = 0)

| | |
|---|---|
| A | If fatal errors have occurred during compilation, the system aborts the job to the next EXIT(S) control statement (NOS/BE 1 and SCOPE 2) or EXIT control statement (NOS 1). If no such control statement is found, the job is terminated. This option has no effect on interactive jobs. A takes precedence over GO but not over D. |
| A = 0 | System advances to the next control statement at end of compilation whether or not fatal errors have been found. |

## B BINARY OBJECT FILE (Default: B = LGO)

| | |
|---|---|
| B | Generated binary object code is output on file LGO. |
| B = lfn | Generated binary object code is output on file lfn. |
| B = 0 | No binary object file is produced. Cannot be specified with GO. |

The B option conflicts with the Q and E options.

## BL BURSTABLE LISTING (Default: BL = 0)

| | |
|---|---|
| BL | Generates output listing that is easily separable into components by issuing page ejects between source code, error summary (if present), cross reference map, and object code (if requested); and ensures that each program unit listing contains an even number of pages (page parity) issuing a blank page at the end if necessary. |
| BL = 0 | Generates listings in compact format. |

## C COMPASS ASSEMBLY (Default: C = 0)

| | |
|---|---|
| C | Selects the COMPASS assembler to process the symbolic object code generated by FORTRAN Extended. When the C parameter is specified, FTNMAC is selected as a system text for the COMPASS assembly; therefore, if the C option is selected, the maximum number of system texts that can be specified with the G and S parameters is six. |
| C = 0 | Selects the FORTRAN Extended internal assembler (regardless of installation default), which is two to three times faster than the COMPASS assembler. |

The C option conflicts with the TS, Q, and E options.

## CC CONTROL STATEMENT CONTINUATION PARAMETER (Default: CC = 0)

| | |
|---|---|
| CC | Causes the FORTRAN Extended compiler to interpret the following control statement as a continuation of the FTN control statement, thus allowing the FTN control statement to be continued on more than one line. The CC parameter must be repeated on each statement in the sequence with the exception of the last statement in the sequence; the CC parameter must not appear on the last statement in the sequence. Each statement in the sequence of continued statements must be terminated by a period or a right parenthesis. |
| CC = 0 | The FTN control statement appears on one line only. |

Example:

```
FTN,I=INPUT,CC.
L=OUTPUT,CC.
B=LGO.
```

## D DEBUGGING MODE PARAMETER (Section 9) (Default: D = 0)

| | |
|---|---|
| D = lfn | This option must be specified if the debug utility described in section 9 is to be used. lfn is the name of the file where the user debug deck resides (see figure 9-4, section 9). Binary object code is generated on the file indicated by the B parameter regardless of compilation errors or the exit parameter A. Interspersed COMPASS code, if present, is assembled under the COMPASS D option. Specifying D automatically activates OPT=0 and the T option; thus, FTN(D) is equivalent to FTN(D,OPT=0,T,A=0). |
| D | Implies D = INPUT |
| D = 0 | Debug statements are ignored. |

OPT=1 and OPT=2 are ignored if D or D=lfn is specified. The D option conflicts with the TS option.

## DB CYBER INTERACTIVE DEBUG PARAMETER (Default: DB = 0)

| | |
|---|---|
| DB = ID | This option must be specified if the program is to be debugged using CYBER Interactive Debug and the DEBUG control statement (NOS 1 and NOS/BE 1 only) has not been included. If the DB parameter is specified, the binary object code is complemented by a line number table and a symbol table. CYBER Interactive Debug uses these tables while processing the user's program to determine variable locations, source line locations, and other useful debugging information. |
| DB = 0 | No debug tables are generated. If CYBER Interactive Debug has been turned on with the DEBUG control statement, specifying DB = 0 turns it off for the duration of the compilation. |
| DB | Implies DB = ID. |

Specifying the DB option automatically activates the TS option. The DB option conflicts with the D and OPT = 0, 1, or 2 options. For more information, refer to the CYBER Interactive Debug reference manual.

## E EDITING PARAMETER (Default: E = 0)

| | |
|---|---|
| E = lfn | Generated object code is output as COMPASS line images on the file lfn, which is rewound at the end of compilation. Each program unit is prefaced with the line image, *DECK,program, so that the file will be suitably formatted for input to UPDATE or MODIFY. Binary object code is not produced, and COMPASS is not called. When the file lfn is assembled subsequently, S=FTNMAC must be specified on the COMPASS control statement. |
| E | Implies E = COMPS |
| E = 0 | Object file is generated in normal binary code rather than as COMPASS line images. |

The E option conflicts with the B, C, GO, OL, TS, and Q options.

## EL ERROR LEVEL (Appendix B)

(Default: EL = I)

| | |
|---|---|
| EL = A | Lists diagnostics indicating all non-ANSI usages, as well as fatal diagnostics; lists informative diagnostics if compiling under OPT = 0, 1, or 2; lists note and warning diagnostics if compiling in TS mode. |
| EL = I | Lists informative and fatal diagnostics if compiling under OPT = 0, 1, or 2; lists note, warning, and fatal diagnostics if compiling in TS mode. |
| EL = N | Lists note, warning, and fatal diagnostics if compiling in TS mode; lists fatal diagnostics if compiling under OPT = 0, 1, or 2. |
| EL = W | Lists warning and fatal diagnostics if compiling in TS mode; lists fatal diagnostics if compiling under OPT = 0, 1, or 2. |
| EL = F | Lists fatal diagnostics. |

## ER ERROR RECOVERY

(Default: ER if in TS or OPT=0 mode
ER=0 if in OPT=1 or 2 mode)

| | |
|---|---|
| ER | Code is generated for object time reprieve. When this option is selected, any of the following execution time errors are reprieved: arithmetic mode error, bad system request in RA + 1, CP or IO time limit exceeded, mass storage limit exceeded, or an operator drop. When the error occurs within the field length occupied by the user program, the name of the program unit and number of the line in which the error occurred are written to the job dayfile and (under NOS 1 only) the OUTPUT file. (Under OPT=1 or 2, the line number might be approximate, since optimization can rearrange portions of the code.) When the error occurs outside the user's field length, only the P-register contents are shown. This option increases the size of object code and should be used only while a program is being debugged. |
| ER=0 | No code is generated for object time reprieve. |

## G GET SYSTEM TEXT FILE

(Default: G = 0)

| | |
|---|---|
| G = lfn | Loads the first system text overlay from the sequential binary file, lfn. |
| G = lfn/ovl | Searches the sequential binary file, lfn, for a system text overlay with the name ovl and loads the first such overlay encountered. |
| G | Implies G = SYSTEXT |
| G = 0 | Prevents system text loading from sequential binary file. |

A maximum of seven system texts can be specified by any combination of the G, S, and C parameters.

This feature is for COMPASS subprograms only.

## GO AUTOMATIC EXECUTION (LOAD AND GO)

(Default: GO = 0)

| | |
|---|---|
| GO | Binary object file (B option) is loaded and executed at end of compilation; file is not rewound before compilation. |

GO = 0 — Binary object file is not loaded and executed.

The GO option conflicts with the Q, E, and B = 0 options.

## I SOURCE INPUT FILE (Default: I = INPUT)

I = lfn — Source code to be compiled appears on file lfn. Compilation ends when an end of section, end of partition, or end of information is encountered.

I — Implies I = COMPILE

## L LIST OUTPUT FILE (SECTION 12) (Default: L = OUTPUT)

L = lfn — Listable output (specified by list control options BL, EL, OL, R, and SL) is to be written onto file lfn. If list control options are not specified, the listing consists of the source program, informative and fatal diagnostics, and a short reference map.

L — Implies L = OUTPUT

L = 0 — Fatal diagnostics and the statement that caused them are listed on the file OUTPUT. All other compile-time output, including intermixed COMPASS, is suppressed. List control options are ignored.

## LCM LEVEL 2 AND LEVEL 3 STORAGE ACCESS† (Default: LCM = D)

LCM = D — Direct mode: selects 17-bit address mode for level 2§ or 3 data. This method produces more efficient code for accessing data ssigned to level 2 or 3. User LCM, ECS, or UEM field length must not exceed 131,071 words.

LCM = I — Indirect mode: selects 21-bit address mode for level 2§ or 3 data. This mode depends heavily upon indirect addressing. LCM = I must be specified if the execution LCM, ECS, or UEM field length exceeds 131,071 words.

LCM — Implies LCM = D

In TS mode, all LCM addressing is done in 21-bit mode, regardless of the LCM parameter.

## ML MODLEVEL (Default: ML)

ML = nnn — Specifies nnn as the value of the MODLEVEL micro used by COMPASS. nnn consists of 1 to 7 letters and digits.

ML — Uses current date in the form yyddd (where yy is the year and ddd is the number of day within the year) for the MODLEVEL micro.

## OL OBJECT LIST (SECTION 14) (Default: OL = 0)

OL — Generated object code is listed on the list output file.

OL = 0 — Object code is not listed.

The OL option conflicts with the Q and E options.

---

†See LEVEL statement, section 3, for further information.

§Applies only to Control Data CYBER 170 Model 176, CYBER 170 800 Series, CYBER 70 Model 76, and 7600 computers.

### OPT OPTIMIZATION PARAMETER (SECTION 11) (Default: OPT = 1)

| | |
|---|---|
| OPT = 0 | Fast compilation (automatically activates T and ER options). |
| OPT = 1 | Standard optimization |
| OPT = 2 | Maximum optimization |
| OPT | Implies OPT = 2 |

The OPT option conflicts with the TS and SEQ options.

### P PAGINATION (Default: P = 0)

| | |
|---|---|
| P | Page numbering of output listing is continuous from subprogram to subprogram, including intermixed COMPASS output. |
| P = 0 | Page numbers begin at 1 for each subprogram. |

### PD PRINT DENSITY (Default: job default)

| | |
|---|---|
| PD = 6 | Compile time listings are produced at a density of six lines per inch. |
| PD = 8 | Compile time listings are produced at a density of eight lines per inch. |
| PD | Implies PD = 8. |

The job default print density is assumed upon entry. If print density is changed from job default, it will be returned to job default when finished.

### PL PRINT LIMIT (Default: PL = 5000)

PL = n

n is the maximum number of records (print lines) that can be written at execution time to the file OUTPUT. Under NOS/BE 1 and SCOPE 2, n must not exceed ten characters. If n is suffixed with the letter B, it is interpreted as an octal number and must not exceed 777 777 777B; otherwise, it is interpreted as a decimal number and must not exceed 9 999 999 999.

Under NOS 1, n must not exceed seven characters. The maximum value is therefore 777 777B if octal or 9 999 999 if decimal.

The PL parameter is operative only when appearing on an FTN control statement used to compile a main program.

The print limit (specified at compilation-time either explicitly or by default) can be overridden at execution-time by a parameter of the same format appearing on the LGO or EXECUTE control card; see Execution Control Statement, section 15.

### PMD POST MORTEM DUMP (Default: PMD = 0)

| | |
|---|---|
| PMD | This parameter must be specified if the Post Mortem Dump Facility is to be used. Symbol tables are written to separate files that are accessed by the Post Mortem Dump Facility so that a symbolic analysis of error conditions, variable names and values, and traceback information can be written to an output file. |
| PMD = 0 | No symbol table files are generated. |

### PS PAGE SIZE (Default: job default)

| | |
|---|---|
| PS = n | n is the maximum number of lines per page for compiler listings (including headers). If n < 4, the default value is used. |

## PW PAGE WIDTH

(Default: PW = job default if a printer output file
PW = 72 if a terminal output file)

| | |
|---|---|
| PW | Implies PW = 72 |
| PW = n | n is the number of characters on a line of listable output. Values less than 50 or greater than 136 are diagnosed and ignored. |

The PW option is valid only with TS mode.

## Q PROGRAM VERIFICATION

(Default: Q = 0)

| | |
|---|---|
| Q | Quick mode: compiler performs full syntactic scan of the program, but no object code is produced. No code addresses are provided if a reference map is requested. This mode is substantially faster than a normal compilation; but it must not be selected if the program is to be executed. |
| Q = 0 | Normal compilation. |

The Q option conflicts with the B, C, GO, OL, TS, and E options.

## R SYMBOLIC REFERENCE MAP (SECTION 13)

(Default: R = 1)

| | |
|---|---|
| R = 0 | No map |
| R = 1 | Short map (symbols, addresses, properties, DO loop map) |
| R = 2 | Long map (short map plus references by line number) |
| R = 3 | Long map plus listing of common block members and equivalence classes |
| R | Implies R = 2 |

In TS mode, R = 3 is identical to R = 2; common and equivalence classes are not listed.

## ROUND ROUNDED ARITHMETIC COMPUTATIONS

(Default: ROUND = 0)

| | |
|---|---|
| ROUND = op | op is any combination of the arithmetic operators + - * / specified with no separators. Single precision real and complex floating point arithmetic operations are performed using the hardware rounding feature, as described in the various computer systems reference manuals. |
| ROUND = 0 | Computation is not rounded. |
| ROUND | Implies ROUND = + - * / |

The ROUND option controls only the in-line object code compiled for arithmetic expressions; it does not affect computations by library subprograms or input/output routines.

## S SYSTEM TEXT (LIBRARY) FILE

(Default: S = SYSTEXT if G parameter = 0
S = 0 if G parameter is other than G = 0)

| | |
|---|---|
| S = ovl | System text overlay, ovl, is loaded from the job's current library set. |
| S = lib/ovl | System text overlay, ovl, is loaded from the user library file or system library, lib. |
| S = 0 | System text file is not loaded when COMPASS is called to assemble any intermixed COMPASS programs. |
| S | Implies S = SYSTEXT |

This feature is for COMPASS subprograms only. Up to seven system texts can be specified by repeating this option.

## SEQ SEQUENCED INPUT (SECTION 11)

(Default: SEQ = 0)

| | |
|---|---|
| SEQ | Source input file is in sequenced line format. |
| SEQ = 0 | Source input file is in standard FORTRAN format. |

Specifying the SEQ option automatically activates the TS option; sequenced line format is not recognized in optimizing mode or by COMPASS. The SEQ option conflicts with the OPT option.

## SL SOURCE LIST (SECTION 12)

(Default: SL)

| | |
|---|---|
| SL | Source program is listed on the file specified by the L parameter. |
| SL = 0 | Source program is not listed. |

## STATIC STATIC LOADING (NOS 1, NOS/BE 1 only)

(Default: STATIC = 0)

| | |
|---|---|
| STATIC | Inhibits dynamic memory management at execution time by CRM. The compiler generates a set of LDSET,USE directives specifying each of the capsules needed by the program. The specified library programs are then statically loaded. STATIC is required for any program that dynamically extends blank common. |
| STATIC = 0 | No special LDSET directives are generated and CRM uses dynamic memory management at execution time. This option results in a decrease in field length needed at execution time. |

## SYSEDIT SYSTEM EDITING

(Default: SYSEDIT = 0)

| | |
|---|---|
| SYSEDIT | All input/output references are accomplished indirectly through a table search at object time. File names are not entry points in the main program, and subprograms do not produce external references to the file name. |
| SYSEDIT = 0 | Input/output references are accomplished directly; file names are used as entry points in the main program, and subprograms produce external references to the file name. |

This option is used when building libraries that contain more than one relocatable main program. It is also necessary when compiling subroutines containing input/output references to files declared in COBOL 4 or 5 programs.

## T ERROR TRACEBACK

(Default: T = 0)

| | |
|---|---|
| T | Full error traceback occurs when an error is detected. Calls to basic external functions are made with call-by-name sequence (section 17). |
| T = 0 | No traceback occurs when an error is detected. Calls to basic external functions are made with the more efficient call-by-value sequence. A saving in memory space and execution time is realized. |

This option is provided to assist in debugging programs. Selecting the D parameter or OPT=0 automatically activates the T option. Only the execution-time errors listed in appendix B are traced.

## TS TIMESHARING MODE (SECTION 11)

(Default: OPT = 1)

| | |
|---|---|
| TS | In time-sharing mode, compilation speed and field length are optimized at the expense of execution speed and field length. Time-sharing mode is preferable to the optimizing compilation modes (OPT = 0, 1, or 2) for the debugging stages of a program. Specifying option TS together with option C, D, E, Q, or OPT constitutes a fatal control statement error. |

## UO UNSAFE OPTIMIZATION (SECTION 11)

(Default: UO = 0)

| | |
|---|---|
| UO | Allows the compiler to perform certain optimizations which are potentially unsafe. UO is ignored unless OPT = 2 is also specified. |
| UO = 0 | Unsafe optimization is not performed. |

## X EXTERNAL TEXT NAME

(Default: X = OLDPL)

| | |
|---|---|
| X = lfn | File lfn is source of external text (XTEXT) when location field of XTEXT pseudo instruction is blank. Only one X parameter may be specified. |
| X | Implies X = OPL. |

This feature is for COMPASS subprograms only.

## Z ZERO PARAMETER

(Default: Z = 0)

| | |
|---|---|
| Z | All subroutine calls having no parameters are forced to pass a parameter list consisting of a zero word. This feature is useful to COMPASS-coded subroutines expecting a variable number of parameters. Z should not be specified unless necessary, since programs require less memory if Z is omitted. |
| Z = 0 | The zero word parameter list is not passed for calls with no parameters. |

# FTN CONTROL STATEMENT EXAMPLES

**Example 1:**

```
FTN (A,EL=F,GO,L=SEE,R=2,S=0,SL=0)
```

Selects the following options:

| | |
|---|---|
| A | Skip to an EXIT (NOS 1) or EXIT(S) (NOS/BE 1 and SCOPE 2) control statement if fatal errors occur during compilation. |
| 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. |
| R = 2 | Long reference map is listed. |
| S=0 | When COMPASS is called to assemble an intermixed COMPASS subprogram, it does not read in a system text file. |
| SL=0 | Source program is not listed. |

**Example 2:**

```
FTN (GO,T)
```

Source program on INPUT file; object code on LGO; source program, short map, informative and fatal diagnostics listed on file OUTPUT; call-by-name sequence generated for calls to basic external functions; no debug package; optimizing compilation mode; and unrounded arithmetic. Program is executed if no fatal errors occur.

**Example 3:**

```
FTN.
```

Selects the following options (unless option default values are changed by the installation):

| | | |
|---|---|---|
| A=0 | I=INPUT | R=1 |
| B=LGO | L=OUTPUT | ROUND=0 |
| BL=0 | LCM=D | S=SYSTEXT |
| C=0 | ML=yyddd | SEQ=0 |
| CC=0 | OL=0 | SL |
| D=0 | OPT=1 | STATIC=0 |
| DB=0 | P=0 | SYSEDIT=0 |
| E=0 | PD=6 | T=0 |
| EL=I | PL=5000 | TS=0 |
| ER=0 | PS=60 | UO=0 |
| G=0 | PW=126 (if not connected file) | X=OLDPL |
| GO=0 | Q=0 | Z=0 |

# COMPILATION MODES AND OPTIMIZATION 11

FORTRAN Extended provides several alternative modes for compilation. Their characteristics, together with the FTN control statement parameter required to activate them, are as follows:

| | |
|---|---|
| Q | Fastest compilation; compiler performs full syntactic scan of source code, but produces no object code. Minimum field length required for compilation approximates that of OPT=0. OPT=0, OPT=1, and OPT=2 are ignored if specified. Expedient for finding errors in a program before attempting to execute it. |
| TS | Very fast, one-pass compilation. Little optimization of object code; execution time of object code approximates that of OPT=0. Minimum field length† for compilation is 40000‡ or 35000§. Expedient for a program which is recompiled before each execution, unless execution time is over twice as large as compilation time. |
| OPT=0 | Fast, two-pass compilation; little optimization of object code. Most programs can be compiled in the minimum field length of 46000‡ or 43000§. |
| OPT=1 | Two-pass compilation; moderate optimization of object code. Most programs can be compiled in the minimum field length of 46000‡ or 43000§. Expedient for programs which are recompiled before each execution but require excessive execution time in TS mode. |
| OPT=2 | Relatively slow, two-pass compilation; extensive optimization of object code; fastest execution. Minimum field length required for compilation is 54000‡ or 51000§. Programs in which the longest program unit consists of less than about 600 statements can be compiled in a field length of 60000; above that, field length required for compilation is proportional to the number of executable statements in, and the complexity of, the longest program unit. This optimization level is expedient for programs whose code is executed many times per compilation; it should not be used for undebugged programs since code redistribution in optimization renders debugging difficult if the executing program terminates abnormally. |
| D | Activates FORTRAN Extended debugging facility (see section 9). Minimum field length required for compilation is 63000‡ or 61000§. Automatically activates OPT=0; OPT=1 and OPT=2 are ignored if specified. Specification of TS is a fatal error. Necessary for programs in which execution-time debugging is desired. |
| UO | Provides additional potentially unsafe object code optimization when both the OPT=2 and UO options are specified. |

---

†Field lengths are given in octal.

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2.

# OPTIMIZING MODE

When TS is not present on the FTN control statement (OPT=0, 1, or 2) the compiler functions in optimizing mode. Time-sharing mode and optimizing mode differ not only in the kinds of optimizations performed, but also in the listing format produced. Source listings are described in section 12, reference map format in section 13, object code format in section 14, and diagnostics in Appendix B.

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 below.

## OBJECT CODE OPTIMIZATION

### OPT=0

In the OPT=0 compilation mode, compile time evaluations are made of constant subexpressions, redundant instructions and expressions within a statement are eliminated, and PERT critical path scheduling is done to utilize the multiple functional units efficiently.

### OPT=1

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. 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:

   No entries other than by normal entry at the beginning of the loop.

   No exits other than by normal termination at the end of the loop.

   No external references (user function references or subroutine calls; input/output, STOP, or PAUSE statements, or basic external function references) in the loop.

   No IF or GOTO statement in the loop branching backward to a statement appearing previously in the loop.

### OPT=2

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).

## UO

In unsafe optimization mode, the optimizations listed below are made, in addition to the optimizations made under OPT=2, since OPT=2 must also be selected. If OPT=2 is omitted, UO is not invoked.

1. In small loops, indexed array references are prefetched unconditionally without any safety checks.

   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,110) 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 a basic external function is referenced, the compiler assumes that the contents of certain B registers are preserved for use following the function processing.

   Example:

   ```
     REAL A(10),C(10)
     .
     .
     .
     DO 10 I = 1,N
   10 C(J) = 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 basic external functions (SIN, COS, SQRT, EXP, and so on).

When UO 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 basic external functions, the B register portion of the UO 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 multi-dimensional arrays should be nested so that the range of the DO loop indexing over the first subscript is executed first.

   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.

   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.

7. The first n-1 dimensions of an n-dimensional array should be either a non-negative power of 2 or the sum or difference of two non-negative powers of 2.

8. Recurrent expressions should be grouped so that they can be recognized for optimization.

   Example:

   ```
   AA = X*A/Y
   BB = X*B/Y
   ```

   is less efficient than

   ```
   AA = A*(X/Y)
   BB = B*(X/Y)
   ```

   Likewise, invariant and constant expressions should be grouped appropriately.

   Example:

   ```
      DO 10 I = 1, 50
   10 B(I) = 1. + A(I) + X
   ```

   is less efficient than

   ```
      DO 10 I = 1, 50
   10 B(I) = (1. + X) + A(I)
   ```

   Example:

   ```
   X = 1024. * B * 3.14159
   ```

   is less efficient than

   ```
   X = (1024. * 3.14159) * B
   ```

9. Multiple references to a basic external function within a statement should be algebraically reduced to a single reference.

   Example:

   ```
   Y = ALOG(A) + ALOG(B)
   ```

   is less efficient than

   ```
   Y = ALOG(A*B)
   ```

10. In a small summation loop, it is better to use a temporary variable to keep the sum than to reference an array element directly.

Example:

```
        S = 0
        DO 100 K = 1,N
100     S = S + A(I,K) * B(K,J)
        C(I,J) = S
```

is more efficient than

```
        C(I,J) = 0
        DO 100 K = 1,N
100     C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

## TIME-SHARING MODE

When the TS option is specified on the FTN control statement, FORTRAN Extended operates in time-sharing (TS) mode. Compilation is one-pass; therefore, no overlay reloading is required to compile multiple program units, and the number of disk accesses is reduced. The minimum compilation field length is 40000 octal. The CPU time spent in compilation is 30% to 75% less than that for optimizing mode (OPT=0, OPT=1, or OPT=2). The object code is not highly optimized and thus executes approximately at the rate of that produced by OPT=0.

Time-sharing mode is permissive in that it accepts some keyword misspellings and punctuation errors. When this occurs, a warning level diagnostic is issued, since the program may not compile under optimizing mode.

Misspelled keywords will be recognized if the string length matches the keyword length, the first four characters match, and the context is unambiguous.

For example,

```
COMMUN A(2)
```

will be recognized as a COMMON declaration and a warning diagnostic will be issued. However,

```
COMMUNC(I) = 2+I
```

will be correctly interpreted as a replacement statement or a statement function definition, depending on whether or not COMMUNC was previously dimensioned.

Some punctuation errors which do not inhibit the compiler from correctly interpreting a statement will be accepted.

For example, in

```
DO 10, I = 1,10
```

the first comma will be diagnosed and ignored.

## TS LISTINGS

Listings in time-sharing mode differ from those produced in optimizing mode. These differences are described in section 12 (Compiler Listings) and under Cross Reference Map (section 13), Diagnostics (Appendix B), and Object Code (section 14).

## SEQUENCED LINE FORMAT

When time-sharing mode is selected for program compilation, a FORTRAN Extended program may be coded in sequenced line format instead of in the standard format described in section 1. If the source code is in sequenced line format, the option SEQ must be specified on the FTN control statement.

The format for sequenced line coding is as follows:

seqnum d sl stat

| | | | |
|---|---|---|---|
| seqnum | Sequence number consisting of 1-5 digits, assigned in ascending order | | |
| d | blank | First line of a statement | 1 column immediately following sequence number |
| | + | Continuation line | |
| | Any other character | Comment line | |
| sl | Optional statement label consisting of 1-5 digits. | | |
| stat | FORTRAN source statement; may begin anywhere after d and continue through column 80 | | |

Example:

```
00100 PROGRAM XYZ (OUTPUT)
00110C COMPUTE AREA
00120 DIMENSION A(100), B(100),
00130+ C(200)
00140 10 CALL SUB(A,B,C,100)
00150 STOP
00160 END
```

# COMPILER LISTINGS 12

The listings produced by FORTRAN Extended during compilation are affected by control statement options (the defaults are SL, EL=I, OL=0, R=1, and L=OUTPUT). The types of listings produced, and the control statement options that influence them, are as follows:

Source listing — Includes all source lines submitted for compilation as part of the source input file. Determined by SL control statement option.

Diagnostics — Includes informative, note, warning, ANSI, and fatal diagnostics, as determined by the EL control statement option (see Appendix B). Fatal diagnostics cannot be suppressed.

Object code — Includes generated object code, listed as assembly language instructions (see section 14). Selected by OL control statement 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 (see section 13). Determined by R control statement option.

The file to which listings are written is determined by the L control statement option; specifying L=0 suppresses all listings except fatal diagnostics (which are then written to OUTPUT).

The formats of the listings produced are also influenced by the compilation mode (time-sharing or optimizing), and by the presence of listing control directives (C/ directives), discussed below.

## OPTIMIZING MODE LISTINGS

In optimizing mode, a header line at the top of each page of compiler output contains the program unit type and name, the computer used to compile and the target computer for which the compiler was assembled, some of the control statement options, compiler version and mod-level, date, time, and page number.

The source program is listed 60 lines per page (including headers), unless a different value is specified by the PS control statement option. Every fifth source line is numbered. These numbers are used in the error messages and in the cross reference map.

Diagnostics are collected and listed at the end of each program unit (subprogram or main program). Listed with each diagnostic is the line number of the source line during the processing of which the error was detected, as well as possible information in the DETAILS column relating to the nature of the error, and the severity level of the error. Diagnostic format is explained fully in Appendix B.

Object code listings, if selected, are collected and listed together at the end of each program unit. Object listing format is described in section 14. Cross reference listing format for optimizing mode compilation is described in section 13.

## TIME-SHARING MODE LISTINGS

Time-sharing mode listings differ from optimizing mode listings in several ways. A diagnostic is listed on the listing file immediately after the source line that caused detection of the error. Object listings, when requested by the OL control statement option, are interspersed with the source code.

When the page width (PW) parameter on the FTN control statement is less than 126, the output listing is reformatted so that source statements and error messages fit within the specified limits. Source statements break at the maximum line length and resume in the tenth printed column with >>>> in columns three through six. Error messages break at the nearest blank with the second line flagged the same as source statements.

Any listing made on a file connected to a terminal with no page width specified automatically has a line length of 72 characters (the PW default for files connected to terminals).

If PW is greater than or equal to 126 (either by default or by specific setting), the header line is identical to that produced in optimizing mode. If PW is less than 126, the header line is split into two lines.

## LISTING CONTROL DIRECTIVES

LIST directives permit control over the listings produced by the EL (error level), OL (object listing), R (reference map), and SL (source listing) options selected on the FTN control statement. The LIST directives affect only the program unit in which they appear. Options are controlled only if they have been selected by the FTN control statement or by default; LIST directives cannot produce a listing for an option that was not selected.

The format of LIST directives is:

```
      7
C/    LIST,option
```

| | |
|---|---|
| C/ | must appear in columns 1 and 2 with columns 3 through 6 blank |
| option | NONE stops source program listing and can suppress the other listings |
| | ALL resumes source program listing |

LIST, option appears anywhere within columns 7 through 72. Leading, trailing, and embedded blanks are allowed; continuation is not permitted.

Statements that have a C/ in columns 1 and 2 but do not conform to the directive format are processed as comments with no diagnostic issued. A LIST directive cannot be combined with another statement through a $ separator. A LIST directive within a continuation sequence causes a fatal diagnostic to be issued. Directives can appear in a C$ debug deck.

LIST,NONE stops source program listing. The directive itself is listed but subsequent source lines, including additional LIST,NONE directives, are not listed. However, when LIST,NONE is the first physical line of a program unit, neither it nor the page header is listed.

LIST,ALL resumes source program listing beginning with the directive itself. The listing will be restarted immediately regardless of the number of preceding LIST,NONE directives. If no further LIST,NONE directives are encountered, any information requested on the FTN control statement that is normally listed following the END line is listed in full.

In optimizing mode, if LIST,NONE is active when an END statement is encountered, no reference map or object listing is output and no diagnostic summary appears unless the program unit contained at least one fatal error. If fatal errors are detected, the incorrect statements are listed as well as a complete diagnostic summary with errors of all levels requested by the EL control statement parameter.

In time-sharing mode, LIST,NONE inhibits listing of interspersed blocks of object code requested by the OL parameter. Any requested reference map is not listed if LIST,NONE is active when an END statement is encountered. Diagnostics, and the statements causing them, are listed together even if LIST,NONE is active.

Example 1:

If the R=3 (long reference map) control statement option is chosen and LIST,NONE is active for 90 lines of the 150-line source program, 60 lines of the source program are listed but map information is accumulated for all 150 lines. The complete map is listed unless LIST,NONE is active when the END statement is encountered.

Example 2:

Assume the following program is compiled with TS, EL=A, and R=3 options:

```
      PROGRAM P
C/    COMMENT
C/    LIST,NONE
C/    COMMENT
      DIMENSION A(10)
      INTEGER B/C
C/    LIST,ALL
      DO 100 I=I,10
100   A(I)=0.
C/    LIST,NONE
      RETURN
      END
```

Since LIST,NONE is active when the END statement is encountered, no reference map is produced. Listing output is:

```
         1                    PROGRAM P
                       C/     COMMENT
ANSI       *           STATEMENT IS NOT DEFINED IN ANSI
                       C/     LIST,NONE
         6                    INTEGER B/C
FATAL      *           EXPECTED COMMA FOUND */
                       C/     LIST,ALL
                              DO 100 I=I,10
                       100    A(I)=0.
        10             C/     LIST,NONE
        11                    RETURN
ANST       *           RETURN IN MAIN PROGRAM
NOTE       *           RETURN ACTS AS END

           1  FORTRAN ERROR IN  P
```

Example 3:

Assume the program of Example 2 is compiled with EL=A, R=3, and OPT=1 options. Listing output is:

```
         PROGRAM P          74/74    OPT=1


     1              PROGRAM P
             C/     COMMENT
     3       C/     LIST,NONE
     6              INTEGER B/C
     7       C/     LIST,ALL
                    DO 100 I=I,10
            100     A(I)=0.
    10       C/     LIST,NONE


CARD NR. SEVERITY  DETAILS     DIAGNOSIS OF PROBLEM

     1    ANSI                THIS STATEMENT TYPE IS NON-ANSI.
     6     FE      /          ILLEGAL SYNTAX AFTER INITIAL KEYWORD OR NAME.
     8     I       I          THIS STATEMENT REDEFINES A CURRENT LOOP CONTROL VARIABLE OR PARAMETER.
    11     I                  RETURN STATEMENT APPEARS IN MAIN PROGRAM.
```

A full error summary is produced since a fatal error was detected at line 6.

Example 4:

Example 4 shows the listing produced by a compilation resulting from the control statement:

FTN,TS,OL,R=0,PW=50.

The listing is as follows:

```
        FTN 4.6+420          02/10/76  15.32.26   PAGE     1
                              73/74    TS

             1          SUBROUTINE INIT (A,M,V)
               C        INIT PUTS THE VALUE V INTO EVERY EL
          >>>> EMENT OF THE ARRAY A
                            IDENT  INIT
           0   NE INIT
           3B   L.5         BSS     0
           3B               SB0     B2-LEN.
                            SB0     B2+L.0       TRACE.
                        DO 1 I= 1,M
           4B               BSS     0
           4B               SB0     B2+0+3
                     1  A(I)= V
             5 C
        F * STATEMENT FUNCTION DEFINITION MUST OCCUR
          >>>>    BEFORE FIRST EXECUTABLE
        W * STATEMENT LABEL IGNORED
                        ENTRY ADDIT
               C        ADDIT ADDS THE VALUE V TO EVERY ELE
          >>>> MENT IN ARRAY A
        W * ENTRY INSIDE DO LOOP IS IGNORED
                            EQ      L.11
                        DO 2 I = 1,M
        F * INDEX OF OUTER DO REDEFINED BY CURRENT DO
               C/       LIST,NONE
            10     2    A(I) = A(I) + V
        F * STATEMENT FUNCTION DEFINITION MUST OCCUR
          >>>>    BEFORE FIRST EXECUTABLE
        W * STATEMENT LABEL IGNORED
               C/       LIST,ALL
                        END
        F * STATEMENT LABEL  .2         REFERENCED BUT
          >>>>    NOT DEFINED
        F * STATEMENT LABEL  .1         REFERENCED BUT
          >>>>    NOT DEFINED
        F * DO LOOP  .1         NOT TERMINATED BEFORE END
          >>>>    OF PROGRAM
        F * DO LOOP  .2         NOT TERMINATED BEFORE END
          >>>>    OF PROGRAM
```

The error messages result from the omission of a DIMENSION statement for the array A. No object code is produced for statements following the statement in which the first fatal error occurs. Between the LIST, NONE directive and the LIST,ALL directive, only statements causing error messages are listed.

# CROSS REFERENCE MAP 13

The cross reference map is a dictionary of all programmer created symbols appearing in a program unit, with the properties of each symbol and references to each symbol listed by source line number. 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.

## OPTIMIZING COMPILATION MODE

The kind of reference map produced is determined by the R option on the FTN control statement.

| | |
|---|---|
| R = 0 | No map |
| R = 1 | Short map (symbols, addresses, properties, and a DO loop map) |
| R = 2 | Long map (short map plus references by line number) |
| R = 3 | Long map and printout of common block members and equivalence classes |
| R | Implies R = 2 |

If R is not specified, the default option is R = 1; however, L = 0 forces R = 0.

Fatal errors in the source program will cause certain parts of the map to be suppressed, incomplete, or inaccurate. Fatal to execution (FE) and fatal to compilation (FC) errors will cause the DO-loop map to be suppressed, and assigned addresses will be different; symbol references may not be accumulated for statements containing syntax errors.

For the long map, it may be necessary to increase field length by 1000(octal).

The number of references that can be accumulated and sorted for mapping is: field length minus 20000 (octal) minus 4 times the number of symbols. For example, in a source program containing 1000 (decimal) symbols, approximately 8000 (decimal) references can be accumulated with a field length of 50000 octal.

Examples from the cross-reference map produced by the program which follows are interspersed with the general format discussions.

The source program and the reference maps produced for both R = 1 and R = 3 follow. A complete set of maps for R = 2 is not included, but samples are shown with the discussion.

## SOURCE PROGRAM

### Main Program

```
          PROGRAM MAPS                                                  MAPS 005
         1(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)                       MAPS 006
          INTEGER SIZE1, S1, SIZE2, S2     ,STRAY                       MAPS 007
          EQUIVALENCE(SIZE1,S1),(SIZE2,S2)                              MAPS 008
          NAMELIST/PARAMS/SIZE1,SIZE2                                   MAPS 009
          DATA S1,S2/12,12/                                             MAPS 010
  100     READ(5,PARAMS)                                                MAPS 011
          WRITE(6,PARAMS)                                               MAPS 012
          PRINT1                                                        MAPS 013
    1     FORMAT(≠0SAMPLE PROGRAM TO ILLUSTRATE THE VARIOUS COMPILER MAPS.≠)MAPS 014
          CALL PASCAL(S1)                                               MAPS 015
          PRINT2                                                        MAPS 016
    2     FORMAT(≠0THE FOLLOWING WILL HAVE NO HEADINGS.≠)               MAPS 017
          CALL NOHEAD(S2)                                               MAPS 018
          STOP                                                          MAPS 019
          END                                                           MAPS 020
```

### Block Data Subprogram

```
          BLOCK DATA                                                    MAPS 021
          COMMON/ANARRAY/X(22)                                          MAPS 022
          INTEGER X                                                     MAPS 023
          DATA X(22)/1/                                                 MAPS 024
          END                                                           MAPS 025
```

### Subprogram with second entry

```
          SUBROUTINE PASCAL(SIZE)                                       MAPS 026
          INTEGER L(22),SIZE                                            MAPS 027
          COMMON/ANARRAY/L                                              MAPS 028
          PRINT4, (1,I=1,SIZE)                                          MAPS 029
    4     FORMAT(44H0COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/MAPS 030
         $22I6)                                                         MAPS 031
          ENTRY NOHEAD                                                  MAPS 032
          M=MIN0(21,MAX0(2,SIZE-1))                                     MAPS 033
          DO2I=1,M                                                      MAPS 034
          K=22-I                                                        MAPS 035
          L(K)=1                                                        MAPS 036
          DO1J=K,21                                                     MAPS 037
    1     L(J)=L(J)+L(J+1)                                              MAPS 038
    2     PRINT3,(L(J),J=K,22)                                          MAPS 039
    3     FORMAT(22I6)                                                  MAPS 040
          RETURN                                                        MAPS 041
          END                                                           MAPS 042
```

## R=1 MAPS

```
      SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS
 4111  MAPS

VARIABLES     SN  TYPE            RELOCATION
 4176  SIZE1      INTEGER                                4177  SIZE2     INTEGER
 4175  STRAY      INTEGER   *UNDEF                       4176  S1        INTEGER
 4177  S2         INTEGER

FILE NAMES        MODE
    0  INPUT                  2041  OUTPUT    FMT               0  TAPE5     NAME          2041  TAPE6     NAME

EXTERNALS             TYPE   ARGS
       NOHEAD                  1                              PASCAL              1

NAMELISTS
       PARAMS

STATEMENT LABELS
 4153  1        FMT                      4166  2        FMT                              0  100        INACTIVE

STATISTICS
  PROGRAM LENGTH               75B      61
  BUFFER LENGTH              4103B    2115

      BLOCK DATA            74/74    OPT=1       FTN 4.4+REL.        02/28/75  09.32.57.       PAGE      1

      SYMBOLIC REFERENCE MAP (R=1)

VARIABLES     SN  TYPE            RELOCATION
    0  X          INTEGER    ARRAY    ANARRAY

COMMON BLOCKS   LENGTH
       ANARRAY      22

STATISTICS
  PROGRAM LENGTH                 0B       0
  CM LABELED COMMON LENGTH      26B      22

      SUBROUTINE PASCAL     74/74    OPT=1       FTN 4.4+REL.        02/28/75  09.32.57.       PAGE      1

      SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS
   27  NOHEAD            3  PASCAL

VARIABLES     SN  TYPE            RELOCATION
  115  I          INTEGER                                 120  J         INTEGER
  117  K          INTEGER                                   0  L         INTEGER     ARRAY     ANARRAY
  116  M          INTEGER                                   0  SIZE      INTEGER                F.P.

FILE NAMES        MODE
       OUTPUT     FMT

INLINE FUNCTIONS    TYPE    ARGS
       MAX0       INTEGER    0  INTRIN                    MIN0      INTEGER     0  INTRIN

STATEMENT LABELS
    0  1                                    0  2                                     113  3        FMT
   76  4        FMT

LOOPS  LABEL     INDEX      FROM-TO      LENGTH      PROPERTIES
   21            I            4 4          4B                  EXT REFS
   44  2         I            9 14        24B                  EXT REFS  NOT INNER
   52  1         J           12 13         3B          INSTACK

COMMON BLOCKS   LENGTH
       ANARRAY      22

STATISTICS
  PROGRAM LENGTH               123B      83
  CM LABELED COMMON LENGTH      26B      22
```

```
      SYMBOLIC REFERENCE MAP (R=3)

ENTRY POINTS      DEF LINE      REFERENCES
 4111  MAPS              1

VARIABLES      SN  TYPE             RELOCATION
 4176  SIZE1       INTEGER                          REFS     3      4      5
 4177  SIZE2       INTEGER                          REFS     3      4      5
 4175  STRAY    *  INTEGER    *UNDEF                REFS     3
 4176  S1          INTEGER                          REFS     3      4     11    DEFINED      6
 4177  S2          INTEGER                          REFS     3      4     14    DEFINED      6

FILE NAMES         MODE
    0  INPUT
 2041  OUTPUT     FMT                     WRITES       9      12
    0  TAPE5      NAME                     READS       7
 2041  TAPE6      NAME                    WRITES       8

EXTERNALS          TYPE    ARGS          REFERENCES
       NOHEAD                1               14
       PASCAL                1               11

NAMELISTS          DEF LINE      REFERENCES
       PARAMS            5          7         8

STATEMENT LABELS               DEF LINE   REFERENCES
 4153  1       FMT                10          9
 4166  2       FMT                13         12
    0  100          INACTIVE       7

EQUIV CLASSES   LENGTH      MEMBERS - BIAS NAME(LENGTH)
       SIZE1        1                   0 S1       (1)
       SIZE2        1                   0 S2       (1)
```

← omitted from R=2 map

```
STATISTICS
  PROGRAM LENGTH                  75B      61
  BUFFER LENGTH                 4103B    2115


      BLOCK DATA           74/74   OPT=1                FTN 4.4+REL.      02/28/75  09.36.38.      PAGE     2

      SYMBOLIC REFERENCE MAP (R=3)

VARIABLES      SN  TYPE           RELOCATION
    0  X           INTEGER    ARRAY     ANARRAY       REFS       2        3    DEFINED      4

COMMON BLOCKS    LENGTH     MEMBERS - BIAS NAME(LENGTH)
       ANARRAY       22                 0 X       (22)
```

← omitted from R=2 map

```
STATISTICS
  PROGRAM LENGTH                 0B        0
  CM LABELED COMMON LENGTH     26B       22


      SUBROUTINE PASCAL    74/74   OPT=1                FTN 4.4+REL.      02/28/75  09.36.38.      PAGE     2

      SYMBOLIC REFERENCE MAP (R=3)

ENTRY POINTS      DEF LINE     REFERENCES
   27  NOHEAD            7        16
    3  PASCAL            1

VARIABLES      SN  TYPE           RELOCATION
  115  I           INTEGER                          REFS       4      10    DEFINED      4       9
  120  J           INTEGER                          REFS     3*13     14    DEFINED     12      14
  117  K           INTEGER                          REFS      11      12      14    DEFINED     10
    0  L           INTEGER    ARRAY     ANARRAY     REFS       2       3     2*13     14    DEFINED     11     13
  116  M           INTEGER                          REFS       9   DEFINED     8
    0  SIZE        INTEGER              F.P.        REFS       2       4       8    DEFINED      1

FILE NAMES         MODE
       OUTPUT     FMT                    WRITES       4       14

INLINE FUNCTIONS    TYPE     ARGS        DEF LINE  REFERENCES
       MAX0        INTEGER    0  INTRIN                   8
       MIN0        INTEGER    0  INTRIN                   8

STATEMENT LABELS              DEF LINE   REFERENCES
    0  1                         13         12
    0  2                         14          9
  113  3       FMT               15         14
   76  4       FMT                5          4

LOOPS  LABEL    INDEX     FROM-TO    LENGTH    PROPERTIES
   21            I          4  4       4B               EXT REFS
   44  2         I          9 14      24B               EXT REFS  NOT INNER
   52  1         J         12 13       3B     INSTACK

COMMON BLOCKS    LENGTH     MEMBERS - BIAS NAME(LENGTH)
       ANARRAY       22                 0 L       (22)
```

← omitted from R=2 map

```
STATISTICS
  PROGRAM LENGTH                123B       83
  CM LABELED COMMON LENGTH       26B       22
```

General Format:

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.
>
> Properties associated with the symbol
>
> List of references to the symbol (for R=2 and R=3 only)

All line numbers in the reference list refer to the line of the statement in which the reference occurs. Multiple references in a statement are printed as n*i where n is the number of references on line i.

All numbers to the right of the name are decimal integers unless they are suffixed with B 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.

## ENTRY POINTS

Entry point names include program and subprogram names and names appearing in ENTRY statements. The format of this map is:

| ENTRY POINTS | | DEFINITION | REFERENCES |
|---|---|---|---|
| **addr** | **name** | **def** | **ref** |

| | |
|---|---|
| **addr** | Relative address assigned to the entry point. |
| **name** | Entry point name as defined in FORTRAN source. |
| **def** | Line number on which entry point **name** is defined (PROGRAM statement, SUBROUTINE statement, ENTRY statement, etc.). (Not on R=1 maps.) |
| **ref** | In subprograms only, line number of RETURN statements. (Not on R=1 maps.) |

R=1:

```
ENTRY POINTS
   27  NOHEAD          3  PASCAL
```

R=2 and R=3:

```
ENTRY POINTS     DEF LINE    • REFERENCES
   27  NOHEAD          7         16
    3  PASCAL          1
```

## VARIABLES

Variable names include local and COMMON variables and arrays, formal parameters, RETURNS names, and for FUNCTION subprograms, the defined function name when used as a variable. The format of this map is:

| VARIABLES | | SN | TYPE | | RELOCATION | |
|---|---|---|---|---|---|---|
| **addr** | **name** | * | **type** | **prop** | **block** | **refs** |

**addr** — Relative address assigned to variable **name**. If **name** is a member of a COMMON block, **addr** is relative to the start of **block**.

**name** — Variable name as it appears in FORTRAN source listing. Variables are listed in alphabetical order.

* — SN = stray name flag. (No entry appears under SN when R=1 is specified.) Variable names that appear only once in a subprogram are indicated by * under the SN headline. Such variable names are likely keypunch errors, misspellings, etc. In the long map, DO loops where the index variable is not referenced cause the index variable to be flagged as a stray name.

**type** — LOGICAL, INTEGER, REAL, COMPLEX, or DOUBLE. Gives the arithmetic mode associated with the variable **name**. RETURNS appears if **name** is a RETURNS formal parameter.

**prop** — Properties associated with variable **name** and printed by keywords in this column:

*UNDEF — Variable **name** has not been defined. A variable is defined if any of the following conditions holds:

**name** 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.

ARRAY — Variable **name** is dimensioned.

*UNUSED — **name** is an unused formal parameter.

**block** — Name of COMMON block in which variable **name** appears. If blank, **name** is a local variable.

// — indicates **name** is in blank COMMON.
F.P. — indicates **name** is a formal parameter.

refs (Does not appear in short map, R=1.)
References and definitions associated with variable **name** are listed by line number, beginning with the following in-line subheadings:

REFS — All appearances of **name** in declarative statements or statements where the value of **name** is used.

DEFINED — All appearances of **name** where its value may be altered such as in DATA, ASSIGN, READ, ENCODE, or DECODE, BUFFER IN, assignment statements, or as a DO loop index.

IO REFS — All appearances of **name** in use as a variable file name in I/O statements.

R=1: This map form uses a double column format to conserve space. Headings appear only on the first columns.

```
VARIABLES     SN  TYPE           RELOCATION
  115  I          INTEGER                               120  J        INTEGER
  117  K          INTEGER                                 0  L        INTEGER    ARRAY    ANARRAY
  116  M          INTEGER                                 0  SIZE     INTEGER              F.P.
```

R=2 and R=3:

```
VARIABLES     SN  TYPE           RELOCATION
  115  I          INTEGER                         REFS     4       10    DEFINED      4       9
  120  J          INTEGER                         REFS   3*13      14    DEFINED     12      14
  117  K          INTEGER                         REFS    11       12       14    DEFINED    10
    0  L          INTEGER    ARRAY    ANARRAY     REFS     2        3     2*13       14   DEFINED     11      13
  116  M          INTEGER                         REFS     9   DEFINED       8
    0  SIZE       INTEGER              F.P.       REFS     2        4        8    DEFINED     1
```

## FILE NAMES

File names include those explicitly defined in the PROGRAM statement as well as those implicitly defined (in subprograms) through usage in input/output statements. The format of this map is:

| FILE NAMES | MODE |
|---|---|
| **addr name** | **mode refs** |

**addr** Relative address of the file information table (FIT) associated with the file **name**. The file's buffer starts at **addr**+34B This column appears only in main programs (where the file is actually defined). In subprograms, this column is blank.

**name** Name of the file as defined in PROGRAM statement or implied from usage in input/output statements. For example, in a subprogram, WRITE(2) implies a reference to file TAPE2.

**mode** Indicates the mode of the file, as implied from it usage. One of the following will be printed:

| | | |
|---|---|---|
| FMT | Formatted I/O e.g. | READ(2,901) |
| FREE | List Directed I/O | READ(2,*) |
| UNFMT | Unformatted I/O | READ(2) |
| NAME | Namelist Name I/O | READ(2,NAMEIN) |
| BUF | Buffer I/O | BUFFER IN(2,0) |
| MIXED | Some combination of the above. | |
| blank | Mode cannot be determined. | |

**refs** (Does not appear in short map, R=1.)
References are divided into three categories by in-line subheadings:

| | |
|---|---|
| READS | followed by list of line numbers referencing file **name** in input operations. |
| WRITES | line numbers of output operations on file **name**. |
| MOTION | line numbers of positioning operations (REWIND, BACKSPACE, ENDFILE) on file **name**. |

R=1:

```
FILE NAMES      MODE
     0  INPUT              2041  OUTPUT    FMT              0  TAPE5     NAME           2041  TAPE6     NAME
```

R=2 and R=3:

```
FILE NAMES      MODE
     0  INPUT
  2041  OUTPUT    FMT                   WRITES        9         12
     0  TAPE5     NAME                   READS        7
  2041  TAPE6     NAME                  WRITES        8
```

When a variable is used as a unit number in an input/output statement the following message is printed:

VARIABLE USED AS FILE NAMES, SEE ABOVE

## EXTERNAL REFERENCES

External references include names of functions or subroutines called explicitly from a program or subprogram, as well as names declared in an EXTERNAL statement. Implicit external references, such as those called by certain FORTRAN source statements (READ, ENCODE, etc.) are not listed. The format of this map is:

| EXTERNALS | TYPE | ARGS | | REFERENCES |
|---|---|---|---|---|
| **name** | **type** | **args** | **prop** | **refs** |

**name** External name as it appears in source listing.

**type** Applies to externals used as functions. Possible keywords are:

REAL, INTEGER, COMPLEX, DOUBLE, LOGICAL

Gives the arithmetic mode of external function.

NO TYPE — No specific arithmetic mode defined. Applies to certain library functions listed as externals in T mode. (T mode is implied when OPT=0 or D mode is selected.)

This column will be blank for all externals used as subroutines in CALL statements.

**args** Number of arguments in call to external **name**.

**prop** Special properties associated with external **name**:

F.P — **name** is a formal parameter (applies only for references within a subprogram).

LIBRARY — **name** is a library function called by value. In T compile modes, no LIBRARY entries appear since all references to library functions (SIN, COS, etc.) will be by name. (OPT=0 or D mode automatically implies T mode.)

**refs** Line number on which **name** is referenced. (Does not appear in short map, R=1.)

R=1:

```
EXTERNALS          TYPE   ARGS
       NOHEAD               1                         PASCAL              1
```

R=2 and R=3:

```
EXTERNALS          TYPE   ARGS        REFERENCES
       NOHEAD               1             14
       PASCAL               1             11
```

## INLINE FUNCTIONS

Inline functions include names of intrinsic and statement functions appearing in the subprogram. The subtitle line is:

| INLINE FUNCTIONS | TYPE | ARGS | DEF | LINE | REFERENCES |
|---|---|---|---|---|---|
| **name** | **mode** | **args** | **ftype** | **def** | **refs** |

**name** Symbol name as it appears in the listing.

**mode** Arithmetic mode, NO TYPE means no conversion in mixed mode expressions.

**args** Number of arguments with which the function is referenced. For functions with a variable number of arguments (such as MAX, AND, etc.) 0 is shown.

**ftype** INTRIN — Intrinsic function.

SF — Statement function.

**def** Blank for intrinsic functions; the definition line for statement functions.

**refs** Lines on which function is referenced.

R=1:

```
INLINE FUNCTIONS   TYPE   ARGS
      MAX0       INTEGER    0  INTRIN                        MIN0       INTEGER    0  INTRIN
```

R=2 and R=3:

```
INLINE FUNCTIONS   TYPE   ARGS          DEF LINE   REFERENCES
      MAX0       INTEGER    0  INTRIN                      8
      MIN0       INTEGER    0  INTRIN                      8
```

## NAMELISTS

| NAMELISTS | DEF LINE | REFERENCES |
|---|---|---|
| **name** | **def** | **refs** |

**name** Namelist group name as defined in FORTRAN source.

**def** Line on which namelist is defined. (Does not appear in short map.)

**refs** Line numbers of references to **name**. (Does not appear in short map.)

R=1:

```
NAMELISTS
      PARAMS
```

R=2 and R=3:

```
NAMELISTS       DEF LINE     REFERENCES
      PARAMS        5            7            8
```

## STATEMENT LABELS

The statement label map includes all statement labels defined in the program or subprogram. The format of this map is:

| STATEMENT LABELS | | | DEF LINE | | REFERENCE |
|---|---|---|---|---|---|
| **addr** | **label** | **type** | **act** | **def** | **refs** |

**addr** Relative address assigned to statement **label**. Inactive labels will have **addr** zero. Terminal statements of a DO loop also will have **addr** zero (unless referenced as the object of a transfer of control). 400 000 will be shown if no address is assigned; usually, a fatal error occurred and the final phase of compilation did not take place.

**label** Statement label from FORTRAN source program. Statement labels are listed in numerical order.

**type** One of the following keywords:

FMT Statement **label** is a FORMAT statement.

UNDEF Statement **label** is undefined. **refs** lists all references to this undefined label.

blank Statement **label** appears on a valid executable statement.

| | | |
|---|---|---|
| **act** | One of the following keywords: | |
| | INACTIVE | **label** is considered inactive. It may have been deleted by optimization. Inactive labels will have **addr** zero. |
| | NO REFS | **label** is not referenced by any statements. This label may be removed safely from the FORTRAN source program. |
| | blank | **label** is active or referenced. |
| **def** | Line number on which **label** was defined. (Does not appear in short map.) | |
| **refs** | Line numbers on which **label** was referenced. (Does not appear in short map.) | |

R=1:

```
STATEMENT LABELS
    0  1                                0  2                                113  3       FMT
   76  4       FMT
```

R=2 and R=3:

```
STATEMENT LABELS             DEF LINE   REFERENCES
    0  1                        13         12
    0  2                        14          9
  113  3       FMT              15         14
   76  4       FMT               5          4
```

## DO LOOPS

The DO-loop map includes all DO loops as well as implied DO loops not in DATA statements that appear in the program and lists their properties. This map is suppressed if fatal errors have been detected in the source program or if Q was specified on the FTN control statement. Loops are listed in order of appearance in the program. The format of this map is:

| LOOPS | LABEL | INDEX | FROM-TO | LENGTH | PROPERTIES |
|---|---|---|---|---|---|
| **fwa** | **term** | **index** | **first-last** | **len** | **prop** |

| | |
|---|---|
| **fwa** | Relative address assigned to the start of loop body. |
| **term** | Statement label defined as end of loop, or blank for implied DO loops in input/output statements. |
| **index** | Variable name used as control index for loop, as defined by DO statement. |
| **first-last** | Line numbers of the **first** and **last** statements of the loop. |

**len** Number of words generated for the body of the loop (octal).

**prop** Various keywords might appear, describing optimization properties of the loop:

| | |
|---|---|
| OPT | Loop has been optimized. |
| INSTACK | Loop fits into instruction stack (less than or equal to 7‡ or 10§ words); likely to run two to three times as fast as a comparable loop that does not fit into the stack. |
| EXT REFS | Loop not optimized because it contains references to an external subprogram, or it is the implied loop of an input/output statement. |
| ENTRIES | Loop not optimized because it contains entries from outside its range. |
| NOT INNER | Loop not optimized because it is not the innermost loop in a nest. |
| EXITS | Loop not optimized because it contains references to statement labels outside its range. |

R=1, R=2, and R=3:

```
LOOPS  LABEL    INDEX     FROM-TO    LENGTH    PROPERTIES
  20            I           4           48                 EXT REFS
  43   2        I           9 14       208                 EXT REFS  NOT INNER
  52   1        J          12 13        28      INSTACK
```

## COMMON BLOCKS

The common block map lists common blocks and their members as defined in the source program. The format of this map is:

```
COMMON BLOCKS                LENGTH          MEMBER – BIAS NAME(LENGTH)
    block          storage    blen             bias      member     (size)
                    type
```

**block** Common block name as defined in COMMON statement.
// represents blank common.

**storage type** Hardware type of storage device where the block is located: ECS, LCM, or blank (blank indicates CM or SCM).

**blen** Total length of **block** in decimal.

‡Applies only to Control Data CYBER 70 Model 74 and 6600 computers.

§Applies only to Control Data CYBER 70 Model 76, CYBER 170 Models 175 and 176, and 7600 computers.

If the long map is specified (R=3) the following details are printed for each member of each block:

| | |
|---|---|
| **bias** | Relative position of **member** in **block**; in decimal, gives the distance from the block origin. |
| **member** | Variable name defined as a member of **block.** |
| **size** | Number of words allocated for **member.** |

Only variables defined as members of a common block explicitly by a COMMON statement are listed in this map. Variables which become implicit members of a common block by EQUIVALENCE statements are listed in the EQUIV CLASS map and the variable map.

R=1 and R=2:

```
COMMON BLOCKS   LENGTH
       ANARRAY      22
```

R=3:

```
COMMON BLOCKS   LENGTH     MEMBERS - BIAS NAME(LENGTH)
       ANARRAY      22                 0 L     (22)
```

## EQUIVALENCE CLASSES

This map appears only when R=3 is selected. All members of an equivalence class of variables explicitly equated in EQUIVALENCE statements are listed. Variables added through linkage to common blocks are not included. The format of the map is:

| EQUIV | CLASSES | LENGTH | MEMBERS – BIAS NAME (LENGTH) | | |
|---|---|---|---|---|---|
| **cbase** | **base** | **clen** | **bias** | **member** | **(size)** |

**cbase** Common base. A variable name appears here if the equivalence class is in a common block. In such a case, **cbase** is the variable name of the first member in that common block.

*UNDEF Indicates this class is in error because more than one member is in common or the origin of the block is extended by equivalence.

**base** If the class is local (not in a common block), **base** is the name of the variable with the lowest address. If the class is in a common block, **base** is the name of the variable in that common block to which other variables were linked through an EQUIVALENCE statement.

**clen** Number of words allocated for **base** (considered the class length).

**bias** Position of **member** relative to **base; bias** is in decimal.

**member** Variable name defined as a member of an equivalence class. (Members having the same **bias** which are associated with the same **base** and thus occupy the same locations.)

**size** Size of **member** as defined by DIMENSION, etc.

R=3 only:

```
EQUIV CLASSES   LENGTH     MEMBERS - BIAS NAME(LENGTH)
      SIZE1          1                 0 S1     (1)
      SIZE2          1                 0 S2     (1)
```

## PROGRAM STATISTICS

At the end of the reference map, the statistics are printed in octal and decimal. The format is:

**STATISTICS**

| | |
|---|---|
| **PROGRAM LENGTH** | Length of program including code, storage for local variables, arrays, constants, temporaries, etc., but excluding buffers and common blocks. |
| **BUFFER LENGTH** | Total space occupied by input/output buffers and file information table. |
| **CM LABELED COMMON LENGTH** | Total length of common, excluding blank common, in CM/SCM and ECS/LCM. Maximum of two entries. |
| **BLANK COMMON** | Length of blank common in CM/SCM or ECS/LCM. |

R=1, R=2, and R=3:

```
STATISTICS
  PROGRAM LENGTH                123B      83
  CM LABELED COMMON LENGTH       26B      22
```

## ERROR MESSAGES

The following error messages are printed if sufficient storage is not available:

CANT SORT THE SYMBOL TABLE     INCREASE FL BY NNNB

or

REFERENCES AFTER LINE NNN LOST  INCREASE FL BY NNNB

## DEBUGGING (USING THE REFERENCE MAP)

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 FE errors until the program compiles.

Using the listing, the R=3 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 inline instead of external

Variables or functions with incorrect type

Unreferenced format statements

Unused formal parameters

Ordering of members in common blocks

Equivalence classes

Existing Program:

The reference map can be used to understand the structure of an existing 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.

## TIME-SHARING MODE

In TS mode, the reference map appears immediately following the source listing of the program (regardless of the BL control statement parameter). Line length of the listing is determined by the PW control statement parameter.

The kind of reference map produced is determined by the R option on the FTN control statement:

| | |
|---|---|
| R=0 | No map |
| R=1 | Short map (symbols, addresses, properties) |
| R=2<br>R=3 | Long map (short map plus references by page and line number) |
| R | Implies R=2 |

If R is omitted, an R=1 map is produced (unless L=0 is specified on the FTN control statement).

On the following pages appear examples of a short and a long map. Portions of these maps appear in the subsequent format discussion.

```
                 SUBROUTINE PASCAL(SIZE)                                             MAPS 026
                 INTEGER L(22),SIZE                                                  MAPS 027
                 COMMON/ANARRAY/L                                                    MAPS 028
                 PRINT4, (I,I=1,SIZE)                                                MAPS 029
            4    FORMAT(44H0COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/MAPS 030
                $22I5)                                                               MAPS 031
                 ENTRY NOHEAD                                                        MAPS 032
                 M=MIN0(21,MAX0(2,SIZE-1))                                           MAPS 033
                 DO2I=1,M                                                            MAPS 034
                 K=22-I                                                              MAPS 035
                 L(K)=1                                                              MAPS 036
                 DO1J=K,21                                                           MAPS 037
            1    L(J)=L(J)+L(J+1)                                                    MAPS 038
            2    PRINT3,(L(J),J=K,22)                                                MAPS 039
            3    FORMAT(22I6)                                                        MAPS 040
                 RETURN                                                              MAPS 041
                 END                                                                 MAPS 042


--COMMON BLOCKS--

           26B  /ANARRAY/

--ENTRY POINTS--

           16B   NOHEAD              66B   PASCAL

--EXTERNALS--

                OUTCI.    OUTCR.    OUTPUT=

--STATEMENT LABELS--

        .1   ID      03          .2   ID      46B          .3    F     102B          .4    F      72B

--VARIABLE MAP--

    I          I  U    117B                       J          I  U    123B
    K          I       121B                       L          I         0B /ANARRAY/  22
    M          I       122B                       MAX0       I            INTRINSIC
    MIN0       I            INTRINSIC             NOHEAD     -        16B ENTRY
    OUTCI.     -            EXTERNAL.             OUTCR.     -            EXTERNAL.
    OUTPUT=    -            EXTERNAL.             PASCAL     -        66B ENTRY
    SIZE       I AU      0B

    151B  PROGRAM-UNIT LENGTH        17 SYMBOLS

    41000B CM STORAGE USED         .077 SECONDS
```

```
                  SUBROUTINE PASCAL(SIZE)                                     MAPS 026
                  INTEGER L(22),SIZE                                          MAPS 027
                  COMMON/ANARRAY/L                                            MAPS 028
                  PRINT4, (I,I=1,SIZE)                                        MAPS 029
           4      FORMAT(44H0COMBINATIONS OF N THINGS TAKEN N AT A TIME.//20X,3H-N-/MAPS 030
                 322I6)                                                       MAPS 031
                  ENTRY NOHEAD                                                MAPS 032
                  M=MIN0(21,MAX0(2,SIZE-1))                                   MAPS 033
                  DO2I=1,M                                                    MAPS 034
                  K=22-I                                                      MAPS 035
                  L(K)=1                                                      MAPS 036
                  DO1J=K,21                                                   MAPS 037
           1      L(J)=L(J)+L(J+1)                                            MAPS 038
           2      PRINT3,(L(J),J=K,22)                                        MAPS 039
           3      FORMAT(22I6)                                                MAPS 040
                  RETURN                                                      MAPS 041
                  END                                                         MAPS 042

--COMMON BLOCKS--

            26B  /ANARRAY/

--ENTRY POINTS--

            16B   NOHEAD                  66B   PASCAL

--EXTERNALS--

                  OUTCI.     OUTCR.     OUTPUTE

--STATEMENT LABELS--

          .1     IO        0B       12         13 L
          .2     ID       46B        9         14 L
          .3      F      102B       14 W       15 L
          .4      F       72B        4 W        5 L

--VARIABLE MAP--

     I          I  U    117B                          4 C       4 W       9 C      10
     J          I  U    120B                         12 C      13 S      13 S      13 S      14 C      14 S
     K          I       121B                         10 =      11 S      12 C      14 C
     L          I        0B /ANARRAY/  22             2 D       3 D      11 =      13        13        13 =      14 W
     M          I       122B                          8 =       9 C
     MAX0       I            INTRINSIC                8 A
     MIN0       I            INTRINSIC                8
     NOHEAD     -        16B ENTRY                    7 E
     OUTCI.     -            EXTERNAL.                4 W      14 W

--VARIABLE MAP--

     OUTCR.     -            EXTERNAL.                4 W       4 W      14 W      14 W
     OUTPUTE    -            EXTERNAL.                4 W      14 W
     PASCAL     -       66B  ENTRY                    1 E
     SIZE       I AU     0B                           1 A       2 D       4 C       8 A

   151B  PROGRAM-UNIT LENGTH        17 SYMBOLS        47 REFERENCES

   41000B CM STORAGE USED          .089 SECONDS
```

## COMMON BLOCKS

The common block map lists common blocks as defined in the source program. The format of this map is:

```
- - COMMON BLOCKS - -
```

**length** **/block/**

**length** Length (in octal) of common block.

**block** Common block name as defined in COMMON statement.
// represents blank common.

R=1, R=2, R=3:

```
--COMMON BLOCKS--

                    26B   /ANARRAY/
```

## ENTRY POINTS

This map lists names of program units, names appearing in ENTRY statements, and (for a main program) all file names defined in the PROGRAM statement. The format is:

```
- - ENTRY POINTS - -
```

**addr** **name**

**addr** Relative address (in octal) of the entry point in the program unit.

**name** Entry point name as defined in source program.

R=1, R=2, R=3:

```
--ENTRY POINTS--

                    16B   NOHEAD                          66B   PASCAL
```

## EXTERNAL REFERENCES

External references include names of functions or subroutines called explicitly from a program or subprogram, names declared in an EXTERNAL statement, and external references generated by the compiler. The format of this map is:

```
- - EXTERNALS - -
```

**name**

**name** Name of routine externally referenced.

R=1, R=2, R=3:

```
--EXTERNALS--

                    OUTCI.    OUTCR.    OUTPUT=
```

## STATEMENT LABELS

This map includes all statement labels defined in the program or subprogram. The format is:

- - STATEMENT LABELS - -

**label** **properties** **addr** **references**

**label** Statement label, preceded by a period. Labels are listed in ascending numerical order.

**properties** Properties as follows:

| | |
|---|---|
| F | **label** references a format statement. |
| D | **label** references a terminal statement of a DO loop. |
| I | **label** is inactive (never referenced by transfer or input/output statement). |
| blank | None of the above properties. |

**addr** Relative address (in octal) assigned to this label. Some inactive labels will have an addr of zero.

**references** Line number and type of reference to statement label. References do not appear in the short map (R=1). The type can be:

| | |
|---|---|
| L | **label** appears in label field. |
| D | **label** referenced in a DO statement. |
| R | **label** referenced in a READ statement. |
| W | **label** referenced in a WRITE or PRINT statement. |
| F | **label** referenced in a FORMAT statement. |
| A | **label** referenced in an ASSIGN statement. |
| blank | Any other reference. |

R=1:

```
--STATEMENT LABELS--

       .1   ID      0B       .2   ID     46B       .3   F     102B       .4   F     72B
```

R=2, R=3:

```
--STATEMENT LABELS--

       .1   ID     0B     12       13 L
       .2   ID    46B      9       14 L
       .3    F   102B     14 W     15 L
       .4    F    72B      4 W      5 L
```

## VARIABLES

All symbolic names referenced in the program unit are listed here. The format of this map is:

-- VARIABLE MAP --

**name type properties addr block length references**

| | |
|---|---|
| **name** | Name of variable as it appears in source listing. |
| **type** | Variable type: |

| | |
|---|---|
| I | INTEGER |
| R | REAL |
| D | DOUBLE PRECISION |
| Z | COMPLEX |
| L | LOGICAL |
| N | NAMELIST name |
| - | No type |

**properties** Properties as follows:

| | |
|---|---|
| A | Variable is used as a formal parameter. |
| U | Variable is undefined. |
| ≡ | Variable is equivalenced to a defined variable. |
| blank | None of the above. |

**addr** Relative address (in octal) assigned to this variable.

**block** Name of common block in which variable appears, or (if no address is specified) a description of the type of symbolic name:

| | |
|---|---|
| ENTRY | **name** is an entry point. |
| SUBROUTINE | **name** is a user supplied SUBROUTINE subprogram or a library utility subprogram. |
| INTRINSIC | **name** is an intrinsic function. |
| STAT-FUNC | **name** is a statement function. |
| B.E.F. | **name** is a basic external function. |
| FUNCTION | **name** is a user supplied FUNCTION subprogram. |
| EXTERNAL | **name** appears in an EXTERNAL statement or is a compiler generated external reference. |

**length** Array length (in decimal) for dimensioned variables.

**references** Line number and type of reference to variable. References do not appear in the short map (R=1). The type can be:

| | |
|---|---|
| A | Variable appears as argument to subroutine or function. |
| C | Variable appears as DO loop control variable. |
| D | Variable appears in specification statement. |
| E | Variable used as entry point. |
| F | Variable appears in IF statement. |

| | |
|---|---|
| I | Variable appears in DATA statement. |
| R | Variable appears in READ statement. |
| S | Variable appears in subscript. |
| W | Variable appears in WRITE or PRINT statement. |
| X | Variable appears as an external reference. |
| = | Variable appears on the left side of an arithmetic replacement statement. |
| ; | Variable appears in ENCODE or DECODE statement. |
| blank | Variable appears on the right side of an arithmetic replacement statement. |

R=1:

```
--VARIABLE MAP--

     I          I  U    117B                        J          I  U    120B
     K          I       121B                        L          I         0B /ANARRAY/  22
     M          I       122B                        MAXO       I            INTRINSIC
     MINO       I              INTRINSIC            NOHEAD     -        15B ENTRY
     OUTCI.     -              EXTERNAL.            OUTCR.     -            EXTERNAL.
     OUTPUTE    -              EXTERNAL.            PASCAL     -        66B ENTRY
     SIZE       I AU      0B
```

R=2, R=3:

```
--VARIABLE MAP--

     I          I  U    117B                        4 C    4 W      9 C   10
     J          I  U    120B                       12 C   13 S     13 S   13 S   14 C      14 S
     K          I       121B                       10 =   11 S     12 C   14 C
     L          I         0B /ANARRAY/   22         2 D    3 D     11 =   13     13        13 =    14 W
     M          I       122B                        8 =    9 C
     MAXO       I              INTRINSIC            8 A
     MINO       I              INTRINSIC            9
     NOHEAD     -        16B   ENTRY                7 C
     OUTCI.     -              EXTERNAL.            4 W   14 W
```

# OBJECT CODE 14

The structure of the object code produced by FORTRAN Extended differs depending on whether the compiler is operating in time-sharing mode (TS control statement option) or optimizing mode (OPT=0, 1, 2 control statement option). The format of the object code listing (if selected by the OL control statement option) also differs.

Both compilation modes produce object code in units called 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. The names of these blocks, as well as their exact contents, differ between the two compilation modes.

Also discussed in this section is the arrangement in memory of user code, library routines, and common blocks after the program is loaded.

## OPTIMIZING MODE

The following description of the arrangement of code and data within main program, subroutine and function program units 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 later in this section illustrates the position of blank common and labeled common blocks.

### SUBROUTINE AND FUNCTION STRUCTURE

The code within subprograms is arranged in the following blocks (relocation bases) in the order given.

| | |
|---|---|
| START. | Code for the primary entry and for saving A0 |
| VARDIM. | Address substitution code and any variable dimension initialization code |
| ENTRY. | Either a full word of NO's (46000. . . 46000B) or no storage used for this block |
| CODE. | Code generated by compiling: |
| | Executable statements |
| | Parameter lists for external procedure references within the current procedure |
| | Storage for compiler-generated temporary entities |
| DATA. | Storage for simple variables, FORMAT statements, and program constants |
| DATA.. | Storage for arrays other than those in common |

| | |
|---|---|
| HOL. | Storage for Hollerith constants |
| formal parameters | One local block for each dummy argument in the same order as they appear in the FUNCTION or SUBROUTINE statement, to hold tables used in address substitution for processing references to dummy arguments. |

## MAIN PROGRAM STRUCTURE

| | |
|---|---|
| START. | Input/output file buffers and a table of file names specified in the PROGRAM statement |
| CODE. | Transfer address code plus the code specified for the subroutine and function CODE. block |
| DATA.<br>DATA..<br>HOL. | Same as SUBROUTINE and FUNCTION structure |

## RENAMING CONVENTIONS

In optimizing mode, the names of some programmer defined and system supplied entities are changed so as to prevent ambiguity for the assembler.

### REGISTER NAMES

The compiler changes some legal FORTRAN names so that FORTRAN object code can be used as assembler input. When a two-character name begins with A, B, or X and the last character is 0 to 7, the compiler adds a dollar sign ($) to the name for the object code listing. (A0-A7, B0-B7, and X0-X7 represent registers that might be used by the FORTRAN Extended compiler.)

### EXTERNAL PROCEDURE NAMES

The name of a system supplied external procedure called by value is suffixed with a decimal point. The entry point is the symbolic name of the external procedure and a decimal point suffix. For example, EXP. COS. CSQRT. The names of all external procedures called by value are listed in table 8-2 (Basic External Functions). A procedure is not called by value and the name is not suffixed with a decimal point if it appears either in an EXTERNAL statement or an overriding type statement, or if option T, D, or OPT=0 is specified on the FTN control statement.

The call-by-name entry point is the symbolic name of the external procedure with no suffix. External procedures called by name appear in section 8 (Utility Subprograms). Any name which appears in table 8-1 (Intrinsic Functions) or table 8-2 (Basic External Functions) is called by name if it appears in an EXTERNAL statement or in an overriding type statement; those listed as Basic External Functions are also called by name if option T, D, or OPT=0 is specified on the FTN control statement.

### LISTING FORMAT

If object code is listed when the compiler is in optimizing mode, the code produced for each program unit is listed following the reference map (if any) for that program unit. If a LIST,NONE directive is in effect when the END line for that program unit is compiled, no object code is listed. Otherwise, the object code for the entire unit is listed, including code generated for lines that fall between LIST,NONE and LIST,ALL directives.

## TIME-SHARING MODE

The following blocks are used in the object code generated by the compiler operating in time-sharing mode for both main programs and subprograms.

| | |
|---|---|
| CODE | Code resulting from compilation of executable statements, and parameter list for current subprogram (not used in main programs) |
| LITERAL | Storage for constants of all kinds |
| FORMAT | Compressed versions of FORMAT declarations (interpreted at execution time) |
| TEMP | Compiler-generated temporary entities |
| ARG | Argument lists for external subprograms called in this program unit |
| NAMELIST | Argument lists for calls to NAMELIST input/output |
| VARIABLE | Storage for variables and arrays not declared in common or in ECS/LCM |
| BUFFER | Input/output buffers |

### LISTING FORMAT

When the compiler is in time-sharing mode, generated object code lines are grouped and listed immediately following the source lines that produced them. The number of lines listed at any one time is usually less than a program unit. If a LIST,NONE directive is in effect at the time that the compiler would normally list a group of object code, the entire group is not listed. Conversely, if no LIST,NONE directive is in effect at such a time, the entire group of object code is listed, even if some of the object lines were generated by source lines whose listing has been suppressed by intervening LIST,NONE directives. Therefore, the object code listed does not always correspond exactly to the source code listed when LIST,NONE directives are present in the program unit. The compiler in time-sharing mode always uses the FORTRAN Extended internal assembler to assemble generated object code.

In time-sharing mode, some of the generated object code lines are not listed, in order to make the listing easier to read. However, all the lines generated from executable statements in the source program are listed.

# EXECUTION CONTROL STATEMENT 15

Optional parameters can be included on the control statement that calls into execution a program compiled by FORTRAN Extended. 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 card 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 several kinds: alternate file names, print limit specification (PL), and Post Mortem Dump output and subscript limit specifications.

## ALTERNATE FILE NAME SPECIFICATION

The file names specified on the PROGRAM statement (INPUT, OUTPUT if the PROGRAM statement is omitted) are compiled into an internal file table within the body of the main program. The address of this table is passed to Q2NTRY (FTNRP2 if ER is specified on the FORTRAN control statement) at execution time.

The logical file name that appears in the file information table is determined in one of three ways:

1. If no file names are specified on the execution control statement, the logical file name is the file name in the PROGRAM statement.

   Example:

   ```
   FTN.
   LGO.
    .
    .
    .
   PROGRAM TEST1 (INPUT,OUTPUT,TAPE1,TAPE2)
   ```

   Contents of internal file table before execution of Q2NTRY:

   ```
   000...017
   ```

   Contents of internal file table and following addresses after execution of Q2NTRY:

   ```
   INPUT...fit address
   OUTPUT..fit address
   TAPE1...fit address
   TAPE2...fit address
   ```

   The logical file names in the file information table will be:

   ```
   INPUT
   OUTPUT
   TAPE1
   TAPE2
   ```

2. If the file names are specified on the execution control statement, the logical file name is the name specified there. A one-to-one correspondence exists between parameters on this statement and parameters in the PROGRAM statement.

The user should ensure that no two file information tables have the same logical file name after this process.

## 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. This parameter has the same form as the PL parameter specified at compilation-time on the FTN control statement. If specified on the execution control statement, it overrides the value specified either explicity or by default at compilation-time (section 10). This parameter may appear anywhere in the parameter list; it does not affect file name substitution.

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 its 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.

Examples:

LGO(PL=2000)

EXECUTE(,FILE1,OUTPUT,PL=1000,FILE2)    FILE2 is placed in internal file table

NOTE

The BACKSPACE, ENDFILE, and REWIND statements cause the line count for the print limit test to be reset to zero at that point in the program. For example, if a program had a PL=100 and 99 lines had been output when a BACKSPACE command was executed, the output line count would be reset to zero, allowing an additional 100 lines to be output before the print limit would be reached.

## POST MORTEM DUMP PARAMETERS

Two parameters can be included on the execution control statement to control Post Mortem Dump output and to specify limits on array subscripts.

The PMD output parameter specifies the destination and format of the dump. The parameter appears on the execution control statement in the following format:

*OP=list

The option list consists of one or more of the following, not separated by separators:

A Causes variables in all active routines to be included in the dump. An active routine is one that has been executed but is not necessarily in the traceback chain.

F Causes a full dump to be written to the file PMDUMP when the job is executed with the file OUTPUT connected. This option is valid for interactive jobs only.

T Causes a condensed form of the dump to be displayed at the terminal. File OUTPUT must be connected. This option is valid for interactive jobs only.

If the *OP parameter is omitted, dumps are sent to file PMDUMP when the job is executed from a terminal with file OUTPUT connected.

Example:

```
LGO(*OP=AF)
```

The PMD subscript limit parameter controls the printing of arrays by PMD. This parameter has the same effect as a CALL PMDARRY in a source program. The parameter appears on the execution control statement in one of the following formats:

```
*DA=i

*DA=i+j

*DA=i+j+k
```

In these formats, i, j, and k represent integers that specify the maximum values of the subscripts of arrays to be printed. The integers specified for i, j, and k apply to the first, second, and third dimensions, respectively.

Example:

```
LGO(*DA=2+5)
  .
  .
  .
DIMENSION RAY(20,20)
```

As a result of these statements, PMD will print the following elements of the array RAY:

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

# INPUT/OUTPUT IMPLEMENTATION 16

This section describes the structure of files read and written by FORTRAN Extended. All files read and written as a result of user requests at execution time are processed through CYBER 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 SCOPE 2 Record Manager when compilation is under SCOPE 2, and by operating system routines when compilation is under NOS 1 or NOS/BE 1.

## EXECUTION-TIME INPUT/OUTPUT

All input and output between a file referenced in a FORTRAN Extended program and the file storage device is under control of Record Manager. The version of Record Manager used depends on the operating system:

NOS 1 and NOS/BE 1 use CYBER Record Manager Basic Access Methods Version 1.5 (BAM), encompassing sequential and word addressable file organizations, for standard input/output statements, and CYBER Record Manager Advanced Access Methods Version 2 (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.

In this manual, the term CRM refers to features supported under BAM and AAM, but not under the SCOPE 2 Record Manager.

These versions of Record Manager normally appear the same to FORTRAN users; however, they do offer substantially different capabilities. Standard file organizations and record formats are defined to facilitate file interchange and access through different products.

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. This section 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 non-standard 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 may have records in one of 8 formats (record types) defined to Record Manager. Only 4 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.

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.

## STRUCTURE OF INPUT/OUTPUT FILES

FORTRAN Extended sets certain values in the file information table depending on the nature of the input/output operation and its associated file structure. Table 16-1 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 overridden by a FILE control statement.

‡ With READ and WRITE statements, the record type (RT) depends on whether the access is formatted or unformatted. 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.

§ 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 W§ type records unblocked for processing on unit record equipment.

BUFFER IN and BUFFER OUT assume S‡ -type or 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.

---

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2.

TABLE 16-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED

| FIT Fields | | Formatted, NAMELIST, & List-Directed READ/WRITE | Unformatted READ/WRITE | BUFFER IN/ BUFFER OUT | Mass Storage Input/Output |
|---|---|---|---|---|---|
| Meaning | Mnemonic | | | | |
| CIO buffer size (words) | (1) BFS‡ | 2002B | 2002B | 2002B | 2002B |
| Block type | BT | C‡/(9)§ | I‡/(9)§ | C‡/(9)§ | n/a |
| Close flag (positioning of file after close) | CF | N* | N* | N* | N‡/R§* |
| Length in characters of record trailer count field (T type records only) | CL | 0 | 0 | 0 | n/a |
| Conversion mode | CM | YES‡/NO§ | NO | (2) | n/a |
| Beginning character position of trailer count field, numbered from zero (T type records only) | CP | 0 | 0 | 0 | n/a |
| Length field (D type records) or trailer count field (T type records) is binary | C1‡ | NO | NO | NO | n/a |
| Type of information to be listed in dayfile | DFC | 3 | 3 | 3 | 3 |
| Type of information to be listed on error file | EFC | 0 | 0 | 0 | 0 |
| Error options | EO | AD | AD | AD | AD |
| Trivial error limit | ERL | 0 | 0 | 0 | 0 |
| Length in characters of an F or Z type record (same as MRL) | FL | 150 (5)* | n/a | n/a | n/a |
| File organization | FO | SQ * | SQ * | SQ * | WA * |
| Character length of fixed header for T type records | HL | 0 | 0 | 0 | n/a |
| Length of user's label area (number of characters) | (7) LBL | 0 * | 0 * | 0 * | n/a |

‡Applies only to NOS 1 and NOS/BE 1

§Applies only to SCOPE 2.

TABLE 16-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED (Contd)

| FIT Fields: Meaning | FIT Fields: Mnemonic | Formatted, NAMELIST, & List-Directed READ/WRITE | Unformatted READ/WRITE | BUFFER IN/ BUFFER OUT | Mass Storage Input/Output |
|---|---|---|---|---|---|
| Logical file name | LFN | (3) | (3) | (3) | (3) |
| Length in characters of record length field (D type records) | LL | 0 | 0 | 0 | n/a |
| Beginning character position of record length, numbered from zero (D type records) | LP | 0 | 0 | 0 | n/a |
| Label type | (7) LT | ANY | ANY | ANY | n/a |
| Maximum block length in characters | MBL | 0 | 0 | 0 | n/a |
| Minimum block length in characters | MNB ‡ | 0 | 0 | 0 | n/a |
| Minimum record length in characters | MNR ‡ | 0 | 0 | 0 | n/a |
| Maximum record length in characters | (5) MRL | n/a | $2^{23}-1$ | (8) * | n/a |
| Multiple of characters per K, E type block | MUL ‡ | 2 | 2 | 2 | n/a |
| Open flag (positioning of file after open) | (7) OF | N* | N* | N* | N‡/R§ * |
| Padding character for K, E type blocks | PC ‡ | 76B | 76B | 76B | n/a |
| Processing direction | PD | IO | IO | IO | IO |
| Number of records per K type block | RB | 1 | 1 | 1 | n/a |

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2.

TABLE 16-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED (Contd)

| FIT Fields | | Formatted, NAMELIST, & List-Directed READ/WRITE | Unformatted READ/WRITE | BUFFER IN/ BUFFER OUT | Mass Storage Input/Output |
|---|---|---|---|---|---|
| Meaning | Mnemonic | | | | |
| Record mark character (R records) | RMK | 62B | n/a | 62B | n/a |
| Record type | RT | Z‡/W§(10) | W (6) | S‡/W§ | U |
| Length field (D type records) or trailer count field (T type records) has sign overpunch | SB ‡ | NO | NO | NO | n/a |
| Suppress buffering | SBF ‡ | NO* | NO* | YES(11) | NO* |
| Suppress read ahead | SPR | NO | NO | NO | n/a |
| Character length of trailer portion of T type records | TL | 0 | 0 | 0 | n/a |
| User label processing | (7) ULP | NO | NO | NO | NO |
| End of volume flag (positioning of file at volume CLOSEM time) | VF | U | U | U | U |

**Notes:**

n/a FIT field not applicable to this input/output mode.

* Default cannot be overridden by a FILE control statement.

(1) Default can be changed by PROGRAM statement. FILE control statement can specify a value smaller than the value established by the program, but the buffer location remains unchanged. If BFS=0, Record Manager allocates a new buffer and computes an appropriate length.

(2) Set by parity designator in BUFFER IN or BUFFER OUT statement.

(3) Set by PROGRAM statement or execution control statement (section 15).

(4) Set by Record Manager.

(5) Default can be changed on PROGRAM statement. For formatted, NAMELIST, and list-directed READ/WRITE statements, a FILE control statement can decrease but not increase the maximum record length.

(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 8) 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.

‡Applies only to NOS 1 and NOS/BE 1.
§Applies only to SCOPE 2.

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 8) 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.

## 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 16-1.

Format of the FILE card is:

FILE(lfn,field=value, . . . )

| | |
|---|---|
| lfn | 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 statement. |
| field | FIT field mnemonic |
| value | Symbolic or integer value |

---

[†]Record type W was written through FORTRAN Extended Version 4.2. Existing files with RT=W are recognized and processed correctly under subsequent versions of FORTRAN Extended without user action.

The FILE control statement may appear anywhere in the control statements prior to program execution, but it must not interrupt a load set.

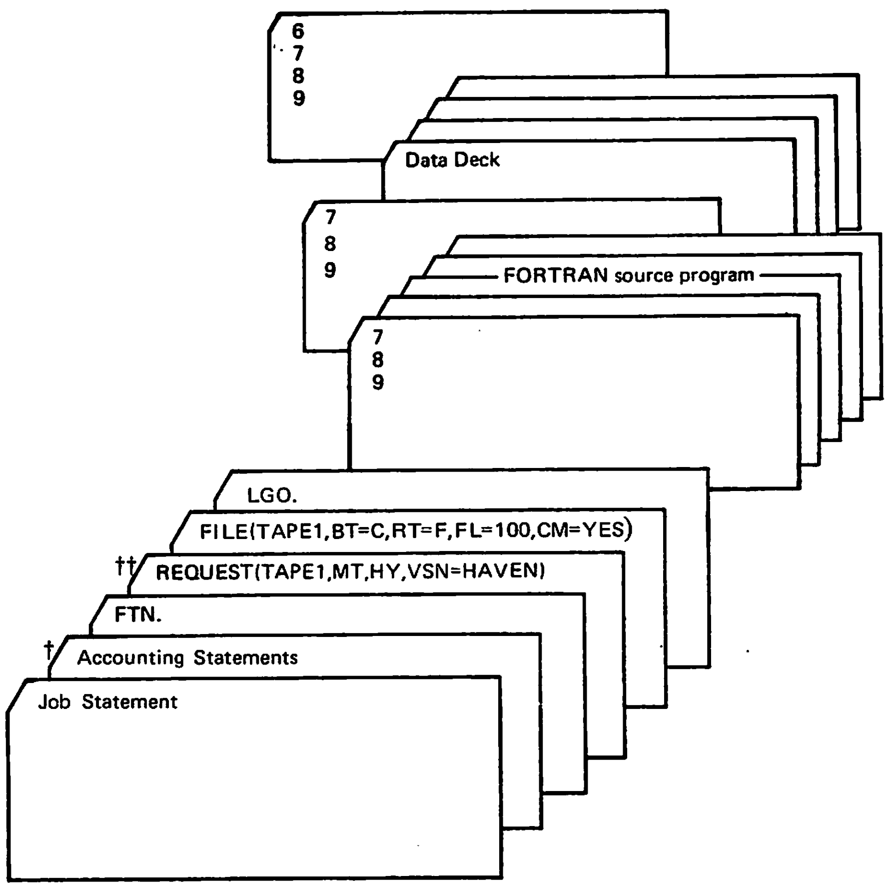
This deck illustrates the use of the FILE control statement to override default values supplied by the FORTRAN compiler. Assuming the source program is using formatted writes and 100-character records are always written, the file is written on magnetic tape with even parity, at 800 bpi. No labels are 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




† As required by the operating system.

†† Format applicable to NOS/BE 1.

## SEQUENTIAL FILE OPERATIONS

### 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.

The following table details the actions performed prior to positioning.

‡

| Condition | Device Type | Action |
|---|---|---|
| Last operation was WRITE or BUFFER OUT | Mass Storage | Any unwritten blocks for the file are written.<br>An end-of-partition is written.<br>If record format is W, a deleted zero length record is written. |
| | Unlabeled Magnetic Tape | Any unwritten blocks for the file are written.<br>If record format is W, a deleted zero length record is written.<br>Two file marks are written. |
| | Labeled Magnetic Tape | Any unwritten blocks for the file are written.<br>If record format is W, a deleted record is written.<br>A file mark is written.<br>A single EOF label is written.<br>Two file marks are written. |

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2.

| Condition | Device Type | Action |
|---|---|---|
| Last operation was WRITE or BUFFER OUT | Mass Storage | ENDFILE is issued.<br>Any unwritten blocks for the file are written. End-of-information is flagged in RBT chain. |
| | Unlabeled Magnetic S or L Tape | ENDFILE is issued.<br>Any unwritten blocks for the file are written.<br>Two file marks are written. |
| | Labeled Magnetic Tape or Unlabeled System Magnetic Tape | ENDFILE is issued.<br>Any unwritten blocks for the file are written.<br>A tape mark is written.<br>A single EOF label is written.<br>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 | § Magnetic Tape | If the file is assigned to on-line magnetic tape, a REWIND request is executed.<br>§ If the file is staged, the REWIND request has no effect. The file is staged and rewound when it is first referenced. |
| | § Mass Storage<br>‡ All Devices | REWIND request causes the file to be rewound when first referenced. |
| Previous operation was REWIND | | Current REWIND is ignored. |

§

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2.

### ENDFILE

The following table indicates 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.

‡

| Record Type | Device Type | |
|---|---|---|
| | S or L Tape | Other Device |
| W | An end-of-partition flag is written.<br>The block is terminated. | An end-of-partition flag is written.<br>The block is terminated with a short PRU of level 0. |
| Other | The block is terminated.<br>A tape mark is written. | The block is terminated with a short PRU of level 0.<br>A zero length PRU of level 17 is written. |

§

| Record Type | Blocking | |
|---|---|---|
| | Blocked | Unblocked |
| W | An end-of-partition flag is written.<br>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.<br>A tape mark recovery control word is written. | Ignored. |
| Others on Magnetic Tape | The block is terminated.<br>A tape mark is written. | Not applicable. |

‡Applies only to NOS 1 and NOS/BE 1.

§Applies only to SCOPE 2

## INPUT/OUTPUT RESTRICTIONS

Meaningful results are not guaranteed in the following circumstances:

1. Mixed formatted and unformatted read/write statements and buffer input/output statements on the same file (without an intervening REWIND, ENDFILE, or without encountering an End of File (EOP) as determined by the EOF Function).

2. Requesting a LENGTH function or LENGTHX call on a buffer unit before requesting a UNIT function.

3. Two consecutive buffer input/output statements on the same file without the intervening execution of a UNIT function call.

4. Failing to close a mass storage input/output file with an explicit CLOSMS in an overlay program that is STATICly loaded.

5. Writing formatted records on a seven-track S or L tape without specifying CM=NO on a FILE control statement.

6. Using items in an input/output list after encountering end-of-file in a read.

7. 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.

## RECORD MANAGER ERROR SUPPRESSION

For formatted, namelist, and list directed sequential reads (coded), MRL has a default size of 150 characters or the record length specified in the PROGRAM statement. If the record read from the file exceeds this value, Record Manager automatically reports an RM142 error condition to the FORTRAN input/output routines and sends a message to the job dayfile. FORTRAN senses that the RM142 error condition is not fatal to the execution of the program and allows the program to continue executing. The user should ignore the RM142 error condition in the dayfile.

# 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.

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 the other operating systems, the compiler makes direct calls to the operating system for input/output; CRM is not used.

The structure of the text files read by the compiler is described in the COMPASS Version 3 reference manual. SCOPE 2 structure is identified in the tables below by the equivalent SCOPE 2 Record Manager parameters.

## SOURCE INPUT FILE STRUCTURE

A source input file must have the following structure. Only the first 90 characters of each record are processed or reproduced in the listing output file.

| File Characteristics | NOS/BE 1 and NOS 1 | 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) |

## CODED OUTPUT FILE STRUCTURE

Two coded output files may be produced: the listing file and the file of COMPASS line images. Format is as follows:

| File Characteristics | NOS/BE 1 and NOS 1 | 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) |

## BINARY OUTPUT FILE STRUCTURE

The format of the executable object code file is as follows: (the content of the file differs, depending on the loader supported by the operating system)

| File Characteristics | NOS/BE 1 and NOS 1 | SCOPE 2 |
|---|---|---|
| File organization | Sequential operating system default format with file terminated by a zero length PRU which is then backspaced 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) |

# COMPASS SUBPROGRAM LINKAGE 17

Both subroutines and functions may be written in COMPASS assembly language and 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.

When a FORTRAN generated subprogram is called, the calling routine must not depend on values being preserved in any registers other than A0.

## CALL BY NAME AND CALL BY VALUE

To increase speed, arguments to library functions are normally passed to subprograms by placing their values in registers. This method is call by value. For user defined subprograms, the addresses of the arguments are passed to the subprogram. This method is call by name.

### CALL BY NAME SEQUENCE

The FORTRAN compiler uses the call by name sequence when a subroutine or function name differs from any of those listed in tables 8-1 and 8-2. Call by name is also used when a listed subroutine or function also appears in an EXTERNAL or overriding type statement, or (except in the case of intrinsic functions) the program unit specifies D, T, or OPT=0 on the FTN control statement.

The call by name sequence generated is shown below:

SA1 Address of the argument list (if parameters appear)
The list contains the addresses of the arguments passed to and returned from the subprogram.

+RJ Subprogram name

-VFD 12/line number, 18/trace word address

| | |
|---|---|
| line number | Source line number of statement containing the reference |
| trace word address | Address of the trace word for the calling routine |

Arguments in the call must correspond with the argument usage in the called routine, and they must reside in the same level.

The argument list consists of consecutive words in the following form followed by a word of binary zeros. The sign bit will be set in the argument list for any argument entry address that is LCM or ECS.

VFD 60/address of argument

When the RETURNS list form is used, the list of return addresses is located immediately after a word of binary ones which follows the argument list. The RETURNS list is terminated by a word of binary zeros. The subroutine accesses the addresses by offsetting the address of the argument list, which is contained in register A1.

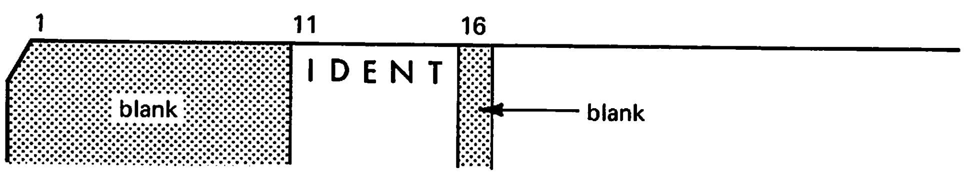
## CALL BY VALUE SEQUENCE

For increased efficiency the compiler generates a call by value code sequence for references to library functions if the function name does not appear in an EXTERNAL or overriding type statement and (in the case of external functions only) the D, T, or OPT=0 options on the FTN control statement are not specified. The name of any library function called by value or generated in line must appear in an EXTERNAL statement in the calling routine if the call by name calling sequence is required (section 8 lists the library functions called by value and generated in-line).

The call by value code sequence consists of code to load the arguments into X1 through X4, followed by an RJ instruction to the function. Two registers are used for each double precision or complex argument.

# INTERMIXED COMPASS SUBPROGRAMS

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 1 through 10 blank, and column 16 blank:




The subprogram ends with any legal COMPASS END line. A COMPASS subprogram cannot interrupt a FORTRAN program unit; it must be placed after the END line of the FORTRAN program unit and before the beginning of the next program unit. A COMPASS subprogram can also be the first or last program unit in a source deck.

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 function reference, the subprogram must return the function result in X6 or X6 and X7 with the least significant or imaginary part of the double precision or complex result appearing in X7.

The COMPASS assembler normally requires the system text SYSTEXT, which is the default for the S parameter. The amount of storage available depends on installation options. Insufficient storage for SYSTEXT causes an error. The user may need to specify a larger field length for compilation or a different option for S. See the COMPASS reference manual and section 10 of this manual for more details on systems texts.

Example:

This example shows a simple COMPASS function and the calling FORTRAN main program. The parity function, PF, returns an integer value; therefore it must be declared integer in the calling program. The argument to PF may be either real or integer.

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 $400\ 000_8$ 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.

**Source Deck**

```
job card
MAP(OFF)
FTN(R=0)
LGO.
7/8/9 in column 1.
      PROGRAM NPSAMP(OUTPUT)
      INTEGER PF, PVAL(24)
      DO1I=1,24
1     PVAL(I)=PF(I)
      PRINT2,(I,I=1,24),PVAL
2     FORMAT(32H0INTEGERS 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          ENTRY/EXIT...
            SA2     X1 <------------------------- get the argument value
            CX3     X2 <------------------------- count the 1 bits in X2 and leave result in X3
            MX0     -1 <------------------------- form a mask in X0
            BX6     -X0*X3          ISOLATE LOWEST BIT <-- put result into X6
            EQ      PF              EXIT..

            END
6/7/8/9 in column 1.
```

(Lines from PROGRAM NPSAMP through END: main program)

**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
```

## ENTRY POINT

For subprograms written in FORTRAN, the compiler uses the following conventions in generating code:

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.

| | | |
|---|---|---|
| Trace word: | VFD | 42/name, 18/entry address |
| A0 word: | DATA | 0 |
| Entry point: | DATA | 0 |

## RESTRICTIONS ON USING LIBRARY FUNCTION NAMES

Functions written in FORTRAN that have library function names listed in tables 8-1 or 8-2, such as AMAX1 or SQRT, must be declared EXTERNAL in the calling program unit. This declaration is necessary because the compiler produced functions always use the call by name calling sequence.

Functions written in COMPASS that have basic external library names listed in table 8-2, such as SQRT, should be written using the call by name sequence when they are declared EXTERNAL in the calling routine; or they should use the call by value rules if they are not declared EXTERNAL.

Functions written in COMPASS that have intrinsic library names listed in table 8-1, such as AMAX1, must be declared EXTERNAL in the calling routine; otherwise in-line coding is generated for them (the COMPASS coding is ignored). Furthermore, the call by name sequence must be used.

If a library function, called by value, is to be overridden by a routine coded in COMPASS, the COMPASS routine must use the library function name with a period appended as the entry point name (e.g., SIN.) to use the call by value calling sequence.

The following sample illustrates the code generated for: a library function call, SQRT; an external function call, ZEUS; and a reference to an intrinsic (in-line) function, AMAX1.

The coding generated for the external function, ZEUS, is illustrated also.

```
MAP(OFF)
FTN(R=0.OL)
7/8/9 in column 1
          PROGRAM SUBLNK
          X=SQRT(7.0)
          Y=ZEUS(X,1.0)
          END
          FUNCTION ZEUS(ARG1,ARG2)
          ZEUS=AMAX1(ARG1,ARG2,0.)
          RETURN
          END
6/7/8/9 in column 1
```

```
                                                            IDENT     SUBLNK
                                                         USEBLK
                                                         LDSET   LIB=FORTRAN
                                                         USE START.

                                    000000  000005  START.     LOCAL
                                    000005  000000  VARDIM.    LOCAL
                                    000005  000000  ENTRY.     LOCAL
                                    000005  000011  CODE.      LOCAL
                                    000016  000004  DATA.      LOCAL
                                    000022  000000  DATA..     LOCAL
                                    000022  000000  HOL.       LOCAL

                                   EXTERNALS
                                 END.       ZEUS       SQRT.      Q2NTRY.

                                                       LIBLNK. BSS 0B
000000 START.    20000000000000000003                    LIBLNK   0B,11610B
000001 START.    00000000000000011610
000002 START.    00000000000000000000
                                                       FILES. BSS 0B
000003 START.    00000000000000000000                    DATA 0
000004 START.    23250214161355000005                    TRACE    SUBLNK,SUBLNK
                                                         USE CODE.
                                                         PENTRY  SUBLNK,,,0
000005 CODE.     5110000000             START.           SA1 LIBLNK.
                           0100000000 <EXT>              RJ Q2NTRY.
                                                         USE DATA.
                                                         USE DATA..
                                                         USE DATA.
000016 DATA.                                           CON. BSS 0B
000016 DATA.     17227000000000000000                    DATA 17227000000000000000B } constant table
000017 DATA.     17204000000000000000                    DATA 17204000000000000000B }
                                                         EXT      END.
                                                         EXT      ZEUS
                                                         EXT      SQRT.
                                                         EXT      Q2NTRY.
000020 DATA.                                           X        BSS 1B
000021 DATA.                                           Y        BSS 1B             source line number
                                                                  USE      CODE.
                                                       *                          LINE    2
000006 CODE.     5110000016             DATA.                     SA1      CON. <---get actual parameter
                           0100000000 <EXT>                       RJ       SQRT.     into X1
000007 CODE.     5160000020             DATA.                     SA6      X
                           5110000013 CODE.                       SA1      [AP1<---get address of parameter
000010 CODE.     0100000000             <EXT>          +          RJT      ZEUS.3R   list into A1
                           0003000004
000011 CODE.     5160000021             DATA.                     SA6      Y
                           5110000004 STAR-.                      SA1      TRACE.
000012 CODE.     0400000000             <EXT>                     EQ       END.
000013 CODE.                                            [AP1      BSS      0B     }
000013 CODE.     00000000000000000020 DATA.                       APL      X      } parameter address list
000014 CODE.     00000000000000000017 DATA.                       APL      CON.+1R}
000015 CODE.     00000000000000000000                             APL             }
                                                        Z.        END      SUBLNK
```

```
                                        IDENT     ZEUS
                                   USEBLK
                                   LDSET   LIB=FORTRAN
                                   USE START.


                  000000   000006  START.    LOCAL
                  000006   000000  VARDIM.   LOCAL
                  000006   000000  ENTRY.    LOCAL
                  000006   000005  CODE.     LOCAL
                  000013   000001  DATA.     LOCAL
                  000014   000000  DATA..    LOCAL
                  000014   000000  HOL.      LOCAL
                  000014   000000  ARG1      LOCAL
                  000014   000000  ARG2      LOCAL


              EXTERNALS
           SPA.

                                   USE DATA.
                                   USE START.
000000 START.    32052523555555000004      TRACE   ZEUS,ZEUS,2B
000001 START.    00000000000000000000
000002 START.    51400000131064446000}     PENTRY  ZEUS,ENTRY.,1,0
000003 START.    51300000015203000000}
000004 START.    04004000046100046000
000005 START.    74600540105160000001
                                   FORPAR  ARG1
                                   FORPAR  ARG2
                                   USE DATA..
                                   USE DATA.
                                   EXT     SPA.
000013 DATA.                      VALUE.  BSS 1B
                                           USE      CODE.
                                  *                          LINE    2
000006 CODE.     54500                     SA5      A0
                      5040000001           SA4      A0+1B
                                53350      SA3      X5
000007 CODE.     53240                     SA2      X4
                      31032                FX0      X3-X2
                           13723           BX7      X2-X3
                                21073      AX0      73B
000010 CODE.     11670                     BX6      X7*X0
                      13063                BX0      X6-X3
                           22700           LX7      B0,X0
                                21773      AX7      73B
000011 CODE.     11670                     BX6      X7*X0
                      13760                BX7      X6-X0
                           5170000013 DATA.  SA7    VALUE.
000012 CODE.     0400000002      START.    EQ       EXIT.
000014 ARG2                           Z.   END
```

name of program unit and entry point address

cell to save A0 in

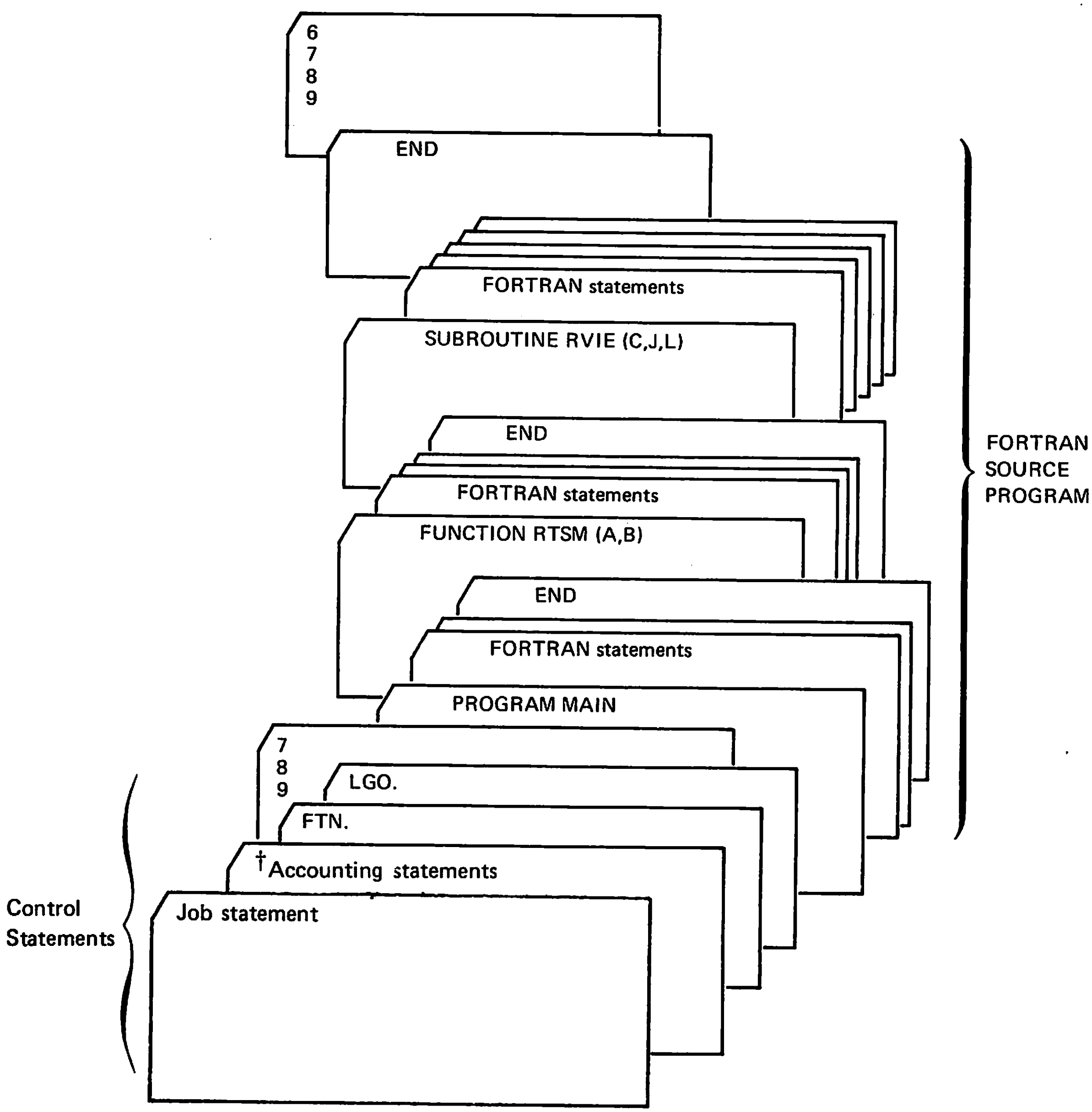
restores A0 on exit

entry point

saves A0 and sets A0 to the new A1

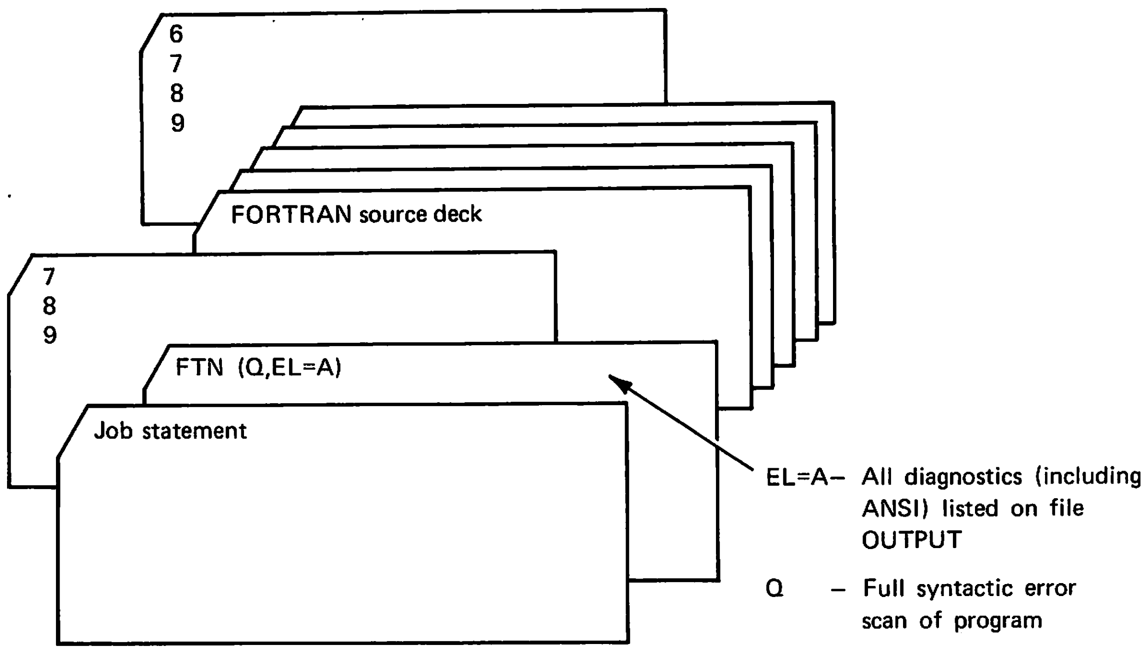
## FORTRAN SOURCE PROGRAM WITH CONTROL STATEMENTS

Refer to the operating system reference manual for details of control statements.

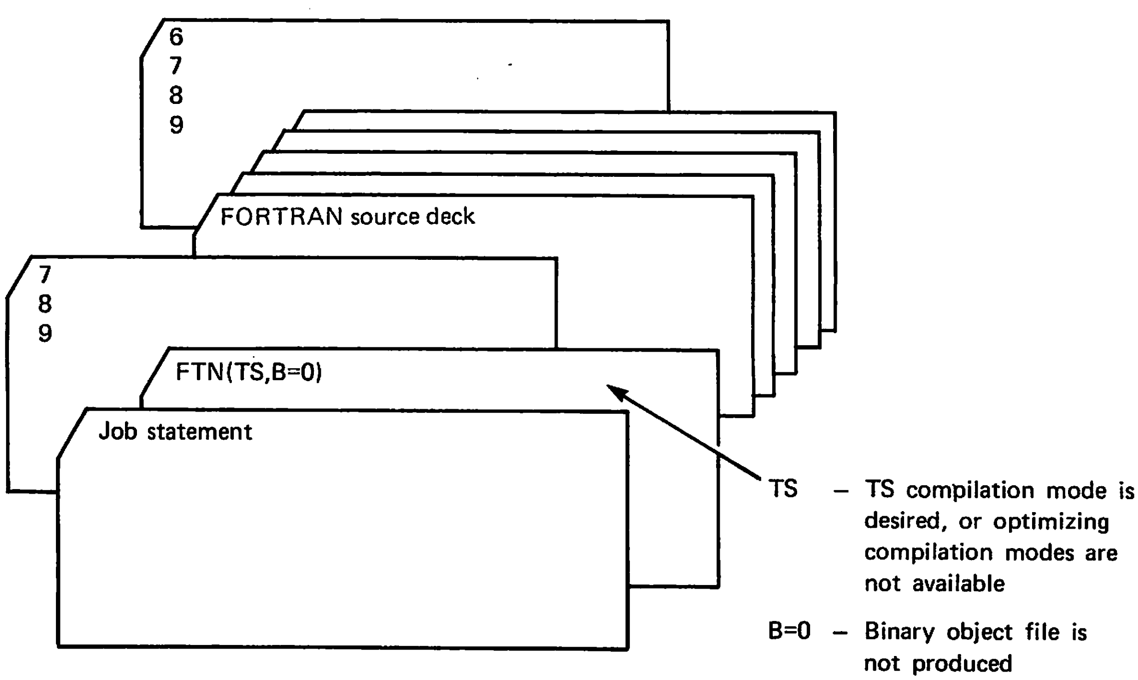



†As applicable for operating system or installation.

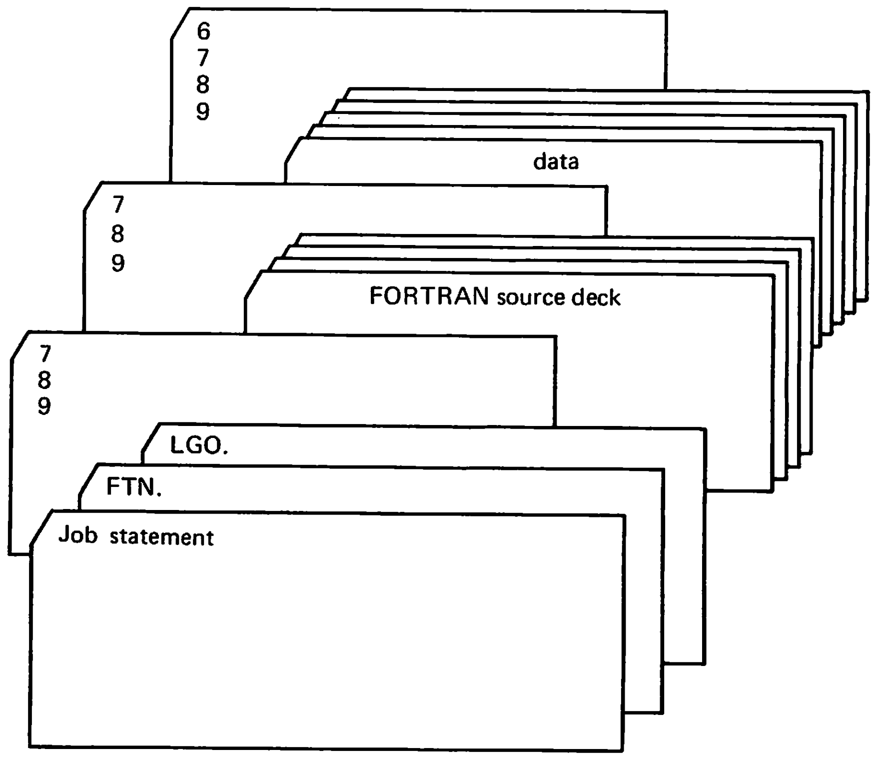
COMPILATION ONLY




TS MODE COMPILATION ONLY

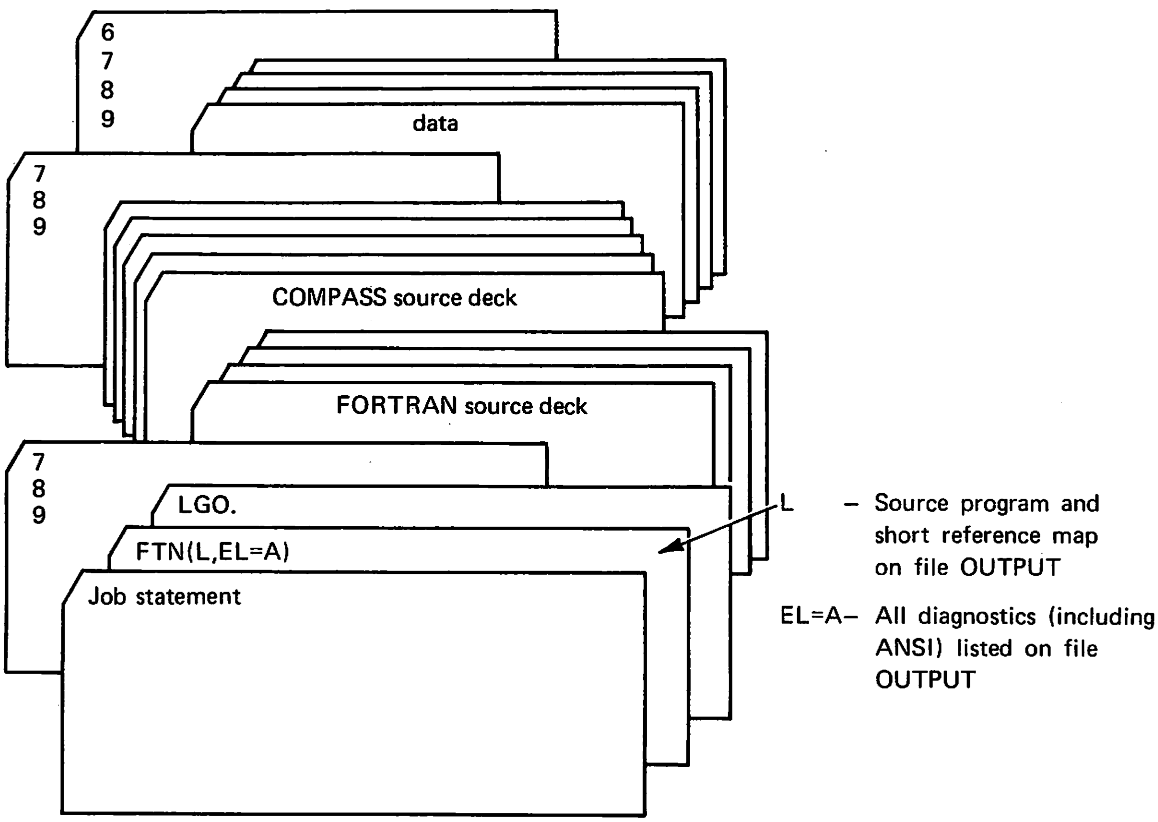


## COMPILATION AND EXECUTION

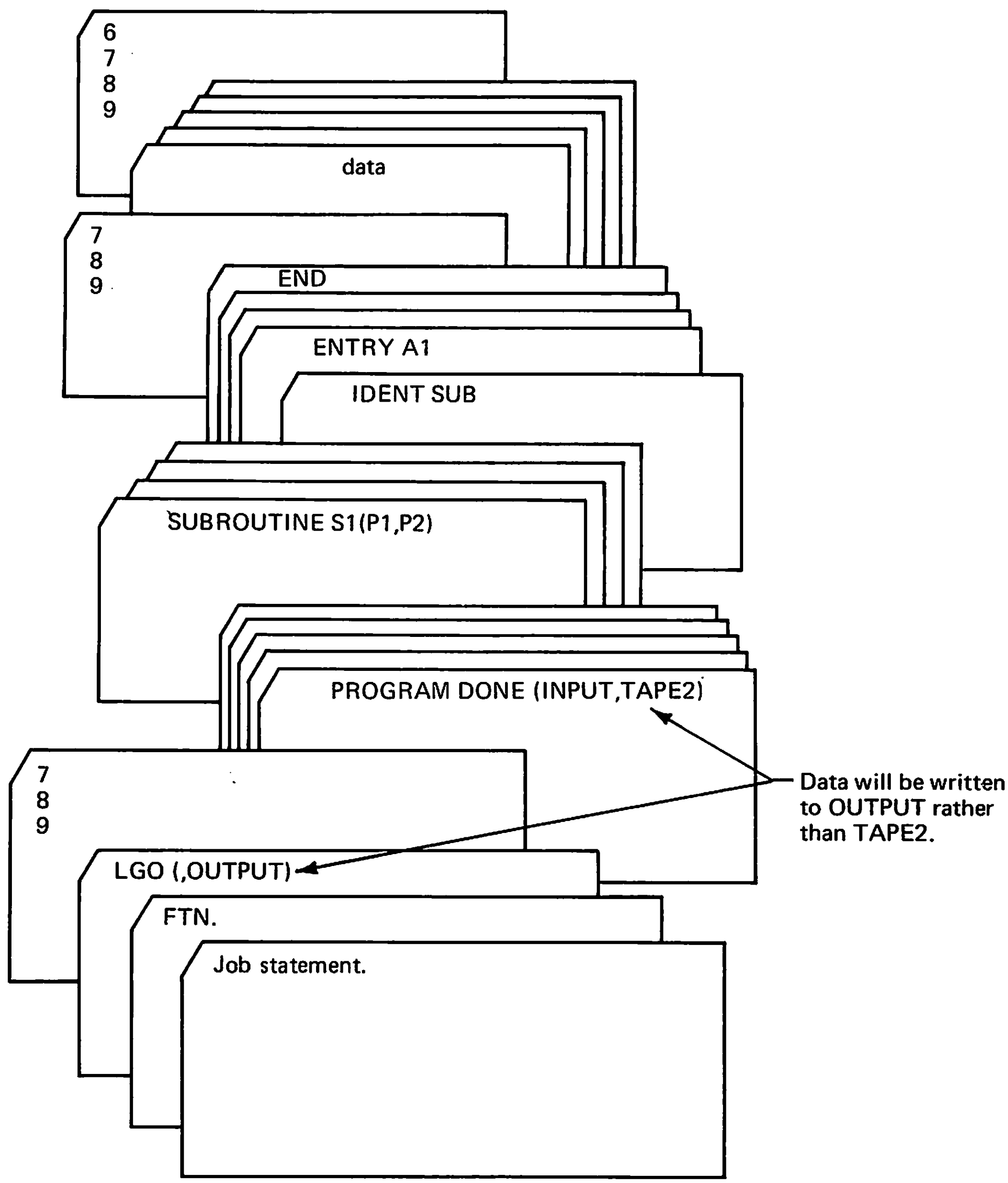


## FORTRAN COMPILATION WITH COMPASS ASSEMBLY AND EXECUTION

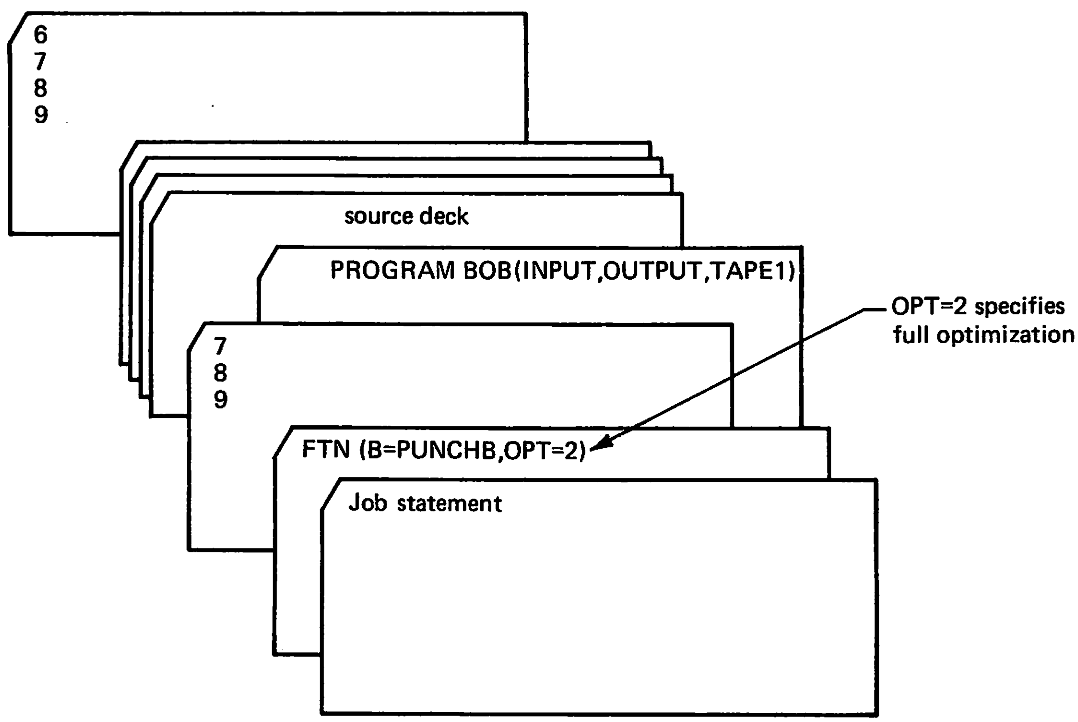
FORTRAN and COMPASS program unit source decks can be in any order. COMPASS source decks must begin with a line containing the word IDENTb in columns 11-16. Columns 1-10 of the IDENT line must be blank.



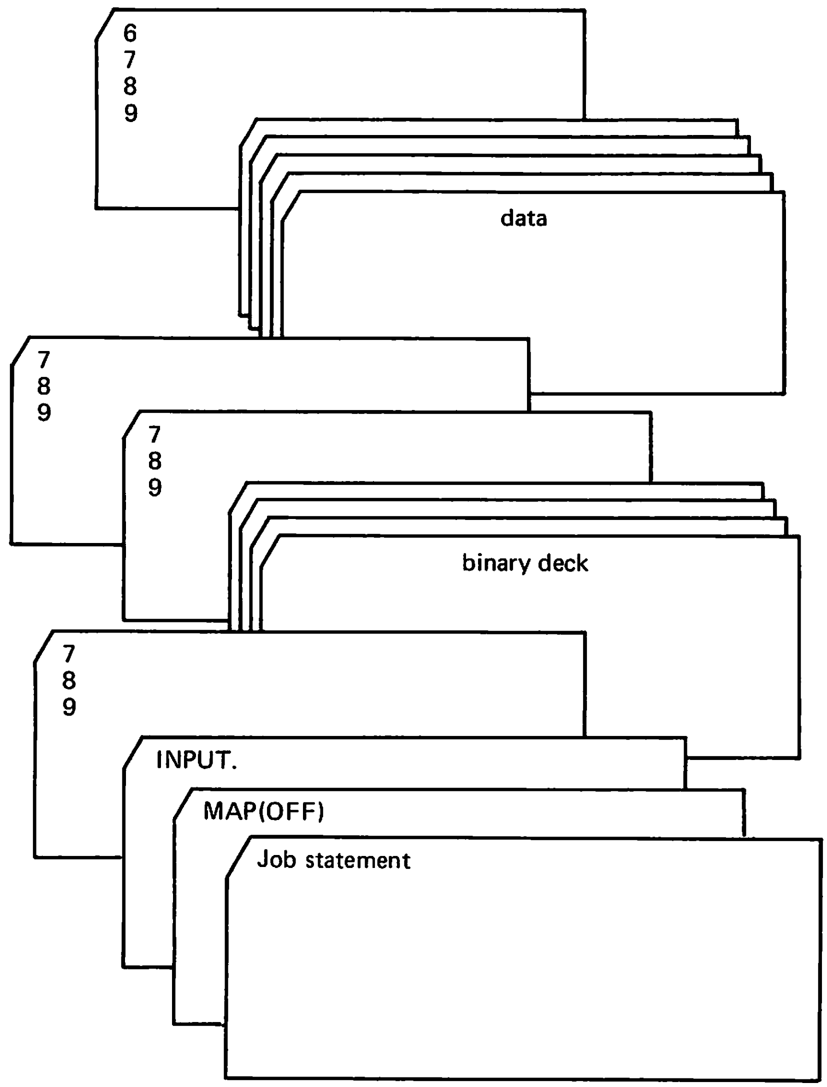
## COMPILE AND EXECUTE WITH FORTRAN SUBROUTINE AND COMPASS SUBPROGRAM



## COMPILE AND PRODUCE BINARY CARDS

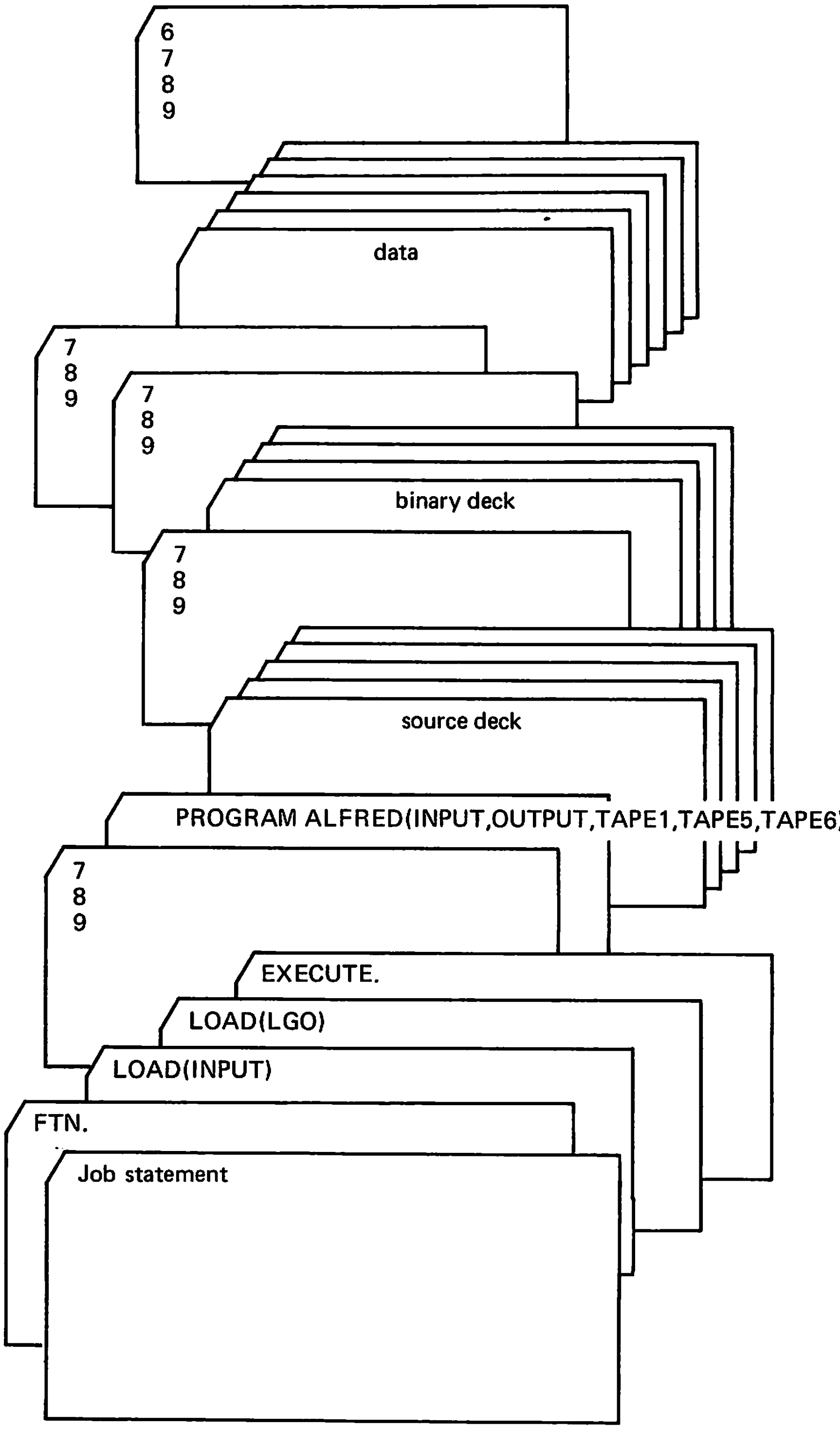


## LOAD AND EXECUTE BINARY PROGRAM†



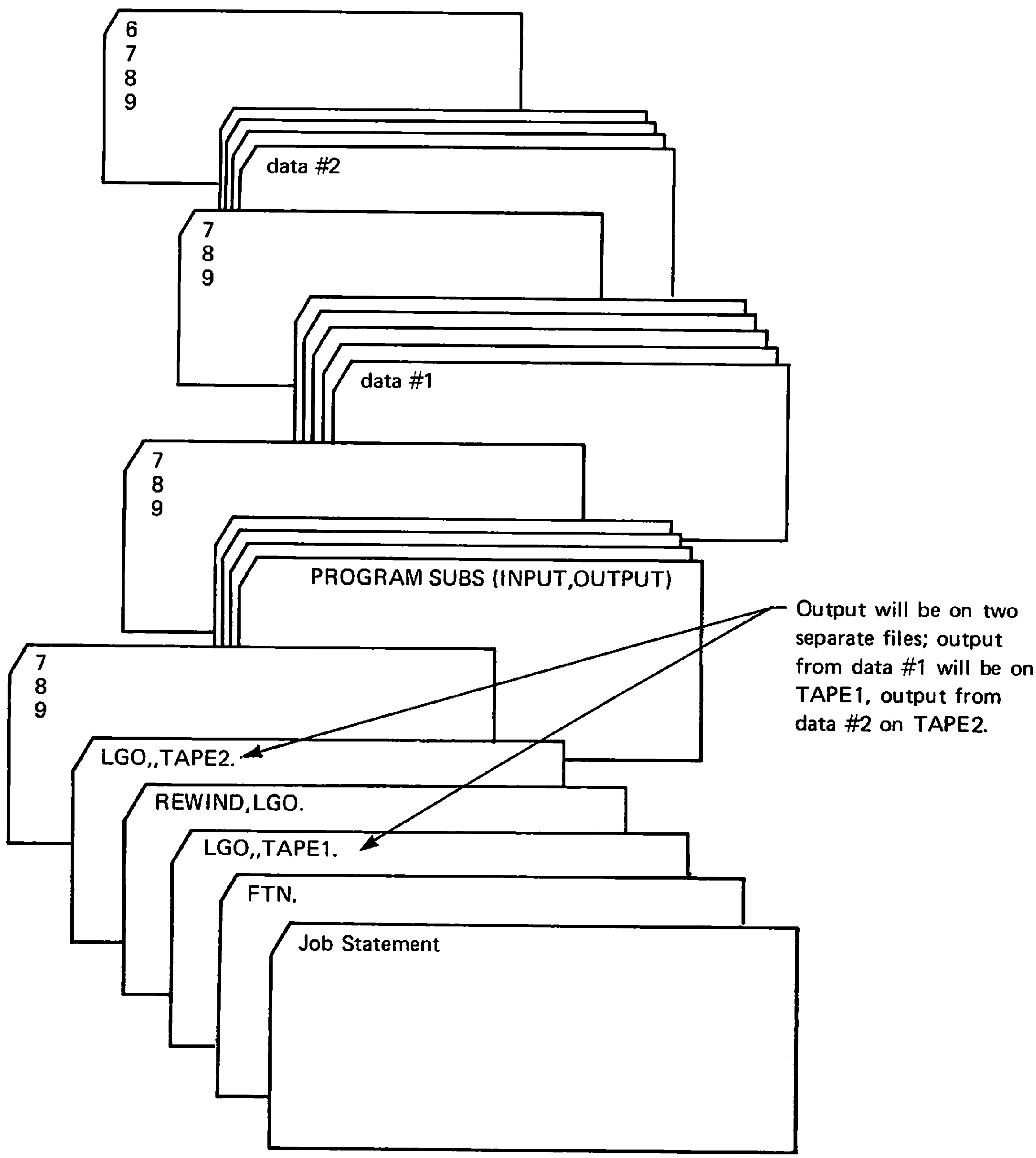

†Under NOS 1, a 6/7/9 card, instead of two 7/8/9 cards, must follow the binary deck to signify end-of-input to the loader.

## COMPILE AND EXECUTE WITH RELOCATABLE BINARY DECK†

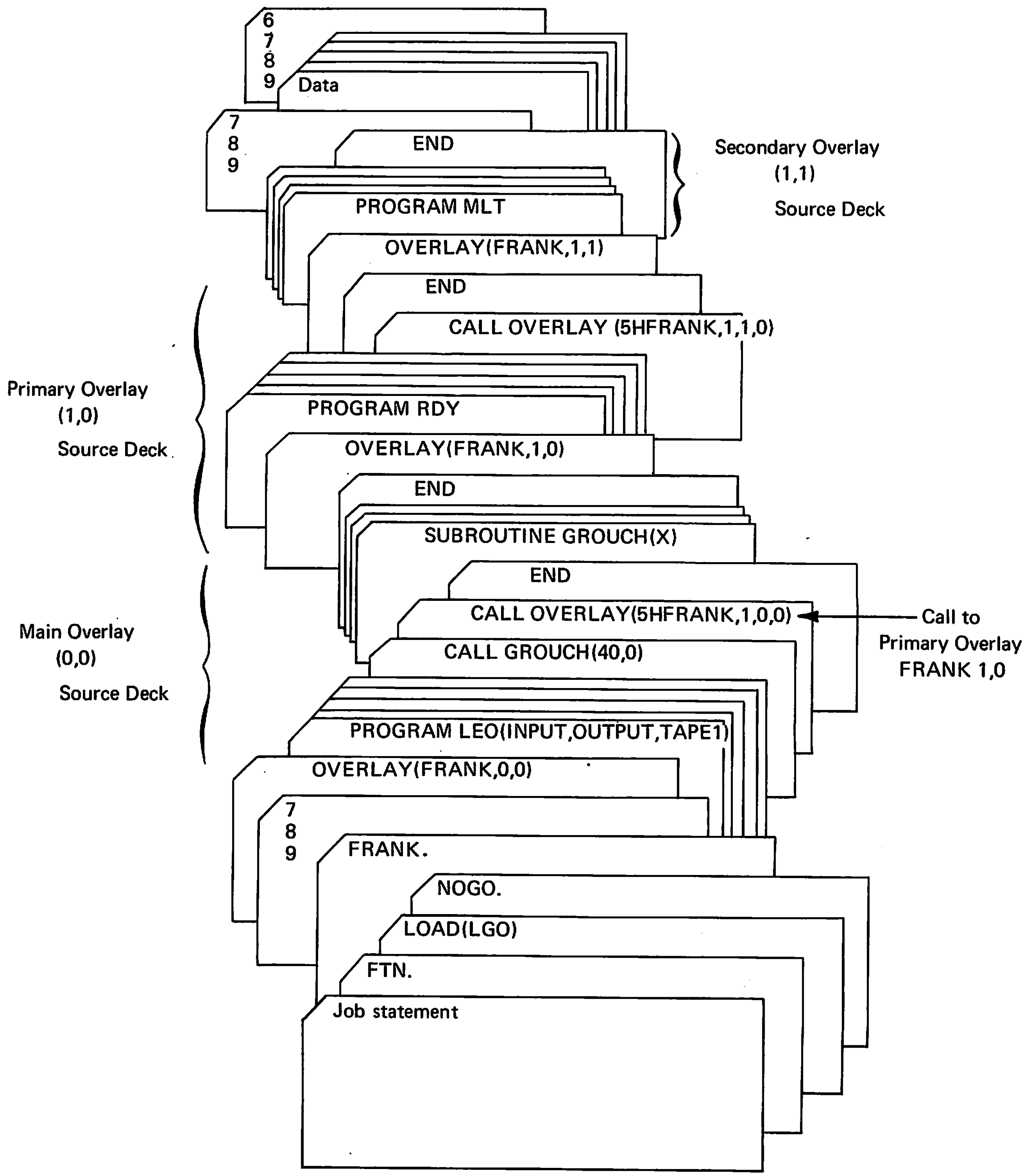



†Under NOS 1, a 6/7/9 card, instead of two 7/8/9 cards, must follow the binary deck to signify end-of-input to the loader.

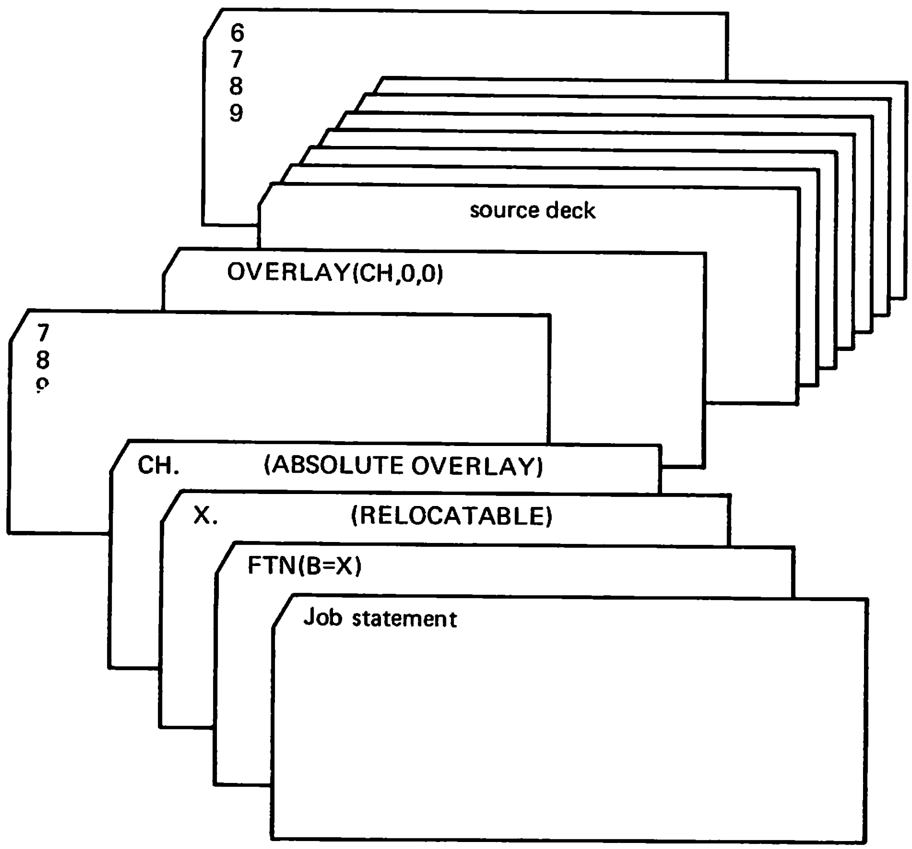
## COMPILE ONCE AND EXECUTE WITH DIFFERENT DATA DECKS



## PREPARATION OF OVERLAYS



## COMPILATION AND 2 EXECUTIONS WITH OVERLAYS



## PROGRAM OUT

Program OUT illustrates the WRITE and PRINT statements.

Features:

Control statements for batch execution

WRITE and PRINT statements

Carriage control

PROGRAM statement

```
PAT,T10
```

The job statement must precede every job. PAT is the job name. T10 specifies a maximum of 10 (octal) seconds central processor time.

```
FTN.
```

Specifies the FORTRAN Extended compiler and uses the default parameters. (section 10)

```
LGO.
```

The binary object code is loaded and executed.

If no alternative files are specified on the FTN control statement, the FORTRAN Extended compiler reads from the file INPUT and outputs to two files: OUTPUT and LGO. Listings, diagnostics, and maps are output to OUTPUT and the relocatable object code to LGO.

```
7/8/9
```

The 7/8/9 card separates control statements from the remainder of the job deck (INPUT file). This card contains a multipunched 7, 8, and 9 in column 1; it follows control statements in every batch job.

```
PROGRAM OUT (OUTPUT,TAPE6=OUTPUT)
```

The PROGRAM statement identifies the main program by the name OUT and specifies the file OUTPUT. Logical unit 6 will be referenced in the program. All files used by FORTRAN input/output statements in a program must be specified in the PROGRAM card of the main program.

TAPE6=OUTPUT is included because unit number 6 is referenced in a WRITE statement. The unit number will be prefixed by the letters TAPE. All data written to TAPE6 will be placed in the file OUTPUT and eventually output to the printer.

```
WRITE (6,200) INK
```

The WRITE statement outputs the variable INK to TAPE6. If a PRINT statement had been used instead of WRITE:

```
PRINT 200, INK
```

TAPE6=OUTPUT would not be needed in the PROGRAM card; PROGRAM OUT (OUTPUT) would be sufficient.

```
100 FORMAT (*1 THIS WILL PRINT AT THE TOP OF A PAGE*)
```

This FORMAT statement uses * * to delimit the literal. 1 is a carriage control character which causes the line to be printed at the top of a page.

```
200 FORMAT (I5,* = INK OUTPUT BY WRITE STATEMENT*)
```

Although the variable INK is 4 digits, a specification of I5 is given because the first character is always interpreted as carriage control. In this case, the carriage control character is a blank and output will appear on the next line.

```
6/7/8/9
```

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 this particular job.

Complete Job Deck:

```
PAT,T10
FTN.
LGO.
7/8/9 in column 1
      PROGRAM OUT (OUTPUT,TAPE 6=OUTPUT)
      PRINT 100
  100 FORMAT (*1 THIS WILL PRINT AT THE TOP  OF A PAGE*)
      INK = 2000+4000
      WRITE (6,200) INK
  200 FORMAT (I5,* = INK OUTPUT BY WRITE STATEMENT*)
      PRINT 300, INK
  300 FORMAT (1H ,I4,30H = OUTPUT FROM PRINT STATEMENT)
      STOP
      END
6/7/8/9 in column 1
```

Output:

```
 THIS WILL PRINT AT THE TOP  OF A PAGE
6000 = INK OUTPUT BY WRITE STATEMENT
6000 = OUTPUT FROM PRINT STATEMENT
```

## PROGRAM B

Program B generates a table of 64 characters indicating the character set being used. The internal bit configuration of any character can be determined by its position in the table. Each character occupies six bits.

Features:

- Octal constants
- Simple DO loop
- PRINT statement
- FORMAT with H,/,I,X and A elements

```
NCHAR= 00 01 02 03 04 05 06 07 00 00B
```

The print statement PRINT1 has no input/output list; it prints out the heading at the top of the page using the information provided by the FORMAT statement labeled 1. 25H specifies a Hollerith field of 25 characters, 1 is the carriage control character, and the two slashes cause one line to be skipped before the next Hollerith field is printed. The slash at the end of the FORMAT specification skips another line before the program output is printed.

```
DO 3 I=1,8
J=I-1
```

These statements generate numbers 0 through 7 (a DO index cannot be a zero).

```
PRINT 2, J, NCHAR
```

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 4. The octal value 01 is a display code A, 02 is a B, 03 is a C, etc.

```
NCHAR=NCHAR + 10 10 10 10 10 10 10 10 00 00B
```

The octal constant 10101010101010100000B is added 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.

Program:

```
      PROGRAM B (OUTPUT)
      PRINT 1
    1 FORMAT(25H1TABLE OF INTERNAL VALUES//12H     01234567,/)
      NCHAR= 00 01 02 03 04 05 06 07 00 00B
      DO 3 I = 1,8
      J=I-1
      PRINT 2, J,NCHAR
    2 FORMAT(I3,1X,A8)
3     NCHAR=NCHAR+10 10 10 10 10 10 10 10 00 00B
      STOP
      END
```

Output:

```
 TABLE OF INTERNAL VALUES

     01234567

  0  :ABCDEFG
  1  HIJKLMNO
  2  PQRSTUVW
  3  XYZ01234
  4  56789+-*
  5  /()$= ,.
  6  ≡[]%≠→∨∧
  7  ↑↓<><≥¬;
```

# PROGRAM MASK

Program MASK reads 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.

Feature:

Masking

```
1  FORMAT (1H1,5X,4HNAME,///)
   PRINT 1
```

The printer is directed to start a new page, print the heading NAME, and skip 3 lines.

```
3  READ 2,LNAME,FNAME,ISTATE,KSTOP
   IF(KSTOP.EQ.1)STOP
```

The last name is read into LNAME, first name into FNAME, and home state into ISTATE. The last record contains a one which will be read into KSTOP as a stop indicator. The IF statement on line 6 tests for the stop indicator.

```
 IF((ISTATE.AND.7777000000000000000OB).NE.(2HCA.AND.777700000000000
KOOOOOB))  GO TO 3
```

The relational operator .NE. tests to determine if the first two letters read into variable ISTATE match the two letters of the Hollerith constant CA. The last eight characters (48 bits) in ISTATE are masked and the two remaining characters are compared with the word containing the Hollerith constant CA, also similarly masked. If the bit string forming one word is not identical to the bit string forming the other word, ISTATE is not equal to CA and the IF statement test is true.

The bit configuration of CALIFORNIA, the Hollerith constant CA and the mask follows:

California

| Hollerith | C | A | L | I | F | O | R | N | I | A |
|---|---|---|---|---|---|---|---|---|---|---|
| Octal | 03 | 01 | 14 | 11 | 06 | 17 | 22 | 16 | 11 | 01 |
| Bit | 000011 | 000001 | 001100 | 001001 | 000110 | 001111 | 010010 | 001110 | 001001 | 000001 |

Constant CA

| Hollerith | C | A | blank | blank | blank | blank | blank | blank | blank | blank |
|---|---|---|---|---|---|---|---|---|---|---|
| Octal | 03 | 01 | 55 | 55 | 55 | 55 | 55 | 55 | 55 | 55 |
| Bit | 000011 | 000001 | 101101 | 101101 | 101101 | 101101 | 101101 | 101101 | 101101 | 101101 |

Mask

| Octal | 77 | 77 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
|---|---|---|---|---|---|---|---|---|---|---|
| Bit | 111111 | 111111 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 |

When the masking expression (ISTATE.AND.77770000000000000000B) is completed, the first two characters of CALIFORNIA remain the same and last eight characters are zeroed out. The AND operation follows:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 000011 | 000001 | 001100 | 001001 | 000110 | 001111 | 010010 | 001110 | 001001 | 000001 |
| 111111 | 111111 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 |
| 000011 | 000001 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 | 000000 |

When (2HCA.AND.77770000000000000000B) is evaluated, the same result is obtained. Thus, in both words, all bits but those forming the first two characters will be masked, making a valid basis for comparing the first two characters of both words. If the result of the mask is true, the last name and first name are printed (statement 10), otherwise the next record is read.

Program:

```
      PROGRAM MASK (INPUT,OUTPUT)
1     FORMAT (1H1,5X,4HNAME,///)
      PRINT 1
2     FORMAT (3A10,I1)
3     READ 2,LNAME,FNAME,ISTATE,KSTOP
      IF(KSTOP.EQ.1)STOP

C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD

      IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.777700000
     K00000000000B))  GO TO 3
11    FORMAT(5X,2A10)
10    PRINT 11,LNAME,FNAME
      GO TO 3
      END
```

Data records:

```
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
                                  1
```

Output:

```
 NAME



BROWN,     PHILLIP M.
CROWN,     SYLVIA
MUNCH,     GARY G.
SMITH      SIMON
```

## PROGRAM EQUIV

Program EQUIV places values in variables that have been equivalenced and prints these values using the NAMELIST statement.

Features:

EQUIVALENCE statement

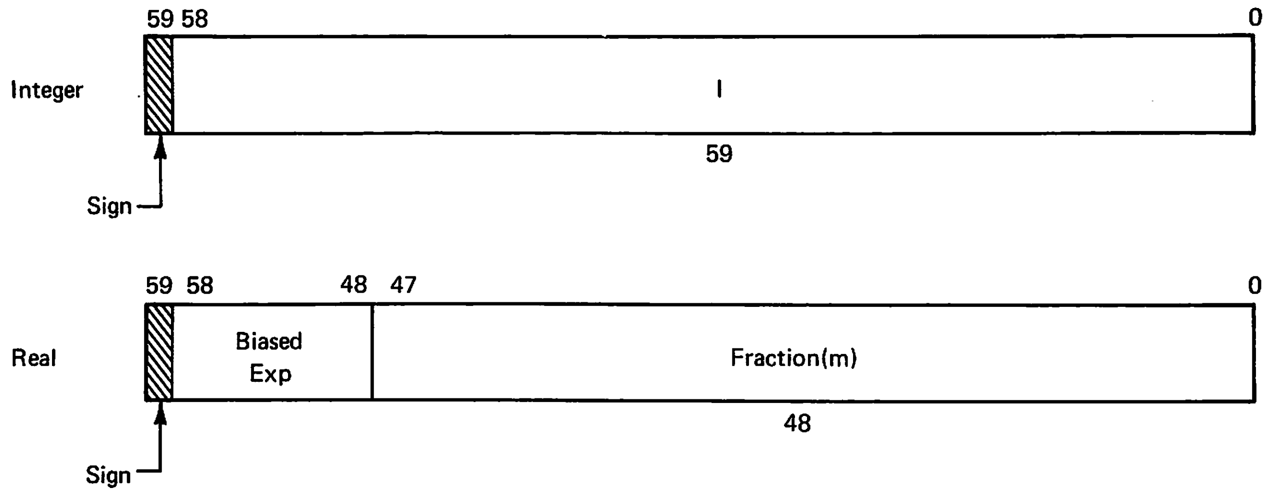
NAMELIST statement

```
EQUIVALENCE (X,Y),(Z,I)
```

Two real variables X and Y are equivalenced; the two variables share the same location in storage, which can be referred to as either X or Y. Any change made to one 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. (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 format of real and integer values differ.




Although they 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 will be small.

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 may be obtained.

```
WRITE(6,OUTPUT)
```

This NAMELIST WRITE statement outputs both the name and the value of each member of the NAMELIST group OUTPUT defined in the statement NAMELIST/OUTPUT/X,Y,Z,I. The NAMELIST group is preceded by the group name, OUTPUT, and terminated by the characters $END.

Program:

```
      PROGRAM EQUIV (OUTPUT,TAPE6=OUTPUT)
      EQUIVALENCE (X,Y),(Z,I)
      NAMELIST/OUTPUT/X,Y,Z,I
      X=1.
      Y=2.
      Z=3.
      I=4
      WRITE(6,OUTPUT)
      STOP
      END
```

Output:

```
$OUTPUT

X         = .2E+01,

Y         = .2E+01,

Z         = 0.0,

I         = 4,

$END
```

## PROGRAM COME

Program COME places variables and arrays 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 A and outputs values in common using the NAMELIST statement.

Features:

COMMON and EQUIVALENCE statements

NAMELIST statement

```
COMMON A(1),B,C,D, F,G,H
```

Variables are stored in common in the order of appearance in the COMMON statement A(1),B,C,D,F,G,H. Variables can be dimensioned in the COMMON statement; and in this instance, A is dimensioned so that it can be subscripted later in the program. If A were not dimensioned, it could not be used as an array in statement 1.

```
INTEGER A,B,C,D,E(3,4),F,H
```

All variables with the exception of G are declared integer. G is implicitly typed real.

```
EQUIVALENCE(A,E,I)
```

The EQUIVALENCE statement assigns the first element of the arrays A and E and an integer variable I to the same storage location. Since A is in common, E and I will be in common. Variables and array elements are assigned storage as follows:

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(1) | B | C | D | F | G | H | | | | | |
| | A(2) | A(3) | A(4) | A(5) | A(6) | A(7) | A(8) | A(9) | A(10) | A(11) | A(12) |

```
   DO 1 J=1,12
1  A(J)=J
```

The DO loop places values 1 through 12 in array A. The first element of array A shares the same storage location with the first element of array E. Since B is equivalent to E(2,1), A(2) is equivalent to B, A(3) to C, A(4) to D, etc.

Any change made to one member of an equivalence group changes the value of all members of the group. When 1 is stored in A, both E(1,1) and I have the value 1. When 2 is stored in A(2), B and E(2,1) have the value 2. Although B and E(2,1) are not explicitly equivalenced to A(2), equivalence is implied by their position in common.

The implied equivalence between the array elements and variables is illustrated by the output.

```
NAMELIST/V/A,B,C,D,E,F,G,H,I
```

The NAMELIST statement is used for output. A NAMELIST group, V, containing the variables and arrays A,B,C,D,E,F,G,H,I 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 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).

Program:

```
      PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
      COMMON A(1),B,C,D,   F,G,H
      INTEGER A,B,C,D,E(3,4),F,   H
      EQUIVALENCE (A,E,I)
      NAMELIST/V/A,B,C,D,E,F,G,H,I

      DO 1 J = 1, 12
    1 A(J)=J

      WRITE (6,V)
      STOP
      END
```

Output:

```
$V
A          =  1,
B          =  2,
C          =  3,
D          =  4,
E          =  1,   2,   3,   4,   5,   6,   7,   8,   9,   10,   11,   12,
F          =  5,
G          =  0.0,
H          =  7,
I          =  1,
$END
```

## PROGRAM LIBS

Program LIBS illustrates library subroutines provided by FORTRAN Extended.

Features:

EXTERNAL used to pass a library subroutine name as a parameter to another library routine.

Division by zero.

LEGVAR used to test for overflow or divide error conditions.

Library functions used:

LOCF

LEGVAR

Library subroutines used:

DATE

TIME

SECOND

RANGET

DATE is a library subroutine which returns the date entered by the operator from the console. DATE is declared external because it is used as a parameter to the function LOCF. Declaring DATE external does not prevent its use as a library subroutine in this program.

```
      PRINT 2,TODAY,CLOCK
2     FORMAT( *1TODAY=*, A10, * CLOCK=*  ,A10)
```

These statements print the date and time. The leading and trailing blanks appear with the 10 alphanumeric characters returned by the subroutine DATE because the operator typed in the date this way. The value returned by TIME is changed by the system once a second, and the position of the digits remain fixed; a leading blank always appears. The format of DATE and TIME can be checked by observing any listing, as the routines DATE and TIME are used by the compiler to print out the date and time at the top of compiler output listings.

```
CALL SECOND(TYME)
```

When SECOND is called, the variable name TYME is used. A variable name cannot be spelled the same as a program unit name. If Program LIBS had not called the subroutine TIME, a variable name could be spelled TIME.

```
LOCATN=LOCF(DATE)
```

DATE is not a variable name as it appears in an EXTERNAL statement.

Library function LOCF returns the address of DATE.

```
CALL RANGET(SEED)
```

Library subroutine RANGET returns the seed used by the random number generator RANF if it is called. 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.

```
      PRINT 3, TYME, LOCATN, LOCATN, SEED, SEED
3     FORMAT(*OTHE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*//* LOCATION OF
     1 DATE ROUTINE IS=*,O15,* OR*,I7,* IN DECIMAL.*/*OTHE INITIAL VALUE
     2 OF THE RANF SEED IS *,O22,*, OR*,G30.15,* IN G30.15 FORMAT.*)
```

These statements print out the values returned by the routines SECOND, LOCF, and RANGET.

Asterisks are used to delineate Hollerith fields in the format specification to illustrate the point that excessive use of asterisks can be extremely difficult to follow.

```
Y=0.0
WOW=7.2/Y
IF(0.NE. LEGVAR(WOW))PRINT4,WOW
```

These statements 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.

The line of -*-* on the output is produced by the FORMAT specification in statement number 4: 50(2H*-).

Program:

```
      PROGRAM LIBS (OUTPUT)
C
      EXTERNAL DATE
C
      CALL DATE (TODAY)
      CALL TIME (CLOCK)

      PRINT 2, TODAY, CLOCK
    2 FORMAT(*1TODAY=*, A10, * CLOCK=*, A10)
C
      CALL SECOND(TYME)
      LOCATN=LOCF(DATE)
      CALL RANGET(SEED)

      PRINT 3,TYME, LOCATN, LOCATN, SEED, SEED
    3 FORMAT(*0THE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*//* LOCATION OF
     1 DATE ROUTINE IS=*,O15,* OR*,I7,* IN DECIMAL.*/*0THE INITIAL VALUE
     2 OF THE RANF SEED IS*,O22,*, OR*,G30.15,* IN G30.15 FORMAT.*)
C
      Y=0.0
      WOW=7.2/Y
      IF(0 .NE. LEGVAR(WOW))PRINT4,WOW
      STOP
    4 FORMAT(1H0,50(2H*-)/* DIVIDE ERROR, WOW PRINTS AS=*,G10.2)
      END
```

Output:

```
TODAY= 07/31/74  CLOCK= 15.47.33.

THE ELAPSED CPU TIME IS    1.0030     SECONDS.

LOCATION OF DATE ROUTINE IS=000000000005347 OR   2791 IN DECIMAL.

THE INITIAL VALUE OF THE RANF SEED IS  17171274321477413155. OR         .170996394044023     IN G30.15 FORMAT.

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
DIVIDE ERROR, WOW PRINTS AS=         R
```

## PROGRAM PIE

Program PIE calculates an approximation of the value of $\pi$.

Feature:

Library 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.

```
DATA CIRCLE,DUD/2*0.0/
```

The DATA statement initializes the variables CIRCLE and DUD with the value 0.0.

Each time the DO loop is iterated, a random number, uniformly distributed over the range 0 through 1, is returned by the library function RANF, and this value is stored in the variable X. The value of X will be $0 \leq X < 1$. DUD is a dummy argument which must be used when RANF is called.

```
Y=RANF(DUD)
```

RANF is referenced again; this time to obtain a value for Y.

```
IF(X*X+Y*Y.LE.1.)CIRCLE=CIRCLE+1.
```

The IF statement and the arithmetic expression 4.*CIRCLE/10000. calculate an approximation of the value of $\pi$. The value of $\pi$ is calculated using Monte Carlo techniques. The IF statement counts those points whose distance from the point (0., 0.) is less than one. The ratio of the number of points within the quarter circle to the total number of points approximates 1/4 of $\pi$. The value PI is printed by the NAMELIST statement WRITE(6,OUT)

Program:

```
      PROGRAM PIE(OUTPUT,TAPE6=OUTPUT)
      DATA CIRCLE,DUD/2*0.0/
      NAMELIST/OUT/PI

      DO 1 I = 1,10000
      X=RANF(DUD)
      Y=RANF(DUD)
      IF(X*X+Y*Y.LE.1.)CIRCLE=CIRCLE+1.
1     CONTINUE

      PI=4.*CIRCLE/10000.
      WRITE(6,OUT)

      STOP
      END
```

Output:

```
$OUT

PI      = .315E+01,

$END
```

# PROGRAM ADD

Program ADD illustrates the use of the DECODE statement. The ENCODE and DECODE statements are simpler to understand when related to the READ and WRITE statements.

Features:

DECODE statement.

## DECODE (READ)

A READ statement 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 DECODE, the array specified in the DECODE statement is used as the input buffer. The number of words moved to the input list from the array is determined by the record length.

With the READ statement, 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 the DECODE statement, when the FORMAT statement indicates a new record is to be processed (by a slash or final right parenthesis), the next part of the array is used as the input buffer. The record length indicates the number of words to move into the array.

## ENCODE (WRITE)

A WRITE statement 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 statement, and placed in the output buffer. When the FORMAT statement indicates an 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 ENCODE statement is processed by compiler routines in the same way as the WRITE statement, but with the array specified within the parentheses of the ENCODE statement used as the output buffer. The number of words per record in the array is determined by the record length.

The number of computer words in each ENCODE or DECODE record is determined by dividing the record length by 10 and rounding up. For example, a record length of 33 requires 4 words, and a record length of 71 requires 8 words.

In the following program, the format of data on input is specified in column 1. If column 1 is a one, each of the remaining columns is a data item. If column 1 is a two, each pair of the remaining columns is a data item. If column 1 is a three or greater, each triplet of the remaining columns is a data item. Based on the information in column 1, the correct DECODE statement (the proper format and item count) is selected. The program then totals and prints out the items in each input record.

```
      INTEGER CARD(8),IN(79),TOTAL
```

CARD is dimensioned 8 to receive the 79 characters in columns 2 through 80. IN is dimensioned 79 to receive the numeric values of the input items.

```
10    READ(5,11)KEY,CARD
11    FORMAT(I1,7A10,A9)
```

The first column of the record is read into KEY under I format, and the remaining 79 characters are read into the array CARD under A format; so they can be converted later to I format with a DECODE statement.

```
      IF(EOF(5).NE.0)STOP
```

Tests for the end of data in which case the program simply stops.

```
      KEY=MAX0(1,MIN0(KEY,3))
```

Guarantees that $1 \leqslant \text{KEY} \leqslant 3$.

```
40    TOTAL=0
      DO41I=1,N
41    TOTAL=TOTAL+IN(I)
```

Adds up the items and leaves the total in TOTAL.

```
      WRITE(6,12)TOTAL,N,KEY,CARD,(IN(I),I= 1,N)
12    FORMAT(/I6,20H IS THE TOTAL OF THE ,I3,20H NUMBERS ON THE CARD/
     1I2,7A10,A9/16H THE NUMBERS ARE/(20I4))
```

Outputs the results.

```
      GOTO10
```

Goes back to process the next record.

Program:

```
      PROGRAM ADD
     1(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      INTEGER CARD(8),IN(79),TOTAL
10    READ(5,11)KEY,CARD
11    FORMAT(I1,7A10,A9)
      IF(EOF(5).NE.0)STOP
      KEY=MAX0(1,MIN0(KEY,3))
      GOTO(1,2,3),KEY
1     DECODE(79,91,CARD)IN
91    FORMAT(79I1)
      N=79
      GOTO40
2     DECODE(78,92,CARD)(IN(I),I=1,39)
92    FORMAT(39I2)
      N=39
      GOTO40
3     DECODE(78,93,CARD)(IN(I),I=1,26)
93    FORMAT(26I3)
      N=26
40    TOTAL=0
      DO41I=1,N
41    TOTAL=TOTAL+IN(I)
      WRITE(6,12)TOTAL,N,KEY,CARD,(IN(I),I= 1,N)
12    FORMAT(/I6,20H IS THE TOTAL OF THE ,I3,20H NUMBERS ON THE CARD/
     1I2,7A10,A9/16H THE NUMBERS ARE/(20I4))
      GOTO10
      END
```

Input:

```
21322554766988775533210332245666877965541233322112365478965412365547896541236028
30214456699877456632214455666655233655222144455663325566699885666554778854887029
55566663223666552332214455666998877655222144455611223303324456669988774558896030
10234566688899887789965554444556665533222111233023333669985555222114444777885031
```

Output:

```
 1900 IS THE TOTAL OF THE 39 NUMBERS ON THE CARD
21322554766988775533210332245666877965541233322112365478965412365547896541236028
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
30214456699877456632214455666655233655222144455663325566699885666554778854887029
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
35566663223666552332214455666998877655222144455611223303324456669988774558896030
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
10234566688899887789965554444556665533222111233023333669985555222114444777885031
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
```

## PROGRAM PASCAL

Program PASCAL produces a table of binary coefficients (Pascal's triangle).

Features:

- Nested DO loops
- DATA statement
- Implied DO loop

```
INTEGER L(11)
```

L is defined as an 11-element integer array.

```
DATA L(11)/1/
```

The DATA statement stores the value 1 in the last element of the array L. When the program is executed L(11) has the initial value 1.

```
PRINT 4,(I,I=1,11)
```

This statement prints the headings. The implied DO loop generates the values 1 through 11 for the column headings.

```
PRINT 3,(L(J),J=K,11)
```

This is a more complicated example of an implied DO loop. The index value J is used as a subscript instead of being printed. The end of the array is printed from a variable starting position. The 1, which appears on the diagonal in the output is not moving in the array; it is always in L(11); but the starting point is moving.

```
DO 2 I=1,10
K=11-I
```

These statements illustrate the technique of going backwards through an array. As I goes from 1 to 10, K goes from 10 to 1. The increment value in a DO statement must be positive, therefore,

```
DO 2 I=1,10
K=11-I
```

provides a legal method of writing the illegal statement DO 2 K=10,1,-1.

```
   DO 1 J=K,10
1  L(J)=L(J)+L(J+1)
```

This inner DO loop generates the line of values output by statement number 2. When control reaches statement 2, the variable J can be used again because statement number 2 is outside the inner DO loop. However, if I were used in statement 2 instead of J, the statement 2 PRINT 3,(L(I),I=K,11) would be an error. Statement 2 is inside the outer DO loop and would change the value of the index from within the DO loop. Changing the value of a DO index from inside the loop is illegal and will cause a fatal error or a never ending loop.

Program:

```
      PROGRAM PASCAL (OUTPUT)
      INTEGER L(11)
      DATA L(11) /1/
C
      PRINT 4, (I,I=1,11)
4     FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/
     S11I5)
      DO 2 I = 1,10
      K=11-I
      L(K)=1
      DO 1 J = K,10
    1 L(J)=L(J)+L(J+1)
    2 PRINT 3,(L(J),J=K,11)
    3 FORMAT(11I5)

      STOP
      END
```

Output:

```
COMBINATIONS OF M THINGS TAKEN N AT A TIME.

                    -N-
    1    2    3    4    5    6    7    8    9   10   11
    2    1
    3    3    1
    4    6    4    1
    5   10   10    5    1
    6   15   20   15    6    1
    7   21   35   35   21    7    1
    8   28   56   70   56   28    8    1
    9   36   84  126  126   84   36    9    1
   10   45  120  210  252  210  120   45   10    1
   11   55  165  330  462  462  330  165   55   11    1
```

## PROGRAM X

Program X references a function EXTRAC which squares the number passed as an argument.

Features:

Referencing a function

Function type

Program X illustrates that a function type must agree with the type associated with the function name in the calling program.

```
K=EXTRAC(7)
```

Since the first letter of the function name EXTRAC is E, the function is 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.

The argument 7 is 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 49 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 defined integer, in the main program it is defined real. Information which the main program needs regarding a subprogram must be supplied in the main program - in this instance the type of the function.

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.

The second time, the program was run with EXTRAC declared integer in the calling program, and the correct result was obtained.

First program:

```
      PROGRAM X (OUTPUT)
C     WITH EXTRAC DECLARED INTEGER THE RESULT SHOULD BE 49, OTHERWISE IT
C           WILL BE ZERO
      K = EXTRAC(7)
      PRINT 1, K
    1 FORMAT (1H1,I5)
      STOP
      END



      INTEGER FUNCTION EXTRAC (K)
      EXTRAC = K*K
      RETURN
      END
```

Output:

```
0
```

Second program:

```
      PROGRAM X (OUTPUT)
C     WITH EXTRAC DECLARED INTEGER THE RESULT SHOULD BE 49, OTHERWISE IT
C            WILL BE ZERO
      INTEGER EXTRAC
      K = EXTRAC(7)
      PRINT 1, K
    1 FORMAT (1H1,I5)
      STOP
      END

      INTEGER FUNCTION EXTRAC (K)
      EXTRAC = K*K
      RETURN
      END
```

Output:

```
49
```

## PROGRAM VARDIM

Program VARDIM illustrates the use of variable dimensions to allow a subroutine to operate on arrays of differing size.

Features:

Passing an array to a subroutine as a parameter.

An array name used as a parameter passes the address of the beginning of the array and no dimension information.

```
COMMON X(4,3)
```

Array X is dimensioned (4,3) and placed in common.

```
REAL Y(6)
```

Array Y dimensioned (6) is explicitly typed real. It is not in common.

```
CALL IOTA(X,12)
```

The subroutine IOTA is called. The first parameter to IOTA is array X, and the second parameter is the number of elements in that array, 12. The number of elements in the array rather than the dimensions (4,3) is used which is legal.

```
SUBROUTINE IOTA(A,M)
DIMENSION A(M)
```

Subroutine IOTA has variable dimensions. Array A is given the dimension 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.

```
CALL IOTA(X,12)
```

When IOTA is called by the main program, the actual argument X replaces A; and 12 replaces M.

```
   DO 1 I=1,M
1  A(I)=I
```

The DO loop places the numbers 1 through 12 in consecutive elements of array X.

```
CALL IOTA(Y,6)
```

When IOTA is called again, Y replaces A and 6 replaces M; and numbers 1 through 6 are placed in consecutive elements of array Y. Notice the type of the arguments in the calling program agree with the type of the arguments in the subroutine. X and A are real, 12 and M are integer.

Names used in the subroutine are related to those in the calling program only by their position as arguments. If a variable I was in the calling program, it would be completely independent of the variable I in the subroutine IOTA.

The WRITE statement outputs the arrays X and Y.

Program VARDIM:

```
      PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
      COMMON X(4,3)
      REAL Y(6)
      CALL IOTA(X,12)
      CALL IOTA(Y,6)
      WRITE (6,100) X,Y
  100 FORMAT (*1ARRAY X = *,12F6.0/*0ARRAY Y = *6F6.0)
      STOP
      END
```

Subroutine IOTA:

```
      SUBROUTINE IOTA (A,M)
C     IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT CF THE ARRAY A
C     STARTING AT 1
      DIMENSION A(M)
      DO 1 I = 1,M
1     A(I)=I
      RETURN
      END
```

Output:

```
ARRAY X =     1.    2.    3.    4.    5.    6.    7.    8.    9.    10.    11.    12.
ARRAY Y =     1.    2.    3.    4.    5.    6.
```

## PROGRAM VARDIM2

VARDIM2 is an extension of program VARDIM. Subroutine IOTA is used; in addition, another subroutine and two functions are used.

Features:

- Multiple entry points
- Variable dimensions
- EXTERNAL statement
- COMMON used for communication between program units
- Passing values through COMMON
- Use of library functions ABS and FLOAT
- Calling functions through several levels
- Passing a subprogram name as an argument

Program VARDIM2 describes 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 IOTA

SUBROUTINE IOTA is described in program VARDIM.

### SUBROUTINE SET

SUBROUTINE SET(A,M,V) 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 DO loop in subroutine SET clears the array to zero.

### 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 declares common to be 12 words and FUNCTION AVG declares common to be 100 words. Function AVG and the main program share the first 12 words in common. 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 to function AVG: PVAL=ABS(WAY(SIZE))

```
COMMON A(100)
```

Function AVG declares common 100 so that varying lengths (less than 100) can be used in calls. In this instance, only 12 of the 100 words are used.

```
   DO 1 I=1,J
1  AVG=AVG+A(I)
```

The DO loop adds the 12 elements in common.

```
AVG=AVG/FLOAT(J)
```

This statement finds the average. The library function FLOAT is used to convert the integer 12 to a floating point (real) number to avoid mixed mode arithmetic.

The average is returned to the statement PVAL=ABS(WAY(SIZE)) in function PVAL.

## FUNCTION 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 library function ABS.

The main program first calls PVAL with the statement AA=PVAL(12,AVG), passing the integer value 12 and the function AVG as parameters.

```
INTEGER SIZE
```

PVAL declares SIZE integer - the type of the argument in the main program (integer 12) agrees with the corresponding dummy argument (SIZE) in the subprogram.

```
PVAL=ABS(WAY(SIZE))
```

The value of PVAL is computed. This value will be returned to the main program through the function name PVAL. Two functions are referenced by this statement; the library function ABS and the user written function AVG. The actual arguments 12 and AVG replace SIZE and WAY.

```
PVAL=ABS(AVG(12))
```

Function AVG is called, and J is given the value 12. The average of the first 12 elements of common are computed by AVG and returned to function PVAL. Library function ABS finds the absolute value of the value returned by AVG.

```
AM=PVAL(12,MULT)
```

In this statement in the main program, PVAL is referenced again. This time the function MULT replaces WAY.

## 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.

```
COMMON ARRAY(12)
```

Common is declared 12; MULT shares the first 12 words of common with the main program.

```
MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
```

The twelfth and first element in common are multiplied and the average of J/2 is subtracted. This is an example of a subprogram calling another subprogram - the function AVG is used to compute the average.

## MAIN PROGRAM — VARDIM2

The main program calls the subroutines and functions described.

```
COMMON X(4,3)
```

Twelve elements in the array X are declared to be in common.

```
REAL Y(6)
```

The real array Y is dimensioned 6.

```
EXTERNAL MULT, AVG
```

Function names MULT and AVG are declared EXTERNAL. Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program. Otherwise it would be treated by the compiler as a variable name.

```
CALL SET(Y,6,0.)
```

Subroutine SET is called. The arguments (Y,6,0.) replace the dummy arguments (A,M,V).

```
   DIMENSION Y (6)
   DO 1 I = 1,6
1  Y(I) = 0.0
```

The array Y is set to zero. The NAMELIST output shows the 6 elements of Y contain zero.

```
CALL IOTA(X,12)
```

Subroutine IOTA is called. X and 12 replace the dummy arguments A and M

```
   DIMENSION X (12)
   DO 1 I=1,12
1  X(I) = I
```

the value of the subscript is placed in each element of the array X. Program VARDIM output shows the value of X is 1 through 12.

```
CALL INC(X,12,-5.)
```

Subroutine SET is called, this time through entry point INC. The arguments (X,12,-5.) replace the dummy arguments (A,M,V)

```
   DO 2 I=1,12
2  X(I) = X(I) + -5.
```

-5. is added to each element in the array X. Program VARDIM2 output shows X is now -4,-3,-2, -1,0,1,2,3,4,5,6,7

```
AA=PVAL(12,AVG)
```

Function PVAL is called and its value replaces AA.

```
AM=PVAL(12,MULT)
```

Function PVAL is called again with different arguments and the value replaces AM.

Complete program:

```
      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
C     THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBFROGRAM CONCEPTS
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      NAMELIST/V/X,Y,AA,AM
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
      AA=PVAL(12,AVG)
      AM=PVAL(12,MULT)
      WRITE(6,V)
      STOP
      END
```

```
      SUBROUTINE SET (A,M,V)
C     SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=0.0
C
      ENTRY INC
C     INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO 2 I = 1,M
    2 A(I) = A(I) + V
      RETURN
      END
```

```
      SUBROUTINE IOTA (A,M)
C     IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C         THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
    1 A(I)=I
      RETURN
      END
```

```
      FUNCTION PVAL(SIZE,WAY)
C  PVAL CCMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALLE IS RETURNED
C     BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED.  SIZE IS AN INTEGER
C     VALUE PASSED CN TO THE FUNCTION.
      INTEGER SIZE
      PVAL=ABS(WAY(SIZE))
      RETURN
      END
```

```
      FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF CCMMON.
      COMMON A(100)
      AVG=0.
      DO 1 I = 1,J
    1 AVG=AVG+A(I)
      AVG=AVG/FLOAT(J)
      RETURN
      END
```

```
      REAL FUNCTION MULT(J)
C      MULT MULTIPLIES THE FIRST AND TWELTH ELEMENTS OF COMMON AND
C      SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C     BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
C
      COMMON ARRAY(12)
      MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
      RETURN
      E  N  D
```

```
$V

X       = -.4E+C1, -.3E+01, -.2E+01, -.1E+01, 0.0, .1E+01, .2E+01, .3E+01,      .4E+01, .5E+01,
  .6E+01, .7E+01,

Y       = 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,

AA      = .15E+01,

AM      = .265E+02,

$END
```

## PROGRAM CIRCLE

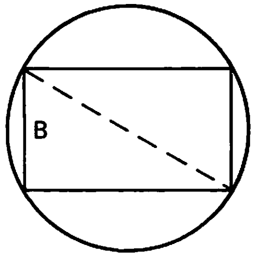
Program CIRCLE finds the area of a circle which circumscribes a rectangle.

Features:

Definition and use of both FUNCTION subprograms and statement functions.

This program has a hidden bug. We suggest you read the text from the start if you intend to find it.

A programmer wrote the following program to find the area of a circle which circumscribes a rectangle, and wrote a function named DIM to compute the diameter of the circle.




The area of a circle is $\pi R^2$, which is approximately the same as 3.1416/4*Diameter**2.

```
      PROGRAM CIRCLE (OUTPUT)
      A=4.0
      B=3.0
      AREA=3.1416/4.0*DIM(A,B)**2
      PRINT 1, AREA
1     FORMAT(G20.10)
      STOP
      END
      FUNCTION DIM(X,Y)
      DIM=SQRT(X*X+Y*Y)
      RETURN
      END
```

Output:

```
      .7854000000
```

The programmer was completely baffled by the result; the area of a circle circumscribing a rectangle 12 square inches should be more than .785! Another programmer quickly pointed out that a simple function like DIM should have been written as a statement function. Since FORTRAN Extended compiles statement functions inline, it would execute much faster because no jump nor return jump would be generated by the function.

The programmer rewrote the program as follows:

```
      PROGRAM CIRCLE (OUTPUT)
      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 (G20.10)
      STOP
      END
```

and obtained the correct result.

> When the programmer wrote the function subprogram, it had the same name as a library intrinsic function. If the name of an intrinsic function is used for a user written function, the user written function is ignored.

## PROGRAM OCON

Program OCON illustrates some problems that may occur with octal or Hollerith constants.

Features:

Octal Constants in expressions

The compiler generally treats both octal and Hollerith constants as having no type; therefore, no mode conversion is done when they are used in expressions. If, however, the compiler is forced to assume a type for an octal or Hollerith constants, it will treat them as integer. When an expression contains only operands having no type, integer arithmetic is used. For example:

```
B=10B+10B
```

The expression 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 in OCON 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 Appendix D discussion of Underflow.) This explains why the output value of A from OCON is zero.

```
C=B+10B
```

Floating point arithmetic is used to evaluate the expression; and the octal constant 10B is used without type conversion, making its value zero. Note in the output from OCON, the values of B and C are equal.

```
D=I+10B
```

No problem arises in the above expression as it is evaluated with integer arithmetic; then the result is converted to real and stored in D.

```
E=B+I+10B
```

The compiler, in scanning the above expression left to right, encounters the real variable B and uses real arithmetic to evaluate the expression. Again, the octal constant 10B has the real value of zero.

If the expression were written as:

E=10B+I+B    or    E=I+10B+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 10B 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 in section 2.

```
F=A.EQ.77B
```

Real arithmetic is used to compare the value because A is a type real name. The value in A and the constant 77B 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 integer names for values that come from octal or Hollerith constants or character data that is input using A or R format elements. To illustrate, this program was rerun with the names A, B, C, D, and E all as type integer.

All these examples use octal constants; however, the same problem occurs with Hollerith, especially when it is right-justified. The following coding illustrates the point:

```
         .
         .
         .
      REAL ANS
         .
         .
         .
      READ 2, ANS
    2 FORMAT(R3)
      IF(ANS .EQ. 3RNO )PRINT3
    3 FORMAT (*-NEGATIVE RESPONSE*)
         .
         .
         .
```

PRINT3 of the logical IF is always executed independently of information in the data records.

With real variables:

```
      PROGRAM OCON(OUTPUT,TAPE6=OUTPUT)
      LOGICAL F
      NAMELIST/OUT/A,B,C,D,E,F
      A=20B
      B=10B+10B
      C=B+10B
      I=5
      D=I+10B
      E=B+I+10B
      F=A.EQ.77B
      WRITE(6,OUT)
      STOP
      END
```

Output:

```
$OUT

A         =  0.0,

B         =  .16E+02,

C         =  .16E+02,

D         =  .13E+02,

E         =  .21E+02,

F         = T,

$END
```

With integer variables:

```
      PROGRAM OCON(OUTPUT,TAPE6=OUTPUT)
      INTEGER A,B,C,D,E
      LOGICAL F
      NAMELIST/OUT/A,B,C,D,E,F
      A=20B
      B=10B+10B
      C=B+10B
      I=5
      D=I+10B
      E=B+I+10B
      F=A.EQ.77B
      WRITE(6,OUT)
      STOP
      END
```

Output:

```
$OUT

A         = 16,

B         = 16,

C         = 24,

D         = 13,

E         = 29,

F         = F,

$END
```

## 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 statements is not required. This facility is particularly useful when FORTRAN programs are being run from a remote terminal.

Example:

```
H2,T10.
MAP(OFF)
FTN(R=0)
LGO.
7/8/9
      PROGRAM EASY IO (INPUT=/80,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
COMPUTE THE AREA AND RADIUS OF AN INSCRIBED CIRCLE OF ANY TRIANGLE.
      REAL SIDES(3)
      EQUIVALENCE(SIDES(1),A),(SIDES(2),B),(SIDES(3),C)
      NAMELIST/OUT/SIDES,AREA,RADIUS
3     READ(5,*)SIDES
      IF(EOF(5).NE.0)STOP
      S=(A+B+C)/2.
      AREA=SQRT(S*(S-A)*(S-B)*(S-C))
      RADIUS=AREA/S
      WRITE(6,OUT)
      GOTO3
      END
7/8/9
3 4 5
6,7,8
3*1
4
5
6
12.5321452, 22.4536,25
6/7/8/9
```

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    = .20333162567589E+02,
RADIUS  = .19364916731037E+01,
$END

$OUT
SIDES   = .1E+01, .1E+01, .1E+01,
AREA    = .43301270189222E+00,
RADIUS  = .28867513459481E+00,
$END

$OUT
SIDES   = .4E+01, .5E+01, .6E+01,
AREA    = .99215674164922E+01,
RADIUS  = .13228756555323E+01,
$END

$OUT
SIDES   = .125321452E+02, .224536E+02, .25E+02,
AREA    = .14040422058737E+03,
RADIUS  = .46812528582998E+01,
$END
```

The user may 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 input value is a whole number.

# STATIC OPTION 20

The STATIC option is available to those users of FORTRAN Extended who wish to disable Common Memory Manager so they can do their own memory management by over-indexing blank common. The practice of over-indexing blank common is not recommended. The STATIC option is provided to allow continued use of programs which rely on over-indexing of blank common.

Since STATIC involves usage of non-ANSI standard capabilities, compliance with future ANSI FORTRAN standards may preclude future availability of this option.

Use of this option is restricted when a mixture of both static and dynamically compiled subprograms is executed. When the main program is compiled with STATIC, the entire set of loaded routines must be STATIC or unpredictable results will occur. If the main program is non-static, there is no restriction on whether all, part, or none of the subprograms are static.

When the STATIC option is on for a main program, the compiler issues a LDSET to FCL which causes CMM to be disabled.

Users wishing to exercise control over memory management but not wishing to use the STATIC option should refer to the Common Memory Manager Interface described in section 8. This facility allows calls to CMM directly from FORTRAN Extended. These calls allow the user to obtain blocks of memory directly from CMM rather than forcing use of the STATIC option and over-indexing of blank common.

# STANDARD CHARACTER SET 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

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). The user, however, may specify the alternate mode by a 26 or 29 punched in columns 79 and 80 of the job card or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS 1, the alternate mode can be specified also by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 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.

When the 63-character set is used, the display code character $00_8$ under A or R FORMAT conversion will be converted to a space, display code $55_8$ for ENCODE and DECODE as well as formatted input/output statement.

No conversions occur with the A or R FORMAT element when the 64-character set is used.

| 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 | ∨ | 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 ($55_8$).

# HEXADECIMAL–OCTAL CONVERSION TABLE

| Second Hexadecimal Digit ╲ First Hexadecimal Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 000 | 020 | 040 | 060 | 100 | 120 | 140 | 160 | 200 | 220 | 240 | 260 | 300 | 320 | 340 | 360 |
| 1 | 001 | 021 | 041 | 061 | 101 | 121 | 141 | 161 | 201 | 221 | 241 | 261 | 301 | 321 | 341 | 361 |
| 2 | 002 | 022 | 042 | 062 | 102 | 122 | 142 | 162 | 202 | 222 | 242 | 262 | 302 | 322 | 342 | 362 |
| 3 | 003 | 023 | 043 | 063 | 103 | 123 | 143 | 163 | 203 | 223 | 243 | 263 | 303 | 323 | 343 | 363 |
| 4 | 004 | 024 | 044 | 064 | 104 | 124 | 144 | 164 | 204 | 224 | 244 | 264 | 304 | 324 | 344 | 364 |
| 5 | 005 | 025 | 045 | 065 | 105 | 125 | 145 | 165 | 205 | 225 | 245 | 265 | 305 | 325 | 345 | 365 |
| 6 | 006 | 026 | 046 | 066 | 106 | 126 | 146 | 166 | 206 | 226 | 246 | 266 | 306 | 326 | 346 | 366 |
| 7 | 007 | 027 | 047 | 067 | 107 | 127 | 147 | 167 | 207 | 227 | 247 | 267 | 307 | 327 | 347 | 367 |
| 8 | 010 | 030 | 050 | 070 | 110 | 130 | 150 | 170 | 210 | 230 | 250 | 270 | 310 | 330 | 350 | 370 |
| 9 | 011 | 031 | 051 | 071 | 111 | 131 | 151 | 171 | 211 | 231 | 251 | 271 | 311 | 331 | 351 | 371 |
| A | 012 | 032 | 052 | 072 | 112 | 132 | 152 | 172 | 212 | 232 | 252 | 272 | 312 | 332 | 352 | 372 |
| B | 013 | 033 | 053 | 073 | 113 | 133 | 153 | 173 | 213 | 233 | 253 | 273 | 313 | 333 | 353 | 373 |
| C | 014 | 034 | 054 | 074 | 114 | 134 | 154 | 174 | 214 | 234 | 254 | 274 | 314 | 334 | 354 | 374 |
| D | 015 | 035 | 055 | 075 | 115 | 135 | 155 | 175 | 215 | 235 | 255 | 275 | 315 | 335 | 355 | 375 |
| E | 016 | 036 | 056 | 076 | 116 | 136 | 156 | 176 | 216 | 236 | 256 | 276 | 316 | 336 | 356 | 376 |
| F | 017 | 037 | 057 | 077 | 117 | 137 | 157 | 177 | 217 | 237 | 257 | 277 | 317 | 337 | 357 | 377 |
| Octal | 000 – 037 | | 040 – 077 | | 100 – 137 | | 140 – 177 | | 200 – 237 | | 240 – 277 | | 300 – 337 | | 340 – 377 | |

# FORTRAN DIAGNOSTICS

Diagnostic messages are produced by the FORTRAN Extended compiler during both compilation and execution to inform the user of errors in the source program, input data or intermediate results.

## COMPILATION DIAGNOSTICS

The compile time diagnostics issued by FORTRAN Extended differ in format and content for optimizing mode and time-sharing mode.

### OPTIMIZING MODE DIAGNOSTICS

Errors detected during compilation are noted on the source listing immediately following the END statement. The format of the message is as follows:

| CARD NO. | SEVERITY | DETAILS | DIAGNOSTIC |
|---|---|---|---|
| n | e | a | error message |

n — Line number where error was detected. This number is assigned by the FORTRAN Extended compiler. Some declarative statement diagnostics show the line number of the first non-declarative statement; END line number is used for undefined statement number diagnostics.

e — Indicates the type of diagnostic. In the following pages, compile time diagnostics are listed alphabetically by error type.

| | |
|---|---|
| I | Informative message which indicates minor syntax errors or omissions which have no effect upon compilation or execution. |
| FC | When an error of this type is encountered during compilation, the remaining portion of the program is checked for syntax errors only. Program is not executed. |
| FE | Error fatal to execution. Program compiles but does not execute. |
| ANSI | Usage does not conform to ANSI standard FORTRAN (X3.9 - 1966). ANSI diagnostics are not listed unless the EL=A parameter is specified on the FTN control statement. |

a — Information in this column will differ according to the type of error encountered. For example, if the same statement label is used more than once, the label number is printed. If a message of the format cn CD n appears, cn is the column number in which the error was detected, and n is the card number.

error message — Error message printed by compiler.

In table B-1, the message "see DETAILS column" refers to the DETAILS column described above.

The optimizing mode diagnostics are shown in table B-1.

Example:

```
 100 WRITE (6,8)
 8   FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
     119X,1H1/19X,1H3)
 101 I=5
  8  A=I
102  A=SQRT(A)
103  J=A
104  DO 1 K=3,J,2
105  L=I/KEXCEEDS
106  IF(L*K-I)1,2,4
   1 GO TO 108
107  WRITE (6,9)
   5 FORMAT (I20)
  2  I=I+2
108  IF (1000-I) 7,4,3
  4  WRITE(6,7)
  9  FORMAT (14H PROGRAM ERROR)
  7  WRITE (6,6)
   6 FORMAT (31H THIS IS THE END OF THE PROGRAM)
109  STOP
     END


CARD NR. SEVERITY  DETAILS   DIAGNOSIS OF PROBLEM

     1     I                  *PROGRAM START. (INPUT,OUTPUT)* ASSUMED WHEN HEADER STATEMENT IS DEFECTIVE OR OMITTED.
     2     FE    72  CD 2     ZERO LEVEL RIGHT PARENTHESIS MISSING.  SCANNING STOPS.
     3     FE                 UNRECOGNIZED STATEMENT.
     5     FE    8            DUPLICATE STATEMENT LABEL.
     9     FE    <EXCEEDS     SYMBOLIC NAME HAS TOO MANY CHARACTERS.
    11     FE                 A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT.
    16     FE     7           PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES.
    21     FE                 UNDEFINED STATEMENT LABEL(S), SEE LIST BELOW.

       UNDEFINED LABELS
      3
```

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI A COMMENT LINE WITHIN A CONTINUED STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI A RELATIONAL HAS A COMPLEX OPERAND. | Complex operands are not permitted in relational expressions. | Self-evident. | Optimizing mode compiler. |
| ANSI AN EXPRESSION IN AN OUTPUT STATEMENT I/O LIST IS NON-ANSI USAGE. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI AN EXPRESSION OF THE FORM A**B**C IS NON-ANSI, AND IS EVALUATED FROM LEFT TO RIGHT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI ARRAY NAME OPERAND NOT SUB-SCRIPTED. FIRST ELEMENT WILL BE USED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI ARRAY NAME REFERENCED WITH FEWER SUBSCRIPTS THAN DIMENSIONALITY OF ARRAY. | Remaining subscripts are assumed to be one. | Self-evident. | Optimizing mode compiler. |
| ANSI ATTEMPT TO BACK UP BEFORE COLUMN ONE CAUSES POSITIONING TO BE SET AT COLUMN ONE. | An attempt has been made to backspace beyond column one. | Check column count in /nH+ specification. | Optimizing mode compiler. |
| ANSI BACKING UP WITH X SPECIFICATION IS NON-ANSI. | -nX specification is not permitted in ANSI. | Change to /nH+ specification. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI CONTROL FLOW INTO END LINE NOT PERMITTED. | The END line is non-executable and must not appear in the flow of execution. | Precede the END line by a STOP, RETURN, or GO TO statement. | Optimizing mode compiler. |
| ANSI DATA VARIABLE TYPE DOES NOT MATCH CONSTANT | The type of a constant does not agree with the type of the variable into which it is stored. | Use the proper type declaration, or change the constant type to agree with the variable type. | Optimizing mode compiler. |
| ANSI DOLLAR SIGN STATEMENT SEPARATOR IS NON-ANSI USAGE. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI END STATEMENT ACTING AS A RETURN IS NON-ANSI. | Control must not flow into an END statement. | Self-evident. | Optimizing mode compiler. |
| ANSI ENTRY STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI FLOATING POINT DESCRIPTOR EXPECTED AFTER SCALE FACTOR DESIGNATOR. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| ANSI GO TO STATEMENT CONTAINS NON-ANSI USAGES. | Computed GO TO variable is an expression or comma following terminal paren is missing. | See DETAILS column. | Optimizing mode compiler. |
| ANSI HOLLERITH CONSTANT APPEARS OTHER THAN IN AN ARGUMENT LIST OF A CALL STATEMENT OR IN A DATA STATEMENT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI HOLLERITH STRING DELIMITED BY SYMBOLS IS NON-ANSI. | Hollerith specification must be denoted by nH. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI IMPLICIT STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI LOGICAL OPERATOR OR CONSTANT USAGE IS NON-ANSI. | ANSI does not permit single character logical operators. | Self-evident. | Optimizing mode compiler. |
| ANSI MASKING EXPRESSION IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI MULTIPLE REPLACEMENT STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI NAMELIST STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI NON-ANSI BLANK LINES OCCURRED IN THIS PROGRAM UNIT. | ANSI interprets blank lines as initial lines. | Add continuation character to column 6 of initial lines following blank lines. | Optimizing mode compiler. |
| ANSI NON-ANSI FORM OF BLOCK DATA STATEMENT. | The form BLOCK DATA name is invalid in ANSI FORTRAN. | Use BLOCK DATA. FORTRAN assigns name BLKDAT. | Optimizing mode compiler. |
| ANSI NON-ANSI FORM OF DATA STATEMENT. | Alternate DATA statement syntax not permitted by ANSI. | Self-evident. | Optimizing mode compiler. |
| ANSI NON-ANSI FORM OF TYPE DECLARATION. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| ANSI OBJECT OF LOGICAL IF IS ILLEGAL DO TERMINATOR. | DO loop cannot terminate on logical IF whose object is GOTO, RETURN, END, STOP, PAUSE, of DO statement. | Self-evident. | Optimizing mode compiler. |
| ANSI OCCURRENCES OF ASTERISK OR DOLLAR SIGN NON-ANSI COMMENT LINES. | ANSI comment line indicated by 'C' in column 1. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI OCTAL CONSTANT OR R,L FORMS OF HOLLERITH CONSTANT IS NON-ANSI. | Only nH Hollerith specification permitted. | Self-evident. | Optimizing mode compiler. |
| ANSI OMISSION OF FIELD SEPARATOR AFTER HOLLERITH STRING IS NON-ANSI. | A comma must follow the Hollerith string. | Self-evident. | Optimizing mode compiler. |
| ANSI ONE OF THE FOLLOWING NON-ANSI FORMS HAS BEEN USED - EW.DDE, EW.DEE, IW.Z, OW.Z. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI PLUS SIGN IS A NON-ANSI CHARACTER. | Non-ANSI allows an optional + to precede format X specification. | Self-evident. | Optimizing mode compiler. |
| ANSI PRECEDING FIELD DESCRIPTOR IS NON-ANSI. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| ANSI REDEFINITION OF PARAMETER. . . . . USED AS DIMENSION INDICATOR IS NON-ANSI. | Integer variable used as an array declarator in a DIMENSION statement was altered in a subsequent expression. | Self-evident. | Optimizing mode compiler. |
| ANSI RETURNS PARAMETERS IN CALL STATEMENT. | Non-standard RETURN is non-ANSI. | Self-evident. | Optimizing mode compiler. |
| ANSI SAME NAME USED AS FUNCTION AND SUBROUTINE | Usage violates ANSI class restrictions. | Reference the subprogram either as a function or as a subroutine. | Optimizing mode compiler. |
| ANSI SUBSCRIPT DOES NOT CONFORM TO ANSI STANDARD. | Non-ANSI allows subscript expression to be any valid arithmetic expression. | Self-evident. | Optimizing mode compiler. |
| ANSI TAB SETTING DESIGNATOR IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI THE EXPRESSION IN AN IF STATEMENT IS TYPE COMPLEX. | Expression in arithmetic. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI THE FORMAT OF THIS END LINE DOES NOT CONFORM TO ANSI SPECIFICATIONS. | ANSI END line cannot be labeled or continued. | Self-evident. | Optimizing mode compiler. |
| ANSI THE NON-STANDARD RETURN STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI THE TYPE COMBINATION OF THE OPERANDS OF A RELATIONAL OR ARITHMETIC OPERATOR (OTHER THAN **) IS NON-ANSI. | Self-evident. | Use FLOAT or IFIX to change type. | Optimizing mode compiler. |
| ANSI THE TYPE COMBINATION OF THE OPERANDS OF AN EQUAL-SIGN OPERATOR IS NON-ANSI. | Self-evident. | Use FLOAT or IFIX to change type. | Optimizing mode compiler. |
| ANSI THE TYPE COMBINATION OF THE OPERANDS OF AN EXPONENT OPERATOR IS NON-ANSI. | Self-evident. | Use FLOAT or IFIX to change type. | Optimizing mode compiler. |
| ANSI THIS FORM OF AN I/O STATEMENT DOES NOT CONFORM TO ANSI SPECIFICATIONS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI THIS FORMAT DECLARATION IS NON-ANSI. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| ANSI THIS STATEMENT TYPE IS NON-ANSI. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI TWO-BRANCH IF STATEMENT IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI USE OF A NUMBER AS LABELED COMMON BLOCK NAME IS NON-ANSI. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| ANSI 7 CHARACTER SYMBOLIC NAME IS NON-ANSI. | Maximum is 6 characters. | Self-evident. | Optimizing mode compiler. |
| ANSI DATA VARIABLE DOES NOT MATCH CONSTANT. | Type of variable in a DATA statement does not match its associated constant. | Change one or the other. | Optimizing mode compiler. |
| FC ERROR TABLE OVERFLOW. | Too many errors have been detected in the user's program unit. Compilation cannot continue. | Correct as many errors as possible and recompile. | Optimizing mode compiler. |
| FC MEMORY OVERFLOW DURING ASF EXPANSION. | Arithmetic statement function caused memory overflow. | Simplify statement function or reduce number of calls. | Optimizing mode compiler. |
| FC NOT ENOUGH ROOM IN WORKING STORAGE TO HOLD ALL OVERLAY CONTROL CARD INFORMATION. | There are too many overlays in the user's program. Compilation terminated. | Reduce the number of overlays and recompile. | Optimizing mode compiler. |
| FC SYMBOL TABLE OVERFLOW. | There are too many symbols in the user's program unit. Compilation terminated. | Reduce the number of symbols in the program unit, or modularize and compile each module separately. | Optimizing mode compiler. |
| FC TABLE OVERFLOW, INCREASE FL. | A compiler table has overflowed. | Increase field length with RFL control statement and recompile. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FC TABLES OVERLAP, INCREASE FL. | Compiler tables have overlapped. | Increase field length when RFL control statement and recompile. | Optimizing mode compiler. |
| FC THIS SUBPROGRAM HAS TOO MANY DO LOOPS. | The compiler cannot process all the DO loops. | Reduce the number of DO loops and recompile. | Optimizing mode compiler. |
| FE A COMPLEX BASE MAY ONLY BE RAISED TO AN INTEGER POWER. | Self-evident. | Self-evident. | |
| FE A CONSTANT ARITHMETIC OPERATION WILL GIVE AN INDEFINITE OR OUT-OF-RANGE RESULT. | A constant expression has a value that will cause an execution error. | Check expression for division by zero. | Optimizing mode compiler. |
| FE A CONSTANT CANNOT BE CONVERTED. CHECK CONSTANT FOR PROPER CONSTRUCT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE A CONSTANT DO PARAMETER MUST BE GREATER THAN OR EQUAL TO 1 AND LESS THAN OR EQUAL TO 131070. | A DO loop indexing parameter was assigned a value outside the legal range. | See DETAILS column. | Optimizing mode compiler. |
| FE A CONSTANT MAY NOT BE FOLLOWED BY AN EQUAL SIGN, NAME, OR ANOTHER CONSTANT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE A CONSTANT OPERAND OF A REAL OPERATION IS OUT OF RANGE OR INDEFINITE. | The constant has an illegal value; the operation cannot be perform | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE A C/ LIST DIRECTIVE MAY NOT BE FOLLOWED BY A CONTINUATION LINE. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE A DO LOOP MAY NOT TERMINATE ON A FORMAT STATEMENT. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE A DO PARAMETER MUST BE A POSITIVE INTEGER CONSTANT OR AN INTEGER VARIABLE. | Self–evident. | Self-evident. | Optimizing mode compiler. |
| FE A FUNCTION REFERENCE REQUIRES AN ARGUMENT LIST. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE A NAME MAY NOT BE FOLLOWED BY A CONSTANT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE A PREVIOUS STATEMENT MAKES AN ILLEGAL TRANSFER TO THIS LABEL. | Self-evident. | Check for illegal transfer to terminal statement of DO or to FORMAT label. | Optimizing mode compiler. |
| FE A PREVIOUSLY MEN- TIONED ADJUSTABLE SUBSCRIPT IS NOT TYPE INTEGER. | Formal parameter used as subscript is not type integer. | Specify proper type statement. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE A REFERENCE TO THIS ARITHMETIC STATEMENT FUNCTION HAS UNBALANCED PARENTHESIS WITHIN THE PARAMETER LIST. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE A REFERENCE TO THIS ASF HAS A PARAMETER MISSING. | The number of parameters in the reference must match the number of parameters in the function definition. | Self–evident. | Optimizing mode compiler. |
| FE A VARIABLE DIMENSION OR THE ARRAY NAME WITH A VARIABLE DIMENSION IS NOT A FORMAL PARAMETER. | Variable dimensions can appear only in a subprogram. The array name and variable representing the adjustable dimension must be passed as arguments. | Self–evident. | Optimizing mode compiler. |
| FE ABSOLUTE VALUE OF INTEGER CONSTANT GREATER THAN 2**59–1. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE ALL LEVEL 2 or 3 ITEMS MUST BE FORMAL PARAMETERS OR IN COMMON. | A data item declared to be level 2 or 3 must appear in a common statement or in subprograms, as dummy arguments in the argument list. | Self–evident. | Optimizing mode compiler. |
| FE AN ARRAY REFERENCE HAS TOO MANY SUBSCRIPTS. | The number of subscripts in an array reference must not exceed the number of declared dimensions. | Self–evident. | Optimizing mode compiler. |
| FE APPEARED WHERE A VARIABLE WAS EXPECTED. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE ARG TO LOCF MAY NOT BE AN EXPRESSION. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ARGUMENT NOT FOLLOWED BY COMMA OR RIGHT PARENTHESIS. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ARITHMETIC STATEMENT FUNCTION REDEFINED. | More than one definition for the same statement function. | Might be an array usage with no DIMENSION statement. | Optimizing mode compiler. |
| FE ARRAY HAS MORE THAN THREE SUBSCRIPTS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ARRAY OR COMMON VARIABLE MAY NOT BE DECLARED EXTERNAL. | An array or COMMON variable appears in an EXTERNAL statement. | Self-evident. | Optimizing mode compiler. |
| FE ARRAY WITH ILLEGAL SUBSCRIPTS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ASF HAS MORE DUMMY PARAMETERS THAN ALLOWED. | The argument list of an arithmetic statement function can contain up to 63 dummy arguments. | Self-evident. | Optimizing mode compiler. |
| FE BAD SUBSCRIPT IN EQUIV STMT. | The subscript of an array element in an EQUIVALENCE statement is invalid. | See DETAILS column. | Optimizing mode compiler. |
| FE BAD SYNTAX ENCOUNTERED. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE BASIC EXTERNAL OR INTRINSIC FUNCTION CALLED WITH WRONG TYPE ARGUMENT. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE BASIC OR INTRINSIC FUNCTION WITH AN INCORRECT ARGUMENT COUNT. | The number of arguments in a function call must agree with the number of arguments defined for the function. | Self-evident. | Optimizing mode compiler. |
| FE COMMON BLOCK LENGTH EXCEEDS 131071 WORDS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE COMMON VARIABLE IS FORMAL PARAMETER OR PREVIOUSLY DECLARED IN COMMON OR ILLEGAL NAME. | A variable cannot appear in more than one COMMON block, or in COMMON and subprogram argument list. | Self-evident. | Optimizing mode compiler. |
| FE CONFLICTING LEVEL DECLARATIONS EXIST IN THIS COMMON BLOCK. | Variables within a COMMON block must exist at the same level in memory, as declared in the LEVEL statement. | Self-evident. | Optimizing mode compiler. |
| FE CONSTANT DATA ITEM MUST BE FOLLOWED BY A , / OR RIGHT PAREN. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE CONSTANT SUBSCRIPT VALUE EXCEEDS ARRAY DIMENSIONS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE DATA ITEM LISTS MAY ONLY BE NESTED 1 DEEP. | Nested implied-DO specifications are not permitted in the DATA statement. | Self-evident. | Optimizing mode compiler. |
| FE DATA VARIABLE LIST SYNTAX ERROR. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE DEBUG EXECUTION OPTION SUPPRESSED DUE TO NATURE OF ABOVE FATAL ERRORS. | In debug mode the partial execution option was specified, but errors have occurred which prohibit execution. | Correct other compilation errors and recompile. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE DECLARATIVE STATEMENT OUT OF SEQUENCE. | See section 1 for required order of nonexecutable statements. | Re-order statements. | Optimizing mode compiler. |
| FE DEFECTIVE HOLLERITH CONSTANT. CHECK FOR CHARACTER COUNT ERROR, MISSING ≠ DELIMITER OR LOST CONTIN CARD. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE DIVISION BY CONSTANT ZERO. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE DO LIMIT OR REP FACTOR MUST BE AN INTEGER OR OCTAL CONSTANT BETWEEN 1 AND 131K. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE DO LOOPS TERMINATING ON THIS LABEL ARE IMPROPERLY NESTED. | Branch to terminal statement occurs from an outer DO. | Check control structure of program unit. | Optimizing mode compiler. |
| FE DOUBLY DEFINED FORMAL PARAMETER. | Parameter occurs twice in a FUNCTION or SUBROUTINE statement. | Self-evident. | Optimizing mode compiler. |
| FE DUMMY PARAMETER IN ASF DEFINITION OCCURRED TWICE. | Dummy parameter can occur only once in an arithmetic statement function definition. | Self-evident. | Optimizing mode compiler. |
| FE DUPLICATE LOOP INDEX OR DOESNT MATCH ANY SUBSCRIPT VARIABLE. | DATA statement implied DO error. | Self-evident. | Optimizing mode compiler. |
| FE DUPLICATE STATEMENT LABEL. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE ECS/LCM REFERENCE MUST BE A STAND-ALONE ARGUMENT TO AN EXTERNAL ROUTINE. | Level 3 variables cannot be used in expressions. | Self-evident. | Optimizing mode compiler. |
| FE ENTRY POINT NAMES MUST BE UNIQUE - THIS ONE HAS BEEN PREVIOUSLY USED IN THIS SUBPROGRAM. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ENTRY STATEMENT MAY NOT APPEAR IN A PROGRAM. | ENTRY statement can appear only in a subprogram. | Self-evident. | Optimizing mode compiler. |
| FE ENTRY STATEMENT MAY NOT BE LABELED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ENTRY STATEMENTS MAY NOT OCCUR WITHIN THE RANGE OF A DO STATEMENT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE EQUATED FILENAME NOT PREVIOUSLY DEFINED. | File name in PROGRAM statement must be defined before appearing on right of equals sign. | Re-order file definitions. | Optimizing mode compiler. |
| FE EQUIVALENCED COMMON BLOCK EXCEEDS 131071 WORDS. | Equivalencing of variables in common extends block past maximum size. | Self-evident. | Optimizing mode compiler. |
| FE EXTERNAL NAME IN ARGUMENT LIST MUST APPEAR IN EXTERNAL STATEMENT. | A subroutine function, or entry point name appearing in an argument list must be declared EXTERNAL. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE FIELD WIDTH IS GREATER THAN 131,071. SCANNING STOPS. | Error in FORMAT statement. | See DETAILS column. | Optimizing mode compiler. |
| FE FILENAME IS GREATER THAN 6 CHARACTERS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE FILENAME PREVIOUSLY DEFINED. | Duplicate file definition in PROGRAM statement. | Eliminate all but one definition. | Optimizing mode compiler. |
| FE FIRST WORD AND LAST WORD ADDRESSES OF DATA TRANSMISSION BLOCK MUST BE IN THE SAME LEVEL. | The first and last word addresses of the data to be moved by a buffer statement or MOVLEV are not in the same level. | Self-evident. | Optimizing mode compiler. |
| FE FOLLOWED BY AN ILLEGAL ITEM. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE FORMAL PARAMETERS MAY NOT APPEAR IN COMMON OR EQUIV STMTS. | A dummy argument in a subprogram has appeared in a COMMON or EQUIVA-LENCE statement. | Self-evident. | Optimizing mode compiler. |
| FE FORMAT REFERENCE ILLEGAL. | Bad syntax for format parameter in input/output statement. | Self-evident. | Optimizing mode compiler. |
| FE FORMAT STATEMENT ENDS BEFORE END OF HOLLERITH STRING. ERROR SCAN FOR THIS FORMAT STOPS HERE. | An error was detected while processing a Hollerith string in a FORMAT statement. | Check FORMAT statements; possible incorrect length specification for Hollerith string or missing continuation line. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE FORMAT STATEMENT ENDS BEFORE LAST HOLLERITH COUNT IS COMPLETE. ERROR SCAN FOR THIS FORMAT STOPS AT H. | An error was detected while processing a Hollerith string in a FORMAT statement. | Check FORMAT statements for incorrect length specification for Hollerith string or missing continuation line. | Optimizing mode compiler. |
| FE FUNCTION NAME DOES NOT APPEAR AS A VARIABLE IN THIS SUBPROGRAM. | In a function subprogram, the function name must be assigned a value. | Self-evident. | Optimizing mode compiler. |
| FE F.P. WITH VARIABLE DIMENSIONS NOT ALLOWED IN A NAMELIST STATEMENT. | An array with adjustable dimensions is not allowed in a NAMELIST statement. | Self-evident. | Optimizing mode compiler. |
| FE GO TO STATEMENT – SYNTAX ERROR. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE GROUP NAME NOT SURROUNDED BY SLASHES. | Self-evident. | Check group names in NAMELIST statements. | Optimizing mode compiler. |
| FE GROUP NAME PREVIOUSLY REFERENCED IN ANOTHER CONTEXT. | NAMELIST group names must be unique. | Self-evident. | Optimizing mode compiler. |
| FE HEADER CARD NOT FIRST STATEMENT. | PROGRAM, FUNCTION or SUBROUTINE statement occurred after the first statement of the program unit. | Statements are out of order or an END line is missing. | Optimizing mode compiler. |
| FE HEADER CARD SYNTAX ERROR. | Error in PROGRAM, FUNCTION or SUBROUTINE statement. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE ILLEGAL BLOCK NAME | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTER FOUND IN IMPLICIT STATEMENT. | The second of two characters in an IMPLICIT specification precedes the first character. | Self-evident. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTER FOLLOWS PRECEDING A, I, L, O, R OR Z DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE. | An error was detected in a field specification in a FORMAT statement. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTER FOLLOWS PRECEDING FLOATING POINT DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE. | An error was detected in a floating point field specification in a FORMAT statement. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTER FOLLOWS PRECEDING SIGN CHARACTER. ERROR SCAN FOR THIS FORMAT STOPS HERE. | Error detected in FORMAT statement. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTERS AFTER TERMINATING RIGHT PARENTHESIS. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL CHARACTER. THE REMAINDER OF THIS STATEMENT WILL NOT BE COMPLETED. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN. | A COMMON block origin has been extended backwards. | Check EQUIVALENCE statements for improper equivalencing of COMMON variables. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE ILLEGAL FORM INVOLVING THE USE OF A COMMA. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL LABEL FIELD. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE ILLEGAL LABELS IN IF STATEMENT. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE ILLEGAL LIST ITEM ENCOUNTERED IN AN I/O LIST SEQUENCE. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL NAMELIST VARIABLE. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL RETURNS PARAMETER. | A parameter in the "RETURNS list" statement contains an error. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SEPARATOR ENCOUNTERED. | A character other than a / or , was used as a separator. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SEPARATOR IN EXTERNAL STATEMENT. | A character other than , was used as a separator in the EXTERNAL statement. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SYNTAX AFTER INITIAL KEYWORD OR NAME. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SYNTAX IN CALL STATEMENT. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SYNTAX IN COMMON DECLARATION. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE ILLEGAL SYNTAX IN IF STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SYNTAX IN IMPLICIT STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL SYNTAX IN IMPLIED DO SPECIFICATION. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ILLEGAL TYPE SPECIFIED IN IMPLICIT STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL USE OF A FUNCTION NAME. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ILLEGAL USE OF THE EQUAL SIGN. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE ILLEGAL VARIABLE NAME FIELD IN ASSIGN OR ASSIGNED GOTO. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE IMPROPER FORM OF ENTRY STATEMENT. ONLY ALLOWABLE FORM IS (ENTRY NAME) | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE INCORRECT SYNTAX FOLLOWING INDICATED ELEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE INTEGER CONSTANT FOR MULTIPLICATION OR DIVISION EXCEEDS 2**48-1. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE INTRINSIC FUNCTION REFERENCE MAY NOT USE A FUNCTION NAME AS AN ARGUMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE INVALID LEVEL NUMBER SPECIFIED. | A LEVEL statement specified a level number other than 1,2,or 3. | Self–evident. | Optimizing mode compiler. |
| FE INVALID USE OF A CHARACTER STRING. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE INVOLVED IN CONTRA-DICTORY EQUIVALENCING. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ITEMS IN DIFFERENT LEVELS OF STORAGE MAY NOT BE EQUIVALENCED. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE LEFT SIDE OF REPLACE-MENT STATEMENT IS ILLEGAL. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE LOADER DIRECTIVE OUT OF SEQUENCE. MUST PRECEDE PROGRAM UNIT HEADER LINE. | The OVERLAY directive must precede the PROGRAM, FUNCTION, or SUBROUTINE statement. | Self–evident. | Optimizing mode compiler. |
| FE LOGICAL AND NON-LOGICAL OPERANDS MAY NOT BE MIXED. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE LOGICAL EXPRESSION IN 3–BRANCH IF STATEMENT. | Only masking and arithmetic expressions can be used in a 3–branch IF statement. | Self–evident. | Optimizing mode compiler. |
| FE LOGICAL OPERAND USED WITH NON–LOGICAL OPERATORS. | Self–evident. | Check type declarations; logical operands must be declared type LOGICAL. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE LOOP BEGINNING AT THIS CARD NO IS ENTERED FROM OUTSIDE ITS RANGE AND HAS NO EXITS. | A DO loop cannot be entered from outside its range, except when control is transferred out of the range and then back in. | Check all GO TO statements for illegal branches into range of DO loop; check for missing branch out of DO loop range. | Optimizing mode compiler. |
| FE LOOPS ARE NESTED MORE THAN 50 DEEP. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE MAXIMUM PARENTHESIS NESTING LEVEL EXCEEDED. ERROR SCAN FOR THIS FORMAT STOPS HERE. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE MAY NOT BE FUNCTION, EXTERNAL, F.P. OR IN BLANK COMMON. | DATA statement tried to initialize an illegal variable. | See DETAILS column. | Optimizing mode compiler. |
| FE MISSING OR SYNTAX ERROR IN LIST OF TRANSFER LABELS. | Bad or no labels on list for assigned or computed GO TO. | Self-evident. | Optimizing mode compiler. |
| FE MISSING, BAD, OR OUT OF RANGE LABEL ON DO STATEMENT. | Self-evident. | Check DO loops for missing label, mispunched label, or incorrect nesting. | Optimizing mode compiler. |
| FE MORE THAN ONE RELATIONAL OPERATOR IN A RELATIONAL EXPRESSION. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE MORE THAN 49 FILES ON PROGRAM CARD OR 63 PARAMETERS ON A SUB-ROUTINE OR FUNCTION CARD. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE MORE THAN 63 ARGUMENTS IN ARGUMENT LIST. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NAMELIST STATEMENT SYNTAX ERROR. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NO MATCHING LEFT PARENTHESIS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NO MATCHING RIGHT PARENTHESIS IN ARGUMENT LIST. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NO MATCHING RIGHT PARENTHESIS IN SUBSCRIPT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NO MATCHING RIGHT PARENTHESIS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NON DIMENSIONED NAME APPEARS FOLLOWED BY LEFT PAREN. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NON-STANDARD RETURN STATEMENT MAY NOT APPEAR IN A FUNCTION SUBPROGRAM. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NUMBER OF ACTUAL PARAMETERS PLUS RETURNS EXCEED 63. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE NUMBER OF CHARACTERS IN AN ENCODE/DECODE STATEMENT MUST BE AN INTEGER CONSTANT OR VARIABLE. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE NUMBER OF SUBSCRIPTS IS INCOMPATIBLE WITH THE NUMBER OF DIMENSIONS DURING EQUIVALENCING. | Number of subscripts in EQUIVALENCE statement must be less than or equal to the number of dimensions. | Self-evident. | Optimizing mode compiler. |
| FE ONLY LIST DIRECTED OUTPUT STATEMENTS MAY END WITH A COMMA. | Self-evident. | Check for incomplete statement; missing continuation. | Optimizing mode compiler. |
| FE ONLY ONE SYMBOLIC NAME IN EQUIVALENCE GROUP. | Symbolic names must appear in pairs. | Self-evident. | Optimizing mode compiler. |
| FE PARAMETER ON NON-STANDARD RETURN STATEMENT IS NOT A RETURNS FORMAL PARAMETER. | The parameter on a non-standard return must appear in a RETURNS parameter list. | Self-evident. | Optimizing mode compiler. |
| FE PRECEDING CHARACTER ILLEGAL AT THIS POINT IN STRING. ERROR SCAN FOR THIS FORMAT STOPS HERE. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE PRECEDING CHARACTER ILLEGAL. SCALE FACTOR EXPECTED. ERROR SCAN FOR THIS FORMAT STOPS HERE. | The scale factor must be an integer constant followed by P. | See DETAILS column. | Optimizing mode compiler. |
| FE PRECEDING HOLLERITH COUNT IS EQUAL TO ZERO. ERROR SCAN FOR THIS FORMAT STOPS HERE. | Self-evident. | Check for mispunched Hollerith count. | Optimizing mode compiler. |
| FE PRECEDING HOLLERITH INDICATOR IS NOT PRECEDED BY A COUNT. SCANNING STOPS HERE. | Self-evident. | Check for mispunched Hollerith count. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES. | The label used in a DO statement follows another usage, i.e., FORMAT. | Self-evident. | Optimizing mode compiler. |
| FE PROGRAM OR SUBROUTINE NAME MAY NOT BE REFERENCED IN A DECLARATIVE STATEMENT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE RECORD LENGTH IS GREATER THAN 131,071. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE REFERENCED LABEL IS MORE THAN FIVE CHARACTERS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE RETURN STATEMENT APPEARS IN MAIN PROGRAM. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE RETURNS LIST ERROR. | Self-evident. | Check for syntax error; non-label parameter. | Optimizing mode compiler. |
| FE RETURNS OR EXTERNAL NAMES MAY NOT APPEAR IN DECLARATIVE STATEMENTS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE RIGHT PARENTHESIS FOLLOWED BY A NAME, CONSTANT, OR LEFT PARENTHESIS. | Incorrect use of parenthesis. | Self-evident. | Optimizing mode compiler. |
| FE SIMPLE VARIABLE OR CONSTANT FOLLOWED BY LEFT PARENTHESIS. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Signnificance | Action | Issued By |
|---|---|---|---|
| FE STATEMENT TOO LONG. | Maximum of 19 continuation lines allowed. | Self-evident. | Optimizing mode compiler. |
| FE SUBROUTINE NAME REFERRED TO BY CALL IS USED ELSEWHERE AS A NON-SUBROUTINE NAME. | Self-evident. | Check for subroutine name used as a variable name. | Optimizing mode compiler. |
| FE SYMBOLIC NAME HAS TOO MANY CHARACTERS. | Maximum of 7 characters is allowed. | Self-evident. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN ASF DEFINITION. | Arithmetic statement function definition contains an error. | Self-evident. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN DATA ITEM LIST. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN DATA STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN DUMMY ARGUMENT LIST OF STATEMENT FUNCTION. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN EQUIVALENCE STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN IMPLIED DO NEST. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN INPUT/OUTPUT STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE SYNTAX ERROR IN LOADER DIRECTIVE. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE SYNTAX ERROR IN SUBSCRIPT LIST, MUST BE OF FORM CON1*VAR*CON2. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE TAB SETTING IS GREATER THAN 131,071. SCANNING STOPS. | The tab control specification Tn contains an error. | Self-evident. | Optimizing mode compiler. |
| FE TABLE OVERFLOW (ARLSZ) – SIMPLIFY EXPRESSION OR SEE ANALYST. | The expression has caused a compiler table to overflow. | Self-evident. | Optimizing mode compiler. |
| FE TABLE OVERFLOW (CONSTORS) – SIMPLIFY STATEMENT OR SEE ANALYST. | The statement has caused a compiler table to overflow. | Self-evident. | Optimizing mode compiler. |
| FE TABLE OVERFLOW (MXFRSTB) – SIMPLIFY EXPRESSION OR SEE ANALYST. | The expression has caused a compiler table to overflow. | Self-evident. | Optimizing mode compiler. |
| FE TABLE OVERFLOW (MXOSE) – SIMPLIFY EXPRESSION OR SEE ANALYST. | The expression has caused a compiler table to overflow. | Self-evident. | Optimizing mode compiler. |
| FE THE CONTROL VARIABLE OF A DO OR DO IMPLIED LOOP MUST BE A SIMPLE INTEGER VARIABLE. | Self-evident. | Check DO statement; expressions and non-integer variables cannot be used. | Optimizing mode compiler. |
| FE THE EXPRESSION IN A LOGICAL IF IS NOT TYPE LOGICAL. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE THE FIELD FOLLOWING STOP OR PAUSE MUST BE 5 OR LESS OCTAL DIGITS OR A QUOTE-DELIMITED STRING. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE THE OPERATOR INDICATED (-,+,*,/, OR **) MUST BE FOLLOWED BY A CONSTANT, NAME OR LEFT PARENTHESIS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE THE STATEMENT IN A LOGICAL IF MAY BE ANY EXECUTABLE STATEMENT OTHER THAN A DO OR ANOTHER LOGICAL IF. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE THE SYNTAX OF DO PARAMETERS MUST BE I=M1,M2,M3 OR I=M1,M2. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE THE TERMINAL STATEMENT OF THIS DO PRECEDES IT. | The terminal statement of a DO loop must follow the DO statement. | Check for incorrect labeling. | Optimizing mode compiler. |
| FE THE TYPE OF THIS IDENTIFIER IS NOT LEGAL FOR ANY EXPRESSION. | Name used in expression has been defined as a non-variable, eg, statement function. | Self-evident. | Optimizing mode compiler. |
| FE THE VALUE OF THE PARITY INDICATOR IN A BUFFER I/O STATEMENT MUST BE 0 OR 1. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE THIS ASSIGN STATEMENT HAS IMPROPER FORMAT, ONLY ALLOWABLE IS (ASSIGN LABEL TO VARIABLE). | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE THIS NAME MAY NOT BE USED IN A DATA STMT. | DATA statement variable name is program or subprogram name. | Self-evident. | Optimizing mode compiler. |
| FE THIS OPERATOR (.NOT. OR A RELATIONAL) MUST BE FOLLOWED BY A CONSTANT, NAME, LEFT PAREN, + OR –. | Self-evident. | Check the statement for a syntax error. | Optimizing mode compiler. |
| FE THIS PROGRAM UNIT CALLS ITSELF. | Self-evident. | Check for a function or subroutine reference with the same name as the containing function or subroutine. | Optimizing mode compiler. |
| FE THIS STATEMENT MAKES AN ILLEGAL TRANSFER INTO A PREVIOUS DO LOOP. | A branch cannot be made into a previous DO loop unless the branch is within the extended range of the DO. | Self-evident. | Optimizing mode compiler. |
| FE THIS STATEMENT TYPE IS ILLEGAL IN BLOCK DATA SUBPROGRAM. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| FE TOO MANY LABELED COMMON BLOCKS, ONLY 125 BLOCKS ARE ALLOWED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE TOO MANY SUBSCRIPTS IN ARRAY REFERENCE. | The number of subscripts in an array reference cannot exceed the number of declared dimensions. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE TOTAL RECORD LENGTH IS GREATER THAN 131,071. SCANNING STOPS. | FORMAT statement has a record width that is too large. | Break the FORMAT down to use two I/O statements or use a slash to create a new record. | Optimizing mode compiler. |
| FE UNDEFINED STATEMENT LABEL(S). SEE LIST BELOW. | The label(s) are referenced in branch statements but do not appear as labels anywhere in the program. | Determine which statements are missing labels; branch statements might specify incorrect labels. | Optimizing mode compiler. |
| FE UNIT NUMBER MUST BE BETWEEN 1 AND 99 INCLUSIVE. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE UNIT NUMBER OR PARITY INDICATOR MUST BE AN INTEGER CONSTANT OR VARIABLE. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE UNMATCHED PARAMETER COUNT IN A REFERENCE TO THIS STATEMENT FUNCTION. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE UNMATCHED PARENTHE-SIS. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE UNRECOGNIZED OPERATOR. | The indicated arithmetic, logical, or relational operator is incorrectly specified. | Self-evident. | Optimizing mode compiler. |
| FE UNRECOGNIZED STATE-MENT. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE USE OF THIS PROGRAM OR SUBROUTINE NAME IN AN EXPRESSION. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| FE VALUE OF ARRAY SUBSCRIPT IS .LT. 1 OR .GT. DIMENSIONALITY IN IMPLIED DO NEST. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FE VARIABLE IN ASSIGN OR ASSIGNED GO TO IS ILLEGAL. | Only statement labels can appear in ASSIGN statement. | Self–evident. | Optimizing mode compiler. |
| FE VARIABLE SUBSCRIPTS MAY NOT APPEAR WITHOUT DO LOOPS. | DATA statement processor expected an implied DO. | Self–evident. | Optimizing mode compiler. |
| FE WAS LAST CHARACTER SEEN AFTER TROUBLE. REMAINDER OF STATEMENT IGNORED. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| FE ZERO IS SPECIFIED AS REPEAT COUNT. SCANNING STOPS. | The repeat count in a FORMAT statement must be GT zero. | Self–evident. | Optimizing mode compiler. |
| FE ZERO LEVEL RIGHT PARENTHESIS MISSING. SCANNING STOPS. | Unbalanced parenthesis. | Self–evident. | Optimizing mode compiler. |
| FE ZERO STATEMENT LABELS ARE ILLEGAL. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE + OR – SIGN MUST BE FOLLOWED BY A CONSTANT. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| FE .NOT. MAY NOT BE PRECEDED BY NAME, CONSTANT, OR RIGHT PARENS. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| I A HOLLERITH CONSTANT IS AN OPERAND OF AN ARITHMETIC OPERATOR. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| I A TYPE WAS DECLARED FOR THIS VARIABLE OR FUNCTION. THIS DECLARATION IGNORED. | If a type is declared more than once, the first declaration is assumed. | Self–evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I AN IF STATEMENT MAY BE MORE EFFICIENT THAN A 2 OR 3 BRANCH COMPUTED GO TO STATEMENT. | In certain cases, an IF statement generates less object code than a 2 or 3 branch computed GO TO statement. | Self-evident. | Optimizing mode compiler. |
| I ARGUMENT COUNT INCONSISTENT WITH PRIOR USAGE. | Subroutine call argument list has a different number of arguments in a prior call. | Self-evident. | Optimizing mode compiler. |
| I ARRAY NAME OPERAND NOT SUBSCRIPTED, FIRST ELEMENT WILL BE USED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I ARRAY REFERENCE OUTSIDE DIMENSION BOUNDS. | This reference can give unpredictable results. | Increase the declared dimension or correct the reference. | Optimizing mode compiler. |
| I CHARACTER BOUNDS REVERSED IN IMPLICIT STATEMENT. | The bounds should be ordered alphabetically. | Self-evident. | Optimizing mode compiler. |
| I COMMA MISSING BEFORE VARIABLE INDICATED. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| I CONSTANT TOO LONG. HIGH ORDER DIGITS RETAINED, BUT SOME PRECISION LOST. | The constant length exceeds the capability of the computer. | Self-evident. | Optimizing mode compiler. |
| I CONTROL VARIABLE IN COMMON OR EQUIVALENCED, OPTIMIZATION MAY BE INHIBITED. | The control variable of a DO statement appears in a COMMON or EQUIVALENCE statement. | Self-evident. | Optimizing mode compiler. |
| I DATA ITEM LIST EXCEEDS VARIABLE LIST, EXCESS CONSTANTS IGNORED. | Self-evident. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I DATA VARIABLE LIST EXCEEDS ITEM LIST, EXCESS VARIABLES NOT INITIALIZED. | Self-evident. | The missing data items should be inserted. | Optimizing mode compiler. |
| I DIMENSIONAL RANGE IS EXTENDED FOR EQUIVALENCING PURPOSES. | Equivalence processing has caused a dimensional array to have its length increased. | Self-evident. | Optimizing mode compiler. |
| I FIELD WIDTH IS GREATER THAN 137 CHARACTERS. IT MAY EXCEED THE I/O DEVICE CAPACITY. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I FIELD WIDTH OF A CONVERSION DESCRIPTOR SHOULD BE AS LARGE AS THE MINIMUM SPECIFIED FOR THAT DESCRIPTOR. | Self-evident. | Reduce program size or requested field length. | Optimizing mode compiler. |
| I FILE LENGTH REQUESTED IS TOO LARGE. STANDARD LENGTH OF 2000B SUBSTITUTED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I FRACTIONAL DIGIT COUNT MISSING FROM CONVERSION DESCRIPTOR. DEPENDING ON DESCRIPTOR, ONE OR ZERO ASSUMED. | Incorrect field width specification in FORMAT statement. | See DETAILS column. | Optimizing mode compiler. |
| I FWA AND LWA NOT IN SAME ARRAY, EQUIVALENCE CLASS, OR COMMON BLOCK. | Will produce unpredictable results. | Self-evident. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I HOLLERITH CONSTANT .GT. 10 CHARACTERS, EXCESS CHARACTERS INITIALIZED INTO SUCCEEDING WORDS. | A word can contain up to 10 characters. This can cause overstoring of other information in succeeding words. | Reduce constant length if succeeding words must be preserved. | Optimizing mode compiler. |
| I INVOLVED IN REDUNDANT EQUIVALENCING. | The indicated variable was equivalenced more than once to the same variable. | Check all equivalence statement. | Optimizing mode compiler. |
| I I/O BUFFER LENGTH SPECIFICATION IS NOT MEANINGFUL - - VALUE IGNORED. | Self-evident. | Buffer size specification must have value between 0 and $2^{18} - 1$. | Optimizing mode compiler. |
| I I/O FILE NOT DEFINED. | The file has not been declared in a PROGRAM statement. | This message should be ignored for all programs residing in primary or secondary overlays. For programs that reside in the main overlay or are not part of an overlay structure, the indicated file must be declared in the PROGRAM statement. | Optimizing mode compiler. |
| I LEVEL CONFLICTS WITH PREVIOUS DECLARATION ORIGINAL LEVEL RETAINED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I LOWER LIMIT .GE. UPPER LIMIT, ONE TRIP LOOP. | If the lower limit of a loop is greater than the upper limit, the loop is executed once. | Self-evident. | Optimizing mode compiler. |
| I MASK ARGUMENT MUST BE NONNEGATIVE AND LESS THAN 61. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I MAY NOT BE USED IN A DEBUG STATEMENT. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| I MORE STORAGE REQUIRED BY DO STATEMENT PROCESSOR FOR OPTIMIZATION. | Loop code is possibly sub-optimized. | Increase field length and recompile. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I NO DIGIT PRECEDED X-FIELD 1X ASSUMED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I NO END CARD, END LINE ASSUMED. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I NON-INNER LOOP BEGINNING AT THIS CARD IS ENTERED. | A DO loop should be entered only through a DO statement. | Self-evident. | Optimizing mode compiler. |
| I NOT ALL ITEMS IN THIS COMMON BLOCK OCCUR IN LEVEL STATEMENTS. | Same level is assumed for all members of the block. | Self-evident. | Optimizing mode compiler. |
| I NUMERIC FIELD FOLLOWING TAB SETTING DESIGNATOR IS EQUAL TO ZERO. COLUMN ONE WILL BE ASSUMED. | Self-evident. | Ensure that the correct setting is specified. | Optimizing mode compiler. |
| I NUMERIC FIELD OMITTED FROM PRECEDING SCALE FACTOR. ZERO SCALE FACTOR ASSUMED. | Self-evident. | Ensure that the correct scale factor is specified. | Optimizing mode compiler. |
| I PRECEDING FIELD WIDTH IS ZERO. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| I PRECEDING FIELD WIDTH SHOULD BE 7 OR MORE. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| I PRECEDING SCALE FACTOR EXCEEDS THE LIMIT OF REPRESENTATION WITHIN THE MACHINE. | Self-evident. | See DETAILS column. | Optimizing mode compiler. |
| I PRESENT USE IN CONTEXT OF THIS NAME DOES NOT MATCH PREVIOUS OCCUR-RENCES IN DEBUG STMTS. | Variable or subroutine name has been used in mixed contexts. | Self-evident. | Optimizing mode compiler. |

TABLE B–1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I PREVIOUSLY DIMENSIONED ARRAY. FIRST DIMENSIONS WILL BE RETAINED. | Self–evident. | All but the desired DIMENSION declaration should be removed. | Optimizing mode compiler. |
| I SEPARATOR MISSING. SEPARATOR ASSUMED HERE. | A / or , is missing. | Self–evident. | Optimizing mode compiler. |
| I SHIFT ARGUMENT MUST BE GREATER THAN –61 AND LESS THAN 61. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| I SINGLE WORD CONSTANT MATCHED WITH DOUBLE OR COMPLEX VARIABLE. PRECISION LOST. | Double and complex variables require 2 words. | Use DOUBLE or COMPLEX type declarations. | Optimizing mode compiler. |
| I SPURIOUS CHARACTERS AFTER CONTINUE IGNORED. | Self–evident. | See DETAILS column. | Optimizing mode compiler. |
| I SUPERFLUOUS SCALE FACTOR ENCOUNTERED BEFORE THE CURRENT SCALE FACTOR. | The superfluous scale factor is ignored. | See DETAILS column. | Optimizing mode compiler. |
| I TAB SETTING MAY EXCEED RECORD SIZE, DEPENDING ON USE. | Self–evident. | Reduce tab setting. | Optimizing mode compiler. |
| I THE UPPER LIMIT AND CONTROL VARIABLES OF THIS DO ARE THE SAME, PRODUCING A NON–TERMINATING LOOP. | Self–evident. | Self–evident. | Optimizing mode compiler. |
| I THERE IS NO PATH TO THIS STATEMENT. | The statement cannot be executed. | Check for logic error; possible missing label. | Optimizing mode compiler. |

TABLE B-1. OPTIMIZING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| I THIS IF DEGENERATES INTO A SIMPLE TRANSFER TO THE LABEL INDICATED. | This statement can be replaced by a GO TO statement. | Self-evident. | Optimizing mode compiler. |
| I THIS STATEMENT BRANCHES TO ITSELF. | This statement results in an infinite loop. | Self-evident. | Optimizing mode compiler. |
| I THIS STATEMENT FORM IS OBSOLETE. USE A LEVEL 3 STATEMENT. | Program contains an obsolete "TYPE ECS" statement. | Self-evident. | Optimizing mode compiler. |
| I THIS STATEMENT MAY REDEFINE A CURRENT LOOP CONTROL VARIABLE OR PARAMETER, OPTIMIZATION INHIBITED. | The variable to the left of the = is also used as a DO loop control variable. | Substitute another variable for the control variable in this statement. | Optimizing mode compiler. |
| I THIS STATEMENT REDEFINES A CURRENT LOOP CONTROL VARIABLE OR PARAMETER. | The variable to the left of the = is also used as a DO loop control variable. | Substitute another variable for the control variable in this statement. | Optimizing mode compiler. |
| I TOTAL RECORD LENGTH IS GREATER THAN 137 CHARAC-TERS. IT MAY EXCEED THE I/O DEVICE CAPACITY. | Self-evident. | Self-evident. | Optimizing mode compiler. |
| I VARIABLE NOT DECLARED IN LABEL COMMON. | All variables within block data subprogram must be declared in COMMON. | Self-evident. | Optimizing mode compiler. |
| I X-FIELD PRECEDED BY A ZERO. NO SPACING OCCURS. | Self-evident. | Ensure that correct spacing is specified. | Optimizing mode compiler. |
| I *PROGRAM START. (INPUT, OUTPUT)* ASSUMED WHEN HEADER STATEMENT IS DEFECTIVE OR OMITTED. | The PROGRAM, FUNCTION, or SUBROUTINE statement contains an error or has been omitted. | Self-evident. | Optimizing mode compiler. |
| I ***DUE TO THE MANY ERRORS NOTED, ONLY THOSE WHICH ARE FATAL WILL BE LISTED HEREAFTER. | Informative messages will not be listed. | Correct as many errors as possible and recompile. | Optimizing mode compiler. |

## TIME-SHARING MODE DIAGNOSTICS

When time-sharing mode is selected, compilation error messages are intermixed with the source listing as they are detected. The format of the error message is:

severity * text

The severity indicator is truncated to the first letter if page width (as specified by the PW control statement option) is less than 126. The indicators are:

| | |
|---|---|
| FATAL | Error is fatal to execution. |
| WARNING | Error is severe, but not fatal. Syntax is incorrect, but probable meaning is presumed. |
| NOTE | Minor syntax error or omission. |
| ANSI | Usage does not conform to ANSI X3.9 - 1966 FORTRAN specification. Listed only if EL=A list option is specified on the FTN control statement. |

In addition to the above, certain unsuppressible nonfatal diagnostics are always listed regardless of the EL specification on the FTN control statement; they are indicated by five dashes as the severity indicator.

The compilation diagnostics produced in time-sharing mode are shown in table B-2. Ellipses denoted by ..... are replaced in an actual message by items from the relevant source statement, distinguished by a preceding ↱ (or ___ ). Micro names delimited by ≠ pairs (such as ≠MAX.SARG≠) are replaced by numerical values supplied by the system.

Example:

```
       1                 SUBROUTINE SUB(A,B)
                         DIMENSION A(2)
                         COMMEN B(4)
-----      *          MISSPELLED KEYWORD -- ►COMMON   ASSUMED
FATAL      *          USAGE CONFLICT -- ►B IS DUMMY-ARG AND CANNOT BE COMMON
                         DO 10, I=1,4
WARNING    *          COMMA AFTER DO LABEL IGNORED
ANSI       *          COMMA NOT PERMITTED AFTER DO LABEL
       5                 WRITE(4,11) A,B
                      11    FORMAT(2A10)
                         VARIABL=A+B
ANSI       *          ARRAY ►A MISSING SUBSCRIPT -- FIRST ELEMENT ASSUMED
ANSI       *          ARRAY ►B MISSING SUBSCRIPT -- FIRST ELEMENT ASSUMED
                         CONTINUE
NOTE       *          CONTINUE WITH NO STATEMENT LABEL -- IGNORED
-----      *          END LINE ABSENT
ANSI       *          CONTROL FLOW INTO END LINE NOT PERMITTED
FATAL      *          STATEMENT LABEL  .10      REFERENCED BUT NOT DEFINED
FATAL      *          DO LOOP  .10      NOT TERMINATED BEFORE END OF PROGRAM

--COMMON BLOCKS--

              0B  //

--EXTERNALS--

                   OUTCI.    TAPE4=

--STATEMENT LABELS--

          .10    DU      0B          .11    F      4B

--VARIABLE MAP--

     A          R AU     0B        2          B        R AU    0B          4
     I          I  U    13B                   OUTCI.   -          EXTERNAL.
     SUB        -        0B ENTRY             TAPE4=   -          EXTERNAL.
     VARIABL    R       14B


   15B  PROGRAM-UNIT LENGTH        9 SYMBOLS

 41000B CM STORAGE USED          .028 SECONDS

     3  FORTRAN ERRORS IN SUB
```

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI A TERM IN SUBSCRIPT . . . ON . . . IS NOT INTEGER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI ANSI REQUIRES AN I/O LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI ANSI REQUIRES THE WORD PRECISION. | The DOUBLE statement is non-ANSI. | Change to DOUBLE PRECISION. | Time-sharing mode compiler. |
| ANSI ARRAY . . . MISSING SUBSCRIPT – FIRST ELEMENT ASSUMED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI ARRAY . . . MUST HAVE IMPLIED LOOP. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMMA AFTER VARIABLE NAME IN ASSIGNED GO TO IS REQUIRED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMMA BEFORE VARIABLE NAME IN COMPUTED GO TO IS REQUIRED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMMA NOT PERMITTED AFTER DO LABEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMPLEX EXPRESSION IN AN IF STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMPUTED GO TO INDEX MUST BE INTEGER. CONVERSION OF HOLLERITH CONSTANTS. | Index is of incorrect type. | Self-evident. | Time-sharing mode compiler. |
| ANSI COMPUTED GO TO INDEX MUST BE VARIABLE. | Constant or expression not permitted. | Self-evident. | Time-sharing mode compiler. |
| ANSI CONTROL FLOW INTO END LINE NOT PERMITTED. | Self-evident. | Insert RETURN or STOP. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI DATA VARIABLE . . . DOES NOT MATCH CONSTANT TYPE. | Self-evident. | Declare proper type in type statement. | Time-sharing mode compiler. |
| ANSI DECIMAL POINT IS NOT SPECIFIED FOR THE CONVERSION CODE . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI EQUAL SIGN = IS SPECIFIED FOR A DIGIT. | Probable syntax error in format specification. | Self-evident. | Time-sharing mode compiler. |
| ANSI EXPONENT LENGTH IS SPECIFIED FOR THE CONVERSION CODE . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI FORM OF SUBSCRIPT . . . ON . . . NOT DEFINED IN ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI FORMAT INDICATOR . . . MUST BE ARRAY. | Variable format indicator must be an array. | Self-evident. | Time-sharing mode compiler. |
| ANSI HOLLERITH ARGUMENT IS NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI HOLLERITH CONSTANT IN EXPRESSION NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI HOLLERITH CONSTANT LONGER THAN 1 WORD. | Excess characters are truncated. | Self-evident. | Time-sharing mode compiler. |
| ANSI HOLLERITH DIMENSION FOR . . . | A Hollerith constant has been used for a dimension. | Self-evident. | Time-sharing mode compiler. |
| ANSI LIST DIRECTED I/O IS NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI MASK EXPRESSION NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI MINIMUM DIGITS SPECIFIED FOR THE CONVERSION CODE . . . | FORTRAN Extended FORMAT minimum digit specified has been used. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI MULTIPLE ASSIGNMENT IS NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI MULTIPLE STATEMENT PER CARD NOT PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI NAMELIST I/O IS NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI NON-ANSI FORM OF BLOCK DATA STATEMENT | The form BLOCK DATA name is invalid in ANSI FORTRAN. | Use BLOCK DATA. FORTRAN assigns name BLKDAT. | Time-sharing mode compiler. |
| ANSI NON-ANSI HOLLERITH FORM. | Hollerith constant delimited by ≠ character is non-ANSI. | Change to Hollerith count of the form nH. | Time-sharing mode compiler. |
| ANSI NON-ANSI SYNTAX IN THIS DATA STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI NON-ANSI TYPE COMBINATIONS WITH . . . . . OPERATOR. | Self-evident. | Use type declaration to ensure correct type. | Time-sharing mode compiler. |
| ANSI NUMERIC BLOCK NAME NOT PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI OBJECT OF IF IS ILLEGAL DO TERMINATOR. | The last statement of a DO loop cannot be branched to by an IF statement. | Self-evident. | Time-sharing mode compiler. |
| ANSI OCTAL DATA TYPE NOT DEFINED IN ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI OCTAL DIGITS REQUIRED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI PAREN REPEAT LIST IS NOT PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI PAUSE MAY NOT BE A DO TERMINAL. | A DO loop must not end with a PAUSE statement. | Self-evident. | Time-sharing mode compiler. |
| ANSI REDEFINITION OF PARAMETER . . . . . USED AS DIMENSION INDICATOR IS NON-ANSI. | Integer variable used as an array declarator in a DIMENSION statement was altered in a subsequent expression. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI RETURN IN MAIN PROGRAM. | Self-evident. | Replace with STOP. | Time-sharing mode compiler. |
| ANSI S CODE IS SPECIFIED. | S-code is non-ANSI. | Self-evident. | Time-sharing mode compiler. |
| ANSI SAME NAME USED AS FUNCTION AND SUBROUTINE. | Usage violates ANSI class restrictions. | Reference the sub-program either as a function or as a subroutine. | Time-sharing mode compiler. |
| ANSI SHORT FORMS OF LOGICAL OPERATORS OR CONSTANT NOT ALLOWED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI SKIP COUNT FOR X CODE IS PRECEDED BY . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI STATEMENT IS NOT DEFINED IN ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI T CODE IS NULL OR ZERO COLUMN POINTER RESET AT 1. | Character T must be preceded by integer. | Self-evident. | Time-sharing mode compiler. |
| ANSI THE WORD TYPE IS NOT PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI THIS FORM OF DATA STATEMENT NOT PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI X CODE PRECEDED BY NON-DIGIT – 1 X ASSUMED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI X CODE PRECEDED BY ZERO – X CODE IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI 2 BRANCH IF IS NON-ANSI. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| ANSI 7 CHARACTER SYMBOL . . . IS NON-ANSI. | ANSI allows only 6 characters. | Self-evident. | Time-sharing mode compiler. |
| ANSI ... BLANK STATEMENTS WERE IGNORED. | ANSI does not permit blank lines. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| ANSI ... INDEX PARAMETER NOT SIMPLE INTEGER VARIABLE OR CONSTANT. | Index parameter cannot be expression or type other than integer. | Self-evident. | Time-sharing mode compiler. |
| ANSI ... IS DEFINED TO BE A BASIC EXTERNAL FUNCTION. | The indicated user-defined name conflicts with a FORTRAN function name. | Self-evident. | Time-sharing mode compiler. |
| ANSI ... IS DEFINED TO BE INTRINSIC. | The indicated user-defined name conflicts with a FORTRAN function name. | Self-evident. | Time-sharing mode compiler. |
| ANSI ... IS SPECIFIED AS CONVERSION CODE. | Illegal format specification. | Self-evident. | Time-sharing mode compiler. |
| CONTINUE ... SYNTAX ERROR IN IMPLIED DO ON ARRAY ... | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL A C/-LIST DIRECTIVE CANNOT BE FOLLOWED BY A CONTINUATION LINE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARGUMENT COUNT NOT EQUAL TO THAT DEFINED FOR INTRINSIC ... | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARGUMENT COUNT ON ... MUST BE MORE THAN ONE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARGUMENT COUNT ON ... EXCEEDS ≠MAX.SARG≠ | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARGUMENT MODE MUST AGREE WITH TYPE DEFINED FOR LIBRARY FUNCTION ... | Self-evident. | Check definition of function to determine correct argument type. | Time-sharing mode compiler. |
| FATAL ARITHMETIC IF HAS STATEMENT AS OBJECT. | Object must be of form s1,s2, or s1,s2,s3, where s1,s2,s3 are statement labels. | Self-evident. | Time-sharing mode compiler. |

TABLE B–2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL ARRAY DECLARATION FOR . . . MISSING RIGHT PAREN. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . DIMENSION INDICA- TOR NOT INTEGER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . DIMENSION INDICATOR . . . EXCEEDS 2** 17–1. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . EXCEEDS ≠ 3 ≠ DIMENSIONS. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . HAS A VARIABLE SUBSCRIPT WITH NO IMPLIED LOOP. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . NULL OR ZERO DIMENSION INDICATOR. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ARRAY . . . SIZE EXCEEDS 2** 17–1. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ASF EXPRESSION TYPE CON- FLICTS WITH FUNCTION TYPE. | If function is logical, expression must be logical. If function is not type logical, expression must not be relational or logical. | Self-evident. | Time-sharing mode compiler. |
| FATAL BUFFER DIRECTION INDICATOR MUST BE IN OR OUT. | BUFFER statement incorrect; correct form is BUFFER IN or BUFFER OUT. | Self-evident. | Time-sharing mode compiler. |
| FATAL BUFFER I/O ADDRESS MUST BE VARIABLE. | BUFFER I/O address must not be constant or expression. | Self-evident. | Time-sharing mode compiler. |
| FATAL BUFFER I/O LWA MUST BE GREATER THAN OR EQUAL TO FWA. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL BUFFER I/O PARITY INDICATOR MUST BE INTEGER CONSTANT OR VARIABLE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL BUFFER I/O PARITY INDICATOR VALUE MUST BE ZERO OR 1. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL CALL STATEMENT MISSING ROUTINE NAME. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL CHARACTER . . . FOUND AFTER TERMINAL RIGHT PAREN. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL CHARACTER . . . NOT DEFINED IN STANDARD FORTRAN – CARD SCAN STOPPED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL COMMA MUST FOLLOW LEVEL NUMBER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL COMMA OR E.O.S. MUST FOLLOW LEVEL LIST NAME. | Comma or end of statement expected; line contains extraneous information. | Self-evident. | Time-sharing mode compiler. |
| FATAL COMPLEX MUST ONLY BE RAISED TO INTEGER POWER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL COMPUTED GO TO INDEX MUST NOT BE LOGICAL. | Index is of incorrect type. | Self-evident. | Time-sharing mode compiler. |
| FATAL CONFLICT IN EQUIVALENCE SPECIFICATION FOR . . . | The indicated EQUIVALENCE specification is inconsistent with a previous EQUIVALENCE specification. | Check all EQUIVALENCE statements. | Time-sharing mode compiler. |
| FATAL CONSTANT CANNOT BE CONVERTED. | Constant contains syntax error. | Self-evident. | Time-sharing mode compiler. |
| FATAL CONSTANT DIVIDE BY ZERO – RESULTS SET TO INFINITE. | Division by zero is an undefined operation. | Self-evident. | Time-sharing mode compiler. |
| FATAL CONSTANT IN INPUT LIST IS ILLEGAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL COUNT FOR H CODE ZERO OR MISSING – SCAN STOPPED. | Hollerith constant must be positive integer. | Self-evident. | Time-sharing mode compiler. |
| FATAL DATA INTO . . . IS ILLEGAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL DATA SUBSCRIPT LIST SYNTAX ERROR. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL DATA VARIABLE LIST SYNTAX ERROR. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL DO CONTROL INDEX MUST BE SIMPLE INTEGER VARIABLE. | DO control index cannot be expression, constant, or type other than integer. | Self-evident. | Time-sharing mode compiler. |
| FATAL DO LOOP . . . PREVIOUSLY DEFINED – ILLEGAL NESTING. | The label is used in a previous DO statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL DO LOOP . . . NOT TERMINATED BEFORE END OF PROGRAM. | Do loop terminator missing. | Rewrite statement or use different variable for DO index. | Time-sharing mode compiler. |
| FATAL DO STATEMENT SYNTAX – EXPECTED CONTROL INDEX – FOUND E.O.S. | The DO statement is incomplete. | Self-evident. | Time-sharing mode compiler. |
| FATAL DUMMY ARGUMENT . . . CAN OCCUR ONLY ONCE IN . . . DEFINITION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL DUMMY ARGUMENT . . . MUST BEGIN WITH LETTER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL DUMMY ARGUMENT . . . PREVIOUSLY DEFINED. | A dummy argument can only appear once in the FUNCTION or SUBROUTINE statement. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL EQUAL SIGN MUST BE FOLLOWED BY NAME, NUMBER, OR SLASH. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXCESS LEFT PAREN IN I/O LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXCESS LEFT PAREN IN I/O SUBSCRIPT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXCESS RIGHT PAREN IN I/O LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXCESS SUBSCRIPTS ON EQUIVALENCE VARIABLE . . . | EQUIVALENCE variable has more subscripts than declared dimension. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXECUTABLE STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED COMMA AFTER COUNT – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED COMMA AFTER FORMAT INDICATOR – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED COMMA FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED COMMA OR SLASH FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED COMMA FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED E.O.S. OR RETURNS PARAMETER FOUND . . . | Syntax error in RETURNS list. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL EXPECTED E.O.S. – FOUND AND IGNORED . . . | End of statement expected. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED LEFT PAREN BEFORE COUNT – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED LEFT PAREN FOR AN ARGUMENT LIST, FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED LEFT PAREN OR COMMA AFTER ROUTINE NAME FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED LEFT PAREN OR COMMA FOUND . . . | Self-evident. | Check for error in argument list. | Time-sharing mode compiler. |
| FATAL EXPECTED LEFT PAREN – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED NAME – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED RETURNS FOUND . . . | Self-evident. | Check for error in SUB-ROUTINE statement. | Time-sharing mode compiler. |
| FATAL EXPECTED RIGHT PAREN AFTER STRING ADDRESS – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED RIGHT PAREN OR COMMA – FOUND . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED RIGHT PAREN – FOUND. . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPECTED SYMBOL BUT FOUND . . . – SCAN OF CARD STOPPED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL EXPONENT EXCEEDS 512. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXPRESSION IN INPUT LIST IS ILLEGAL. | An arithmetic expression cannot appear in an I/O list. | Self-evident. | Time-sharing mode compiler. |
| FATAL EXTERNAL ARGUMENT . . . MUST BE DEFINED AS EXTERNAL. | If a function, subroutine, or entry point name appears in an argument list, it must be declared EXTERNAL. | Self-evident. | Time-sharing mode compiler. |
| FATAL E.O.S. BEFORE END OF HOLLERITH COUNT. | Premature end of statement encountered. | Check for incorrect Hollerith count. | Time-sharing mode compiler. |
| FATAL FIELD WIDTH OF CONVERSION CODE . . . IS ZERO OR NOT SPECIFIED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL FORMAT DESIGNATOR MISSING. | This form of the I/O statement must specify the label FORMAT statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL FORMAT INDICATOR MUST NOT BE EXPRESSION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL FORMAT INDICATOR . . . IS NAMELIST NAME. | NAMELIST name cannot be used in ENCODE/DECODE. | Self-evident. | Time-sharing mode compiler. |
| FATAL FORMAT LABEL PREVIOUSLY REFERENCED AS CONTROL STATEMENT LABEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL FORMAT LABEL PREVIOUSLY REFERENCED AS DO STATEMENT LABEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL FUNCTION MUST HAVE AT LEAST 1 DUMMY ARGUMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL FUNCTION NAME IS NOT ASSIGNED A VALUE. | The function name must be assigned a value within the function. | Self-evident. | Time-sharing mode compiler. |
| FATAL GROUP NAME . . . PREVIOUSLY DEFINED. | The group name appears twice in the same statement or in an earlier statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL HEADER CARD NOT FIRST STATEMENT – IGNORED. | PROGRAM, SUBROUTINE, or FUNCTION must be first statement of program. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL BLOCK NAME IN COMMON STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL CHARACTER COUNT. | Must be integer constant or simple integer variable LE.150. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL CONSTANT FOLLOWING + OR – . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL FORM INVOLVING THE USE OF A COMMA. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL FORM OF EXPONENT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL FORMAT INDICATOR . . . | Must be legal statement label. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL IF STATEMENT – OBJECT MISSING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL NESTING OF DO LOOPS. | The range of an inner DO must be within the range of an outer DO. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL OBJECT OF IF – TROUBLE STARTED AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL ILLEGAL OBJECT OF LOGICAL IF. | Object must be expression or GO TO. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL RANGE – . . . . NOT LESS THAN . . . – TRUNCATED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL REFERENCE TO FORMAT STATEMENT LABEL . . . | The label was previously defined as a FORMAT label. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL REFERENCE TO STATEMENT LABEL . . . AS A FORMAT. | The label referenced as a FORMAT appears as the label of an executable statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL REPEAT CONSTANT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL SEPARATOR AFTER . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL SEPARATOR FOLLOWING DATA CONSTANT. | Separator must be ), /,or , . | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL TRANSFER INTO RANGE OF DO. | Indicated statement branches into a DO loop. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL TRANSFER TO DO . . . TERMINATOR. | A DO terminator cannot be referenced outside the DO loop. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL TRANSFER TO . . . FORMAT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL USE OF ASSIGNMENT OPERATOR. | Illegal use of equal sign. | Self-evident. | Time-sharing mode compiler. |
| FATAL ILLEGAL USE OF ENTRY . . . | ENTRY cannot be labeled; within range of DO; used as object of IF. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL ILLEGAL USE OF OPERATOR / OPERAND – . . . | Self-evident. | See items filled in. | Time-sharing mode compiler. |
| FATAL IMPLICIT STATEMENT MUST OCCUR BEFORE DECLARATIVES. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL IMPLIED DO INCREMENT MUST BE NUMERIC. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL IMPLIED DO INDEX MUST BE FOLLOWED BY EQUAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL IMPLIED DO LOWER LIMIT MUST BE NUMERIC. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL IMPLIED DO NOT TERMINATED. | Self-evident. | Check statement for syntax errors. | Time-sharing mode compiler. |
| FATAL IMPLIED DO UPPER LIMIT MUST BE NUMERIC. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL INDEX OF OUTER DO REDEFINED BY CURRENT DO. | Inner DO index is same as outer. DO index, or inner DO contains statement which redefines outer DO index. | Self-evident. | Time-sharing mode compiler. |
| FATAL INTEGER GREATER THAN 2** 48-1 IN MULTIPLY OR DIVIDE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL INTEGER GREATER THAN 2** 48-1 IN REAL EXPRESSION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL INTEGER OPERATION RESULTS IN OVERFLOW. | Results of operation exceed capacity of machine. | Self-evident. | Time-sharing mode compiler. |
| FATAL INTEGER 1, 2, OR 3 MUST FOLLOW LEVEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL I/O UNIT DESIGNATOR MUST BE INTEGER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL I/O UNIT DESIGNATOR MUST BE SIMPLE VARIABLE. | Cannot be expression or array. | Self-evident. | Time-sharing mode compiler. |
| FATAL LEFT SIDE OF EQUAL SIGN IS ILLEGAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL LEVEL CONFLICT IN COMMON BLOCK . . . | All members of a COMMON block must be assigned to same level. | Self-evident. | Time-sharing mode compiler. |
| FATAL LEVEL 3 NAME . . . MAY NOT OCCUR IN THIS STATEMENT. | Level 3 data cannot be used in expressions. | Transfer to central memory with MOVLEV call. | Time-sharing mode compiler. |
| FATAL LOADER DIRECTIVE MUST BEGIN WITH LEFT PAREN. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL LOCF ARGUMENT MUST NOT BE CONSTANT OR EXPRESSION. | LOCF argument must be a variable. | Self-evident. | Time-sharing mode compiler. |
| FATAL LOGICAL AND NON-LOGICAL OPERANDS MAY NOT BE MIXED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL LOGICAL IF MUST NOT BE OBJECT OF LOGICAL IF. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL LOGICAL OPERAND USED WITH NON-LOGICAL OPERATOR. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING COMMA AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING LABEL IN ARITHMETIC IF. | Arithmetic IF must specify 2 or 3 labels. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING LEFT PAREN AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL MISSING RIGHT PAREN AFTER FORMAT IS ASSUMED. | A right parenthesis was expected at the end of the statement. | Compiler assumes the missing parenthesis. | Time-sharing mode compiler. |
| FATAL MISSING RIGHT PAREN AFTER IMPLIED DO. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING RIGHT PAREN AFTER UNIT IS ASSUMED. | Compiler inserts the missing parenthesis. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING SLASH ON GROUP NAME. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING UNIT DESIGNATOR IN I/O STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MISSING VARIABLE OR ARRAY NAME IN LEVEL LIST. | LEVEL statement has no or illegal names. | Self-evident. | Time-sharing mode compiler. |
| FATAL MORE THAN 3 SUBSCRIPT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MULTIPLE DEFINITION OF CURRENT FORMAT NUMBER. | Self-evident. | Check FORMAT statements for duplicate label. | Time-sharing mode compiler. |
| FATAL MULTIPLE STATEMENT IGNORED AFTER LOADER DIRECTIVE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL MULTIPLY DEFINED STATEMENT LABEL . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NAME . . . IS IN EQUIV. GROUP THAT HAS LEVEL CONFLICT. | Items in an equivalence group must have same level declaration. | Self-evident. | Time-sharing mode compiler. |
| FATAL NAME . . . IS IN LEVEL EQUIVALENCE GROUP AND MUST BE COMMON. | Level 2 or 3 data must appear in COMMON or as dummy arguments. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL NAME . . IS LEVEL AND MUST BE COMMON OR DUMMY ARGUMENT. | Level 2 or 3 data must be in COMMON or in dummy argument list. | Self-evident. | Time-sharing mode compiler. |
| FATAL NAME . . . NOT IN RETURNS LIST. | The indicated name must appear in a RETURNS list in the header statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL NEGATIVE DIMENSION FOR . . . – SET TO 1. . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NESTING OF REPEAT COUNT IN DATA CONSTANT LIST IS ILLEGAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NO CHARACTERS FOUND IN . . . DELIMITED HOLLERITH STRING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NO COMMA AFTER LOWER LIMIT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NO DIMENSION FOUND FOR EQUIVALENCE VARIABLE . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL NO MATCH OF LOOP INDEX AND SUBSCRIPT. | The same variable must be used for loop index and subscripts. | Self-evident. | Time-sharing mode compiler. |
| FATAL OBJECT OF GO TO MISSING. | The GO TO does not specify a statement label. | Self-evident. | Time-sharing mode compiler. |
| FATAL ONLY LIST DIRECTED OUTPUT STATEMENTS MAY END WITH A COMMA. | The comma at the end of this statement is illegal. | Self-evident. | Time-sharing mode compiler. |
| FATAL ONLY 1 LABEL IN IF STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ONLY 9 PAREN LEVELS ALLOWED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL ONLY 19 CONTINUATION CARDS ARE PERMITTED. | Self-evident. | Self-evident. | Time-sharing mode compiler |
| FATAL ONLY 500 COMMON BLOCKS ARE PERMITTED. | Too many common blocks. | Self-evident. | Time-sharing mode compiler. |
| FATAL ONLY 63 DUMMY ARGUMENTS ARE PERMITTED – EXCESS IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL OPERAND TO ** OPERATOR MUST NOT BE LOGICAL. | Self-evident. | SElf-evident. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. | Premature end of statement. | Check for incomplete statement. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. IN ENCODE OR DECODE. | End of statement encountered; statement incomplete. | Self-evident. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. IN I/O SUBSCRIPT. | End of statement encountered; part of statement missing. | Self-evident. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. OR MISSING RIGHT PAREN. | End of statement encountered or missing right parenthesis. | Check for incomplete statement. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. – EXPECTED BLOCK NAME. | Premature end of statement. | Check for incomplete statement. | Time-sharing mode compiler. |
| FATAL PREMATURE E.O.S. – EXPECTED SYMBOL. | Premature end of statement. | Check for incomplete statement. | Time-sharing mode compiler. |
| FATAL PREVIOUS REFERENCE TO DO LABEL . . . IS ILLEGAL. | A DO label must not be referenced outside the DO loop. | Check all previous references to the label. | Time-sharing mode compiler. |
| FATAL PREVIOUS REFERENCE TO FORMAT LABEL . . . IS ILLEGAL. | The label was previously defined as a FORMAT label. | Self-evident. | Time-sharing mode compiler. |
| FATAL PREVIOUS REFERENCE TO THIS DO LABEL IS ILLEGAL. | The label of the terminal statement of a DO loop cannot be referenced by a statement outside the loop. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL PREVIOUS TRANSFER TO . . . IS FROM OUTSIDE CURRENT DO. | A label within a DO loop cannot be referenced outside the loop. | Self-evident. | Time-sharing mode compiler. |
| FATAL PROGRAM LENGTH EXCEEDS 2** 17-1. | Program exceeds machine capability. | Shorten program or use overlay structure. | Time-sharing mode compiler. |
| FATAL RANGE INDICATOR . . . MUST BE A LETTER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL RECORD LENGTH EXCEEDS 2** 17-1 COLUMNS. | Self-evident. | Check for incorrect repeat specification, Hollerith count, format specification. | Time-sharing mode compiler. |
| FATAL RECURSIVE DEFINITION OF STATEMENT FUNCTION . . . | The function appears on both sides of an equal sign. | Self-evident. | Time-sharing mode compiler. |
| FATAL REFERENCE TO B.E.F. . . . REQUIRES AN ARGUMENT LIST. | Reference to basic external function requires argument list. | Self-evident. | Time-sharing mode compiler. |
| FATAL REFERENCE TO FUNCTION . . . REQUIRES AN ARGUMENT LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL REFERENCE TO INTRINSIC . . . REQUIRES AN ARGUMENT LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL REFERENCE TO STATEMENT FUNCTION . . . HAS A NULL PARAMETER. | Function reference requires a parameter. | Self-evident. | Time-sharing mode compiler. |
| FATAL REFERENCE TO VARIABLE . . . AS A FUNCTION OR ARRAY. | The variable has a subscript or argument list but is not declared an array or function. | Self-evident. | Time-sharing mode compiler. |
| FATAL REPEAT COUNT IS NOT ALLOWED BEFORE THE FIELD DESCRIPTOR . . . | A repeat count was used with a descriptor that does not require one. | Self-evident. | Time-sharing mode compiler. |

TABLE B–2. TIME–SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL RESULTS OF CONSTANT USED WITH . . . OPERATOR INFINITE OR INDEFINITE. | The indicated operation is undefined. | Self-evident. | Time-sharing mode compiler. |
| FATAL RETURNS LIST NOT PERMITTED IN FUNCTION STATEMENT. | Only standard RETURN is permitted in FUNCTION subprogram. | Self-evident. | Time-sharing mode compiler. |
| FATAL RETURNS PARAMETER . . . MUST BE NUMERIC LABEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL RETURNS PARAMETER . . . NOT ALLOWED IN THIS STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SIGNED COUNT ALLOWED ONLY BEFORE P OR X CODE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT FUNCTION DEFINI-TION MUST OCCUR BEFORE FIRST EXECUTABLE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT FUNCTION DUMMY PARAMETER . . . NOT SIMPLE VARIABLE. | A constant or expression appears in the parameter list of a function definition. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT FUNCTION . . . REFERENCE – RIGHT PAREN MISSING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT FUNCTION . . . – MISPLACED EQUAL SIGN. | Syntax error in statement function. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT LABEL EXPECTED BUT NOT FOUND. | The statement form requires a statement label. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT LABEL ZERO IS ILLEGAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT LABEL . . . CONTAINS NON–DIGIT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL STATEMENT LABEL . . . EXCEEDS 5 DIGITS. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT LABEL . . . MUST BE NUMERIC. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL STATEMENT LABEL . . . REFERENCED BUT NOT DEFINED. | The indicated label does not appear as a statement label anywhere in the program. | Check for missing label. | Time-sharing mode compiler. |
| FATAL STRING ADDRESS MUST BE ARRAY ELEMENT OR SIMPLE VARIABLE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SUBROUTINE . . . REFERENCE AS A FUNCTION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SUBSCRIPT . . . ON . . . MUST NOT BE LOGICAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SUBSCRIPT . . . EXCEEDS 2** 17–1. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SUBSCRIPT . . . MUST BE NON-ZERO NUMERIC INTEGER CONSTANT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN ARGUMENT LIST. | Self-evident. | Check argument list. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN BLOCK NAME. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN DATA CONSTANT LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN DATA STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL SYNTAX ERROR IN DIMENSION STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN EQUIVALENCE STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN GO TO STATEMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN IMPLIED DO NESTING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN I/O IMPLIED DO. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN NAMELIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN PROGRAM CARD – SCAN STOPPED AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR IN PROGRAM UNIT NAME. | Syntax error on PROGRAM, FUNCTION, SUBROUTINE card. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX ERROR ON DIMENSION INDICATOR FOR . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL SYNTAX OF DO MUST BE I=M1, M2, M3, OR M1,M2. | Syntax error in DO statement. | Self-evident.. | Time-sharing mode compiler. |
| FATAL TABLE OVERFLOW – INCREASE FIELD LENGTH AND RERUN. | Self-evident. | Specify FL parameter on FTN card. | Time-sharing mode compiler. |
| FATAL TERMINAL DELIMITER . . . MISSING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TERMINAL RIGHT PAREN MISSING. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL THE TERMINAL STATEMENT OF DO . . . PRECEDED THE DO DEFINITION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL THIS IS NOT A FORTRAN STATEMENT. | Unrecognizable statement. | Self-evident. | Time-sharing mode compiler. |
| FATAL THIS STATEMENT MAY NOT BE A DO TERMINAL. | Following statements cannot end a DO loop: arithmetic or two-branch logical IF, GO TO, RETURN, END, STOP, PAUSE, or DO. | Self-evident. | Time-sharing mode compiler. |
| FATAL THIS STATEMENT REDEFINES A DO CONTROL INDEX. | Self-evident. | Rewrite statement or use different variable for DO index. | Time-sharing mode compiler. |
| FATAL THIS STATEMENT REQUIRES AN I/O LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TOO FEW LEFT PAREN. | Self-evident. | Self-evident. | Time-sharing mode compiler |
| FATAL TOO FEW LEFT PAREN OR PREVIOUS SYNTAX ERROR – SCAN STOPPED AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TOO FEW RIGHT PAREN. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TOO FEW RIGHT PAREN OR PREVIOUS SYNTAX ERROR – SCAN STOPPED AT . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TOO MANY LABELS IN LOGICAL IE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL TOO MANY SUBSCRIPTS ON . . . | Maximum number of subscripts is 3. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL TYPE MUST BE FOLLOWED BY A TYPE INDICATOR. | Statement is incomplete; one of REAL, INTEGER, COMPLEX, DOUBLE PRECISION, LOGICAL required. | Self-evident. | Time-sharing mode compiler. |
| FATAL UNFORMATTED I/O NOT ALLOWED IN THIS STATEMENT. | This form requires a format specification. | Self-evident. | Time-sharing mode compiler. |
| FATAL UNIT DESIGNATOR EXCEEDS 2 DIGITS. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL UNIT DESIGNATOR . . . NOT SIMPLE INTEGER VARIABLE OR CONSTANT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL UNKNOWN FORMAT CODE . . . – SCAN RESUMES AT NEXT SEPARATOR. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL UNMATCHED PARAMETER COUNT TO STATEMENT FUNCTION . . . | The function reference and function definition contain differing numbers of parameters. | Self-evident. | Time-sharing mode compiler. |
| FATAL USAGE CONFLICT . . . PREVIOUSLY DEFINED AS DO TERMINAL. | Self-evident. | Check previous loops for use of same label. | Time-sharing mode compiler. |
| FATAL USAGE CONFLICT – . . . PREVIOUSLY DEFINED AS FORMAT. | The label was previously used as a format label. | Self-evident. | Time-sharing mode compiler. |
| FATAL USAGE CONFLICT – . . . CANNOT BE STATEMENT FUNCTION. | The indicated function name conflicts with a previous usage. | Check all other usages; function name might be used as variable or array name. | Time-sharing mode compiler. |
| FATAL USAGE CONFLICT – . . . IS . . . AND CANNOT BE . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL USAGE CONFLICT – . . . PREVIOUSLY USED AS A FORMAT LABEL | The label was previously defined as a FORMAT label. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL USAGE CONFLICT – . . . PREVIOUSLY USED AS . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL VARIABLE DIMENSION ARRAY . . . MUST BE DUMMY ARGUMENT. | A variable dimension array can appear only in a subprogram, and must appear as a dummy argument. | Self-evident. | Time-sharing mode compiler. |
| FATAL VARIABLE DIMENSION INDICATOR . . . IS NOT INTEGER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL VARIABLE DIMENSION INDICATOR . . . MUST BE DUMMY ARGUMENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL VARIABLE DIMENSION NOT PERMITTED IN NAMELIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ZERO IS AN ILLEGAL UNIT NUMBER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ZERO IS SPECIFIED AS REPEAT COUNT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ZERO LENGTH SPECIFIED ON HOLLERITH CONSTANT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL 3 BRANCH IF NOT DEFINED FOR LOGICAL RESULTS. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL / NOT ALLOWED IN FORMATTED I/O OR UNFORMATTED INPUT LIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL . . . – ILLEGAL TRANSFER TO INSIDE A CLOSED DO LOOP. | To branch inside a DO loop, a branch must previously have been made out of the loop. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| FATAL ... ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN. | The EQUIVALENCE statement has extended the common block origin backwards. | Check all EQUIVALENCE statements for contradictory equivalencing. | Time-sharing mode compiler. |
| FATAL ... ILLEGAL FIRST ELEMENT OF EXPRESSION. | Self-evident. | See item filled in. | Time-sharing mode compiler. |
| FATAL ... INDEX PARAMETER IS TOO LARGE. | DO index must be $\leqslant$ 131070. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... INDEX PARAMETER MUST BE INTEGER OR OCTAL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... INDEX PARAMETER MUST BE POSITIVE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... INDEX PARAMETER BE SIMPLE VARIABLE. | Index variable cannot be constant or expression. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... IS IN BLANK COMMON – DATA IGNORED. | Blank COMMON cannot be data loaded. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... IS NOT A LEGAL TYPE. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... IS NOT IN LABELED COMMON. | The usage of the indicated variable requires that the variable be in labeled common. | Check common blocks; data cannot be loaded into blank common. | Time-sharing mode compiler. |
| FATAL ... STATEMENT MISPLACED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... SUBSCRIPT EXCEEDS 2**17-1. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| FATAL ... SUBSCRIPT LESS THAN 1 OR EXCEEDS DIMENSION. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING ASF MODELESS EXPRESSION ASSIGNED TO LOGICAL FUNCTION. | If function is type logical, the expression must be logical. | Self-evident. | Time-sharing mode compiler. |
| WARNING ASSUMED COMMA AFTER UNIT OR FORMAT – FOUND . . . | Compiler inserts the missing comma. | Self-evident. | Time-sharing mode compiler. |
| WARNING BUFFER LENGTH FOR FILE . . . EXCEEDS 360000B – 360000B USED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING COMMA AFTER DO LABEL IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING COMMA AFTER STATEMENT LABEL IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING COMMON STATEMENT WITH NO LIST IS IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING CONFLICT IN RANGE INDICATOR – FIRST RETAINED. | Self-evident. | Check for overlap of ranges in IMPLICIT statement. | Time-sharing mode compiler. |
| WARNING . . . CONSTANT TOO LONG – TRUNCATED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING DIMENSION OF . . . IGNORED; PRIOR DIMENSION RETAINED. | If a dimension is declared more than once, first declaration is assumed. | Self-evident. | Time-sharing mode compiler. |
| WARNING ENTRY INSIDE DO LOOP IS IGNORED. | An ENTRY statement cannot appear inside a DO loop. | Self-evident. | Time-sharing mode compiler. |
| WARNING ENTRY STATEMENT IGNORED IN MAIN PROGRAM. | An ENTRY statement in the main program has no purpose. | Self-evident. | Time-sharing mode compiler. |
| WARNING ENTRY . . . MUST NOT BE DECLARED EXTERNAL – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING EXPECTED E.O.S. – FOUND AND IGNORED . . . | End of statement expected. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING EXTRA CHARACTER . . . AFTER FILE NAME IGNORED. | File name in I/O statement must match file on PROGRAM statement. | Self-evident. | Time-sharing mode compiler. |
| WARNING EXTRANEOUS COMMA IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING FILE . . . NOT DEFINED . . . EQUIVALENCE IGNORED. | The file was not declared on the PROGRAM statement. filea = fileb ignored. | Self-evident. | Time-sharing mode compiler. |
| WARNING FILE . . . PREVIOUSLY DEFINED – IGNORED. | Doubly defined file on PROGRAM card. | Self-evident. | Time-sharing mode compiler. |
| WARNING FORMAT MUST HAVE STATEMENT LABEL. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING FOUND . . . AFTER FORMAT – ASSUMED RIGHT PAREN. | The indicated character appeared where a right parenthesis was expected. | Compiler assumes the missing parenthesis. | Time-sharing mode compiler. |
| WARNING FWA AND LWA NOT IN SAME ARRAY, EQUIVALENCE CLASS, OR COMMON BLOCK. | First word address and last word address must be in same COMMON block, equivalence class, or array. | Check declarative section of program for inconsistencies involving FWA and LWA. | Time-sharing mode compiler. |
| WARNING IF RESULTS IN A SIMPLE TRANSFER. | The IF can be replaced by a GO TO. | Self-evident. | Time-sharing mode compiler. |
| WARNING ILLEGAL BUFFER LENGTH FOR FILE . . . – 2003B USED. | An illegal buffer length was specified, e.g., characters, negative number. | Self-evident. | Time-sharing mode compiler. |
| WARNING ILLEGAL CHARACTER AFTER RIGHT PAREN. | Items appearing after argument list are ignored; processing continues. | Check for premature right parenthesis or misspelled RETURNS. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING ILLEGAL NAME – ENTRY STATEMENT IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING ILLEGAL RECORD LENGTH FOR FILE . . . – DEFAULT USED. | Default is 150 characters. | Self-evident. | Time-sharing mode compiler. |
| WARNING INITIAL LINE IS CONTINUATION. | Self-evident. | Check for missing line or misplaced character in column 6. | Time-sharing mode compiler. |
| WARNING I/O FILE . . . NOT DEFINED. | The indicated file has not been declared in the PROGRAM statement. | This message should be ignored for all programs residing in primary or secondary overlays. For programs that reside in the main overlay or are not part of an overlay structure, the indicated file must be declared in the PROGRAM statement. | Time-sharing mode compiler. |
| WARNING I/O LIST IGNORED WHEN USING NAMELIST. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING LAST IF RESULTS IN A NULL TRANSFER TO THIS STATEMENT. | IF acts as a do-nothing statement. | Check syntax of IF. | Time-sharing mode compiler. |
| WARNING LIMIT LESS THAN INITIAL – 1 TRIP LOOP. | Only 1 pass will be made through the loop. | Check DO statement for errors. | Time-sharing mode compiler. |
| WARNING MISSING NAME – ENTRY STATEMENT IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING MISSING SUBSCRIPTS SET TO 1 FOR EQUIVALENCE VARIABLE . . . | EQUIVALENCE variable contains fewer subscripts than declared dimension. | Self-evident. | Time-sharing mode compiler. |
| WARNING MISSPELLED KEYWORD – . . . RETURNS ASSUMED. | Item appearing after argument list is interpreted as the keyword RETURNS. | Check for premature right parenthesis or misspelled RETURNS. | Time-sharing mode compiler. |
| WARNING MULTIPLE IMPLICIT STATEMENTS NOT PERMITTED – IGNORED. | The first IMPLICIT statement is assumed. | Self-evident. | Time-sharing mode compiler. |
| WARNING MULTIPLY DEFINED LEVEL FOR NAME . . . – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING NAME . . . PREVIOUSLY DEFINED – ENTRY STATEMENT IGNORED. | Self-evident. | Check for another usage of the ENTRY name. | Time-sharing mode compiler. |
| WARNING NO PATH TO THE ENTIRE RANGE OF DO. | The statements in the loop cannot be reached. | Check for logic error; incorrect branch. | Time-sharing mode compiler. |
| WARNING NO PATH TO THIS STATEMENT. | The statement cannot be reached. | Check for logic error; missing label. | Time-sharing mode compiler. |
| WARNING NO SEQUENCE NUMBER FOUND ON FOLLOWING STATEMENT – COMMENT ASSUMED. | In SEQ mode all executable statements must contain a sequence number. | Self-evident. | Time-sharing mode compiler. |
| WARNING NON-OCTAL DIGIT IN OCTAL CONSTANT – IGNORED. | Digit must be less than or equal to 7. | Self-evident. | Time-sharing mode compiler. |
| WARNING NULL DATA STATEMENT IS IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING NULL LOADER DIRECTIVE IS IGNORED. | Self-evident. | Check for incomplete loader directive. | Time-sharing mode compiler. |
| WARNING NULL STATEMENT WITH LABEL – CONTINUE ASSUMED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING NUMBER OF ARGUMENTS IN REFERENCE TO . . . IS NOT CONSISTENT. | Number of arguments in reference must agree with number in FUNCTION or SUBROUTINE statement. | Self-evident. | Time-sharing mode compiler. |
| WARNING OBJECT OF GO TO NOT INTEGER VARIABLE. | Object of assigned GO TO must be a simple integer variable. | Self-evident. | Time-sharing mode compiler. |
| WARNING ONLY ≠ MAX. PARG ≠ FILES ARE PERMITTED, EXCESS IGNORED. | Too many files were specified on the PROGRAM statement. | Self-evident. | Time-sharing mode compiler. |
| WARNING PREMATURE E.O.S. OR EXTRA TRAILING SEPARATOR . . . | End of statement encountered or extra / or , . | Check for incomplete statement. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING PREMATURE E.O.S. – EXPECTED VARIABLE AT . . . | End of statement encountered; statement is incomplete. | Self-evident. | Time-sharing mode compiler. |
| WARNING PREVIOUS DEFINITION OF STATEMENT FUNCTION . . . IS OVERRIDDEN. | The function was defined more than once; the most recent definition is in effect. | Self-evident. | Time-sharing mode compiler. |
| WARNING RANGE INDICATOR . . . NOT 1 LETTER – TRUNCATED TO . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING RECORD LENGTH EXCEEDS 137 COLUMN – MAY EXCEED I/O DEVICE. | Self-evident. | Reduce record length. | Time-sharing mode compiler. |
| WARNING RECORD LENGTH FOR FILE . . . EXCEEDS 2** 17-1 – DEFAULT USED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING REDUNDANT EQUIVALENCE SPECIFICATION FOR . . . | Self-evident. | Check for occurrence of indicated symbol in previous EQUIVALENCE statement. | Time-sharing mode compiler. |
| WARNING RESULTS OF CONSTANT EVALUATION WILL BE OUT OF RANGE OR INDEFINITE. | This will cause arithmetic error when used in calculation. | Check for division by zero; addition, subtraction, division, or multiplication whose result is .GT.$10^{322}$ in magnitude. | Time-sharing mode compiler. |
| WARNING STATEMENT LABEL IGNORED. | Non-executable statements should not contain labels. | Self-evident. | Time-sharing mode compiler. |
| WARNING STATEMENT TRANSFERS TO ITSELF. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING T CODE RESETS COLUMN POINTER, OVERLAYING CURRENT LINE. | Self-evident. | Check for missing / or incorrectly set T specification. | Time-sharing mode compiler. |
| WARNING TERMINAL CHARACTER . . . CONVERTED TO RIGHT PAREN. | The indicated character appeared where a right parenthesis was expected. | Compiler assumes a right parenthesis. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING THIS STATEMENT REDEFINES A DO INDEX PARAMETER. | Self-evident. | Correct statement or change index parameter in DO statements. | Time-sharing mode compiler. |
| WARNING TOO FEW CONSTANTS – VARIABLES FROM . . . MAY NOT BE INITIALIZED. | Self-evident. | Initialize the variables; uninitialized variables can cause execution time errors. | Time-sharing mode compiler. |
| WARNING TRIVIAL EQUIVALENCE GROUP WITH ONLY 1 MEMBER IS IGNORED. | An equivalence must contain 2 members. | Self-evident. | Time-sharing mode compiler. |
| WARNING TRIVIAL RANGE – . . . SAME AS . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING TYPING OF . . . IGNORED – PRIOR TYPING RETAINED. | The symbol appeared in more than 1 type statement; first type assumed. | Self-evident. | Time-sharing mode compiler. |
| WARNING UNKNOWN FORM – BLANK ASSUMED. | Unrecognizable form of statement. | Self-evident. | Time-sharing mode compiler. |
| WARNING VARIABLE . . . HAS NO DIMENSION INDICATOR – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING VARIABLE . . . NOT INTEGER. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING VARIABLE . . . REFERENCED AS ARRAY. | The indicated variable was referenced with a subscript but was not dimensioned. | Self-evident. | Time-sharing mode compiler. |
| WARNING X CODE RESETS COLUMN POINTER, OVERLAYING CURRENT LINE. | Self-evident. | Check for missing / or incorrect X specification. | Time-sharing mode compiler. |
| WARNING *TO* ASSUMED FOR . . . | Syntax error in ASSIGN statement. | Self-evident. | Time-sharing mode compiler. |
| WARNING . . . CONSTANT TOO LONG .–– TRUNCATED. | The constant exceeds the length of the variable into which it is stored. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| WARNING ... CONVERSION CODE FIELD WIDTH IS LESS THAN MINIMUM REQUIRED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| WARNING ... IS NOT A LEGAL KEYWORD. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE ARGUMENT ... IS NOT USED IN FUNCTION ... | Self-evident. | Probable error in function definition. | Time-sharing mode compiler. |
| NOTE CONSTANT EXCEEDS 5 DIGITS – TRUNCATED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE CONSTANT MISSING EXPONENT FIELD – ZERO ASSUMED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE CONSTANT MULTIPLY BY ZERO – RESULTS SET TO ZERO. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE CONSTANT TERM OF ZERO – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE CONTINUE WITH NO STATEMENT LABEL – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE DIVIDE BY 1 – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE DIVIDE INTO ZERO – RESULTS SET TO ZERO. | Division by zero is an undefined operation. | Self-evident. | Time-sharing mode compiler. |
| NOTE DO CONCLUSION NOT COMPILED – DO DEFINITION ERROR. | No terminal statement of DO loop was encountered. | Self-evident. | Time-sharing mode compiler. |
| NOTE EXCESS CONSTANTS IGNORED. | Number of constants greater than number of variables. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| NOTE HOLLERITH ARGUMENT MUST NOT EXCEED 70 CHARACTERS. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE HOLLERITH CONSTANT IN EXPRESSION EXCEEDS 10 CHARACTERS. | Constant is truncated. | Self-evident. | Time-sharing mode compile. |
| IF RESULTS IN A TRANSFER TO THE NEXT LINE. | The IF statement is redundant. | Self-evident. | Time-sharing mode compiler. |
| NOTE IMPLIED LOOP IS REDUCED. | An implied DO loop has been reduced to a simple list. | Self-evident. | Time-sharing mode compiler. |
| NOTE INTEGER DIVIDE BY ZERO – RESULTS SET TO ZERO. | Division by zero is an undefined operation. | Self-evident. | Time-sharing mode compiler. |
| NOTE INTEGER ** NEGATIVE CONSTANT – RESULTS ZERO. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE LOADER DIRECTIVE MUST BE CONTAINED ON 1 CARD. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE MISSING PROGRAM STATEMENT – PROGRAM START ASSUMED. | Program is assigned name START. | Self-evident. | Time-sharing mode compiler. |
| NOTE MISSING SUBSCRIPTS ON . . . ARE ASSIGNED VALUE OF ONE. | Array reference has fewer subscripts than declared dimension. | Self-evident. | Time-sharing mode compiler. |
| NOTE MULTIPLY BY 1 – IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE NOT EVERY NAME IN COMMON BLOCK . . . IS IN A LEVEL STATEMENT. | Same level for all members assumed. | No action necessary. | Time-sharing mode compiler. |
| NOTE NULL TRANSFER STATEMENT – TRANSFER IGNORED. | A GO TO statement branches to the following statement. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| NOTE POSSIBLE ILLEGAL TRANSFER FROM OUTSIDE CURRENT DO. | A possible illegal extended range inside a DO has been encountered. | Self-evident. | Time-sharing mode compiler. |
| NOTE RETURN ACTS AS END. | A RETURN statement should not be used in a main program. | Replace with STOP. | Time-sharing mode compiler. |
| NOTE STATEMENT CAN TRANSFER TO ITSELF. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE STATEMENT FUNCTION . . . – HAS NULL DEFINITION – IGNORED. | Statement function expansion reduces to a null code sequence. | Check for error in function definition statement. | Time-sharing mode compiler. |
| NOTE STATEMENT LABEL ZERO IGNORED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE SUBSCRIPT . . . FOR . . . NOT INTEGER – TRUNCATED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| NOTE TRIVIAL DO LOOP – IGNORED. | DO loop contains no executable statements. | Self-evident. | Time-sharing mode compiler. |
| NOTE ZERO ** ZERO – RESULTS INDEFINITE. | Results in arithmetic error at execution time. | Self-evident. | Time-sharing mode compiler. |
| NOTE . . . PREVIOUSLY DEFINED AS EXTERNAL. | The indicated symbol appeared in an EXTERNAL declaration. | Check for use of subprogram name as a variable name. | Time-sharing mode compiler. |
| END LINE ABSENT. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| MISSPELLED KEYWORD – . . . . . . . ASSUMED. | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| MULTIPLE STATEMENT IGNORED AFTER END. | Self-evident. | Self-evident. | Time-sharing mode compiler. |

TABLE B-2. TIME-SHARING MODE DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| NAME EXCEEDS 7 CHARACTERS – TRUNCATED TO . . . | Self-evident. | Self-evident. | Time-sharing mode compiler. |
| TRIVIAL PROGRAM UNIT IGNORED. | A program unit has generated no executable code. | Self-evident. | Time-sharing mode compiler. |

# SPECIAL COMPILATION DIAGNOSTICS

When a compilation is aborted or prematurely terminated for internal reasons, one or more of the messages shown in table B-3 appear. This table also includes messages that appear only in the dayfile that are not caused by dayfile internal error. Unless otherwise noted, these messages appear only in optimizing mode.

TABLE B-3. SPECIAL COMPILATION DIAGNOSTICS

| Message | Significance | Action | Issued By |
|---|---|---|---|
| nnnn ASSEMBLY ERRORS IN prognam | A compiler, operating system, or hardware error has occurred while compiling n prognam. | See systems analyst. | FORTRAN Extended Compiler. |
| COMPILING prognam<br>LAST STATEMENT BEGAN AT LINE nnnnn<br>ERROR AT aaaaa IN ddddddd<br>LAST OVERLAY LOADED - (p,s) | Compiler, operating system or hardware error has occurred while compiling program (TS and OPT mode).<br>prognam — Name of source program unit.<br>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.<br>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.<br>aaaaaa — Address relative to origin of internal deck where abort occurred.<br>p,s — Primary and secondary level numbers of overlay last loaded before abort occurred:<br>0,0 – Control statement cracker; global communication and control | See systems analyst. | FORTRAN Extended Compiler. |

TABLE B-3. SPECIAL COMPILATION DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| | 1,0 - TS–mode compilation overlay<br>2,0 - Optimizing compilation batch controller<br>2,1 - Optimizing compilation normal pass 1 (lexical scan, parse, intermediate language generation)<br>2,2 - Optimizing compilation pass 2 (global and local optimization, object code generation)<br>2,3 - Optimizing compilation diagnostic phase (occurs between pass 1 and pass 2)<br>2,4 - Optimizing compilation C$ DEBUG pass 1<br>2,5 - Optimizing compilation reference map generation and object code assembly phase | | FORTRAN Extended Compiler. |
| DEAD CODE IN program | A section of code is unreachable and cannot be processed. | Same as STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED. | FORTRAN Extended Compiler. |
| ECS READ ERROR<br>ECS WRITE ERROR | ECS/LCM read or write Parity error. Can occur only under OPT=2. | See systems analyst. | FORTRAN Extended Compiler. |
| FTN/FBV - BLOCK READ ERROR | Compiler, operating system, or hardware error. Most probable cause is a disk or READNS (read non-stop) input/output command error. Can occur only under OPT=2. | See systems analyst. | FORTRAN Extended Compiler. |

TABLE B-3. SPECIAL COMPILATION DIAGNOSTICS (CONT'D)

| Message | Significance | Action | Issued By |
|---|---|---|---|
| NULL PROGRAM-HONOR THE A PARAMETER | The A parameter appears on the control statement, and the presence of a null program causes the compiler to instruct the operating system to skip to the next EXIT(S), if any. | Eliminate the null program or eliminate the A parameter or both. | Optimizing Mode Compiler. |
| NULL PROGRAM IGNORED AFTER program. | A program unit (other than a BLOCK DATA subprogram) does not contain any executable statements; it is ignored. Compiler, operating system, or hardware error. | Self-evident. | FORTRAN Extended Compiler. |
| OBJECT CODE END LINE MISSING | Compiler, operating system, or hardware error. | See systems analyst. | FORTRAN Extended Compiler. |
| ** PASS 2 MEMORY OVERFLOW ** | ———————— | Same as similar message in Compiler Output Listing Messages below. | FORTRAN Extended Compiler. |
| ** PREMATURE EOF ON –REFMAP– FILE. | Compiler, operating system, or hardware error. Can occur only when R=2 or 3 is selected. | See systems analyst. | FORTRAN Extended Compiler. |
| * PROGRAM CONTAINS SEQUENCES THAT ARE TOO LONG. CANNOT BE COMPILED. | The program contains one or more long sequences of statements that are not broken up by a statement label definition, an IF statement, or a GO TO statement. | Compile the program at a lower mode of optimization or modify the program so as to break up the long code sequence. | Optimizing Mode Compiler. |
| EMPTY INPUT FILE. NO COMPILATION. | An end–of–partition or end–of–section was encountered on the first read of the input (TS mode only). | Self-evident. | FORTRAN Extended Compiler. |
| CM REQUIRED FOR LOAD EXCEEDS 131K. | Address exceeds 131K. | Reduce size of program. | FORTRAN Extended Compiler. |

## COMPILER OUTPUT LISTING MESSAGES

The error messages can appear in the body of the compilation listing in optimizing mode only. If present, they are located (listed in table B-4) after the source program and standard error summary listings. They may appear before, during or after the reference map and object code listings, depending on the exact error condition. The message format is different from that of the standard error summary; each message is usually left-justified on the output listing page, and may be preceded by several blank lines or a page eject.

An example of dead code, which would produce the last diagnostic in table B-4 is as follows:

```
      A=2.
      GO TO 30
C        THE NEXT STATEMENT CANNOT BE EXECUTED.
      A=A+1.
   30 STOP
      END
```

A more subtle example is:

```
      A=2.
      ASSIGN 40 TO J
      ASSIGN 50 TO J
      ASSIGN 60 TO J
      GO TO J, (40,50)
C        THE NEXT STATEMENT CANNOT BE EXECUTED, BECAUSE
C        ITS LABEL DOES NOT APPEAR IN THE GO-TO TRANSFER
C        LIST.
   60 A=A+1.
   40 STOP
   50 STOP
      END
```

# EXECUTION DIAGNOSTICS

Execution diagnostics are the same whether the source program was compiled in optimizing mode or time-sharing mode. Execution diagnostics are printed on the source listing in the following format:

ERROR NUMBER x DETECTED BY routine AT ADDRESS y

or

ERROR NUMBER x DETECTED BY routine

followed by

CALLED FROM routine AT ADDRESS z

or

CALLED FROM routine AT LINE d

y and z are octal addresses, x is a decimal error number, and d is a decimal line number as printed on the source listing.

TABLE B-4. COMPILER OUTPUT LISTING MESSAGES

| Message | Significance | Action | Issued By |
|---|---|---|---|
| CANT SORT SYMBOL TABLE INCREASE FL BY ffffB. | Not enough CM/SCM field length available to generate a reference map. ffff is an estimate of additional FL necessary for generating the map. Cannot occur if R=0 is specified. | Increase field length before recompilation. | FORTRAN Extended Compiler. |
| *** MEMORY OVERFLOW IN –FAX– | Not enough CM/SCM field length for final assembly of binary object code. Additional memory required varies; 10K to 20K increments are suggested. | Increase field length before recompilation. | FORTRAN Extended Compiler. |
| PASS 2 MEMORY OVERFLOW AT SOURCE LINE nnnn IN compnam | Could be genuine memory overflow, or a compiler error. If compnam is JAM–ERR, a compiler error has occurred. Otherwise, not enough CM/SCM field length was available for pass 2 of an optimizing compilation. | Increase field length or decrease optimization level. | FORTRAN Extended Compiler. |
| REFERENCES AFTER LINE nnnn LOST INCREASE FL BY ffffB | Not enough CM/SCM field length available to generate a long reference map, nnnnn is the approximate compiler-assigned source line number where the difficulty arose. ffff is a rough estimate of the additional field length needed for generating the complete map. This error can occur only when R=2 or 3 is selected. | Increase field length before recompilation. | FORTRAN Extended Compiler. |
| STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UN-REACHABLE (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 prognam. Detected only when OPT=2 has been selected. | Self-evident. | FORTRAN Extended Compiler. |
| LCM FL EXCEEDS 131071 WORDS (LCM=I REQUIRED). | LCM=I must be specified if the execution LCM field length exceeds 131071 words. | Specify LCM=I on compiler call. | FORTRAN Extended Compiler. |

Example:

```
                    PROGRAM EXERR        74/74    OPT=1


        1                       PROGRAM EXERR(INPUT,OUTPUT)
                                N=5
                                GO TO (1,2,3),N
                            1   N=N+1
        5                   2   N=N+2
                            3   STOP
                                END


   CARD NR. SEVERITY  DETAILS    DIAGNOSIS OF PROBLEM

        3     I                     AN IF STATEMENT MAY BE MORE EFFICIENT
                                    THAN A 2 OR 3 BRANCH COMPUTED GO TO
                                    STATEMENT.
   ERROR IN COMPUTED GOTO STATEMENT- INDEX VALUE INVALID

   ERROR NUMBER    1    DETECTED BY GOTOFR= AT ADDRESS 000004
   CALLED FROM EXERR    AT  LINE 3
```

In the list of execution diagnostics shown in table B-5, the letters under class are interpreted as follows:

| | |
|---|---|
| F = Fatal | D = Debug (diagnostic issued only in debug mode) |
| I = Informative, non-fatal | T = Trace (diagnostic issued only in trace mode) |
| | A = Always (diagnostic always issued) |

The severity level (fatal or non-fatal) of any error except for erroneous data input from a connected file can be changed by a call to SYSTEMC (section 8).

In the messages, X and Y are real numbers, D is a double precision number, I is an integer, and Z is a complex number.

NOTE

For some execution time errors, only a dayfile message of FTN — FATAL ERRORn is issued; n contains a meaningless value such as zero. This type of error usually indicates an erroneous branch into the FORTRAN library routines. For example, a missing END card on an intermixed COMPASS subprogram could cause this type of error.

TABLE B-5. EXECUTION 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 | Self-evident. | GOTOER= |
| 2 | I A | ARGUMENT ABS VALUE .GT. 1<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | ACOSIN=(ACOS) |
| 3 | I A | ARGUMENT ZERO<br>ARGUMENT NEGATIVE<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | ALOG |
| 4 | I A | ARGUMENT ZERO<br>ARGUMENT NEGATIVE<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | ALOG10 |
| 5 | I A | ARGUMENT ABS VALUE .GT. 1<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | ACOSIN=(ASIN) |
| 6 | I A | ARGUMENT INDEFINITE | †† | † | ATAN |
| 7 | I A | ARGUMENT VECTOR ZERO<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | ATAN2 |
| 8 | I A | ARGUMENT TOO LARGE<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | CABS |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 9 | I T | ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | ZTOI (ZX*I) |
| 10 | I T | INFINITE ARGUMENT<br>INDEFINITE ARGUMENT<br>ABS (REAL PART) TOO LARGE<br>ABS (IMAG PART) TOO LARGE | †† | † | CCOS |
| 11 | I T | INFINITE ARGUMENT<br>INDEFINITE ARGUMENT<br>ARGUMENT (REAL) OUT OF RANGE<br>ARGUMENT (IMAG) OUT OF RANGE | †† | † | CEXP |
| 12 | I T | ZERO ARGUMENT<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | CLOG |
| 13 | I A | ARGUMENT TOO LARGE, ACCURACY LOST<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | SINCOS=(COS) |
| 14 | I T | INFINITE ARGUMENT<br>INDEFINITE ARGUMENT<br>ABS (REAL PART) TOO LARGE<br>ABS (IMAG PART) TOO LARGE | †† | † | CSIN |
| 15 | I T | INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | CSQRT |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 16 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE BASE IN EXPONENTIATION<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DTOX (D**X) |
| 17 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | DATAN |
| 18 | I A | ARGUMENT VECTOR 0,0<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | DATAN2 |
| 19 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE TO THE DOUBLE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DTOD (D**D) |
| 20 | I T | ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DTOI (D**I) |
| 21 | I T | FLOATING OVERFLOW IN D** REAL(Z)<br>ZERO TO THE ZERO OR NEGATIVE POWER<br>NEGATIVE TO THE COMPLEX POWER<br>IMAG(Z)*LOG(D) TOO LARGE<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DTOZ (D**Z) |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 22 | I T | ARGUMENT TOO LARGE, ACCURACY LOST<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DCOS |
| 23 | I A | ARGUMENT TOO LARGE<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | DEXP |
| 24 | I T | ZERO ARGUMENT<br>NEGATIVE ARGUMENT<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DLOG |
| 25 | I T | ZERO ARGUMENT<br>NEGATIVE ARGUMENT<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DLOG10 |
| 26 | I T | DP INTEGER EXCEEDS 96 BITS<br>2ND ARGUMENT ZERO<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DMOD |
| 28 | I T | ARGUMENT TOO LARGE, ACCURACY LOST<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DSIN |
| 29 | I T | NEGATIVE ARGUMENT<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | DSQRT |
| 30 | I A | ARGUMENT TOO LARGE, FLOATING OVERFLOW<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | Self-evident. | † | EXP |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 31 | I T | INTEGER OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER | Self-evident. | † | ITOJ (I**J) |
| 33 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE TO THE DOUBLE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | XTOD (X**D) |
| 34 | I A | ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT<br>OVERFLOW ARGUMENT<br>UNDERFLOW ARGUMENT | †† | † | XTOI (X**I) |
| 35 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE TO THE REAL POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | XTOY (X**Y) |
| 36 | I A | ARGUMENT TOO LARGE, ACCURACY LOST<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | Self-evident. | † | SINCOS=(SIN) |
| 37 | I T | ILLEGAL SENSE LITE NUMBER | Number not in range 1-6; lights not changed. | Self-evident. | SLITE |
| 38 | I T | ILLEGAL SENSE LITE NUMBER | Number not in range 1-6; lights not changed. | Self-evident. | SLITET |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 39 | I A | ARGUMENT NEGATIVE<br>ARGUMENT INFINITE<br>ARGUMENT INDEFINITE | †† | † | SQRT |
| 40 | I T | ILLEGAL SENSE SWITCH NUMBER | Number not in range 1–6; return parameter set to 2. | Self-evident. | SSWTCH |
| 41 | I A | ARGUMENT TOO LARGE<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | TAN |
| 42 | I T | INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | TANH |
| 44 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE TO THE DOUBLE POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | ITOD (I**D) |
| 45 | I T | FLOATING OVERFLOW<br>ZERO TO THE ZERO POWER<br>ZERO TO THE NEGATIVE POWER<br>NEGATIVE TO THE REAL POWER<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | ITOX (I**X) |
| 46 | I T | FLOATING OVERFLOW IN I** REAL(Z)<br>ZERO TO THE ZERO OR NEGATIVE POWER<br>NEGATIVE TO THE COMPLEX POWER<br>IMAG(Z)*LOG(I) TOO LARGE<br>INFINITE ARGUMENT<br>INDEFINITE ARGUMENT | †† | † | ITOZ (I**Z) |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 47 | I T | FLOATING OVERFLOW IN X** REAL(Z)<br>ZERO TO THE ZERO OR NEGATIVE POWER<br>NEGATIVE TO THE COMPLEX POWER<br>IMAG(Z)*LOG(X) TOO LARGE<br>INFINITE OR INDEF ARGUMENT | †† | † | XTOZ(X**Z) |
| 48 | F D | FATAL ERROR ENCOUNTERED DURING PROGRAM EXECUTION DUE TO COMPILATION ERROR | Self-evident. | Correct the compilation error and recompile. | FTNERR= |
| 49 | I A<br>I A<br>I A | COMMA MISSING AT END OF RECORD – COMMA ASSUMED<br>NAMELIST DATA TERMINATED BY EOF NOT $<br>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. | Self-evident. | |
| 52 | F A | Set by user via subroutine SYSTEM or SYSTEMC. Error numbers larger than those listed in this table become error 52. | Defined by user. | Self-evident. | |
| 53 | F A | NOT ENOUGH FL FOR SORT/MERGE. | More memory required for SORT/MERGE processing. | Extend program field length. | SMXXX= |
| 54 | F A | MIXED IO MODES. nnnnnn AFTER oooooo ON LFN-xxxxxxx where nnnnnn and oooooo may be:<br>1)CODED<br>2)BINARY<br>3)BUFFER | User is trying to switch I/O mode on a file without an intervening REWIND or ENDFILE. | Continue using previous I/O mode or perform a REWIND or ENDFILE. | FORSYS= |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by zero, or by an addition, subtraction, multiplication or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 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. | Self-evident. | BUFIN= |
| 57 | F A | AREA SPECIFICATION SPANS SCM/LCM. | In a buffered I/O statement the first and last word addresses must be on the same level. | Self-evident. | BIFIO= |
| 58 | F A | BUFFER DESIGNATION BAD - - FWA.GT.LWA. | First-word address must be LE last word address. | Self-evident. | BUFIO= |
| 59 | F A | BUFFER SPECIFICATION BAD - - - FWA.GT.LWA. | First-word address must be LE last word address. | Self-evident. | BUFOUT |
| 60 | F A | BFS EXCEEDS ALLOCATED STATIC SIZE, LFN-xxxxxxx. | User supplied FILE card sets BFS larger than FTN PROGRAM statement declaration in STATIC option run. | Omit BFS specification from FILE card. | FORSYS= |
| 62 | F A | FILENAME NOT DECLARED-xxxxxxx. | Filename must be declared in PROGRAM statement. | Self-evident. | GETFIT= |
| 63 | F A | END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx. | Attempt to read past end-of-file. | Rewind file or correct program logic. | INPB= |
| 65 | F A | END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx. | Attempt to read past end-of-file. | Rewind file or correct program logic. | INPC= NAMIN= |

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 66 | F A | NAMELIST NAME NOT FOUND-xxxxxxx. | Error occurred during NAMELIST processing. | Check NAME-LIST input data for errors. | NAMIN= |
| | F A | INCORRECT SUBSCRIPT. | | | |
| | F A | TOO MANY CONSTANTS. | | | |
| | F A | , ( $ OR = EXPECTED, MISSING. | | | |
| | F A | VARIABLE NAME NOT FOUND-xxxxxxx. | | | |
| | F A | CONSTANT MISSING. | | | |
| 67 | F A | DECODE RECORD LENGTH .LE. 0.<br>DECODE LCM RECORD .GT. 150 CHARACTERS. | Bad first parameter to DECODE. | Self-evident. | DECODE= |
| 68 | F A | * ILL-PLACED NUMBER OR SIGN. | Illegal FORMAT. | Self-evident. | |
| | F A | * ILLEGAL FUNCTIONAL LETTER. | Illegal FORMAT. | Self-evident. | FMTAP= |
| 69 | F A | * IMPROPER PARENTHESIS NESTING. | Illegal FORMAT. | Self-evident. | FMTAP= |
| 70 | F A | * EXCEEDED RECORD SIZE. | The maximum record length specified on the PROGRAM statement or on the FILE control statement has been exceeded. | Change rl parameter on PROGRAM statement or MRL parameter on FILE control statement, whichever is appropriate.† | FMTAP= |
| 71 | F A | * SPECIFIED FIELD WIDTH ZERO. | w=0 in FORMAT. | Self-evident. | FMTAP= |
| | | * BAD VALUE FOR = OR V. | Self-evident. | Self-evident. | |
| 72 | F A | * FIELD WIDTH .LE. DECIMAL WIDTH. | $w \leqslant d$ in FORMAT. | Self-evident. | 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= |

† If the maximum record length (MRL) is specified on the PROGRAM statement and on the FILE control statement, the minimum value if used; hence, the FILE control statement can decrease but not increase the maximum record length.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 78 | F A | * ILLEGAL DATA IN FIELD . ↑ . | Usually a non-digit 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 FTN statement or change program to print less. | OUTC= NAMOUT= |
| 85 | F A | ENCODE CHARACTER/RECORD .LE. 0.<br>ENCODE LCM RECORD .GT. 150 CHARACTERS. | Bad first parameter to ENCODE. | Self-evident. | ENCODE= |
| 88 | F A | WRITE FOLLOWED BY READ ON FILE-xxxxxxx. | A READ cannot follow a WRITE unless a REWIND intervenes. | Self-evident. | INPB= |
| 89 | F A | LIST EXCEEDS DATA, FILENAME-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 READING (BINARY) FILE-xxxxxxx. | 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. | Self-evident. | INPC= |
| 92 | F A | PARITY ERROR READING (CODED) FILE-xxxxxxx. | Probable disk or tape error. | See systems analyst. | INPC= NAMIN= |

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 93 | F A | PARITY ERROR ON LAST READ ON FILE-xxxxxxx. | 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= |
| 97 | F A | INDEX NUMBER ERROR. | Non-existent index value specified or bad file. | Self-evident. | RANMS= |
| 98 | F A | FILE ORGANIZATION ERR OR FILE NOT OPEN. | Self-evident. | Call OPENMS. | RANMS= |
| 99 | F A | WRONG INDEX TYPE. | Wrong type specified to OPENMS. | Self-evident. | RANMS= |
| 100 | F A | INDEX IS FULL. | Self-evident. | Increase index size. | RANMS= |
| 101 | F A | DEFECTIVE INDEX CONTROL WORD. | Program overwriting record index or bad file. | Fix program or recreate file. | RANMS= |
| 102 | F A | RECORD LENGTH EXCEEDS SPACE ALLOCATED. | Self-evident. | Increase space allocation. | RANMS= BUFIO= |
| 103 | F A | 6RM/7DM I/O ERR NUMBER xxx. | Record Manager error. | See Record Manager Reference Manual. | RANMS= |
| 104 | F A | INDEX KEY UNKNOWN. | Self-evident. | Self-evident. | RANMS= |
| 105 | F A | RECORD LENGTH NEGATIVE. | Self-evident. | Fix call. | RANMS= |
| 107 | F A | ILLEGAL PARAMETER VALUE. | Argument to SORT/MERGE routine has bad value. | Self-evident. | SMXXXX= |
| 108 | F A | TOO FEW OR TOO MANY PARAMETERS. | Self-evident. | Self-evident. | SMXXX= |

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 109 | F A | KEYWORD (xxxxxxxx) INVALID. | Self-evident. | Self-evident. | SMXXX= |
| 110 | F A | A ROUTINE CALLED OUT OF SEQUENCE. | Sequence (SMSORT, SMSORTB, SMSORTP, or SMMERGE), (other SORT/MERGE calls), (SMEND or SMABT) not followed. | Self-evident. | SMXXX= |
| 111 | F A | LCM BLOCK COPY ERROR. | Parity error. | See systems analyst. | COMIO=, DECODE=, ENCODE=, INPB=, OUTB=, READEC, WRITEC |
| 112 | F A | ECS UNIT HAS LOST POWER OR IS IN MAINTENANCE MODE. | Hardware error. | See systems analyst. | WRITEC |
| 113 | F A | ECS READ PARITY ERROR | Possible hardware error. | See systems analyst. | READEC |
| 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<br>ARGUMENT TOO SMALL, RESULT UNDERFLOW | ††<br>Self-evident. | † | EXP |
| 116 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | HYP=(COSH) |
| 117 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | HYP=(SINH) |
| 118 | I A | ARGUMENT TOO SMALL, RESULT UNDERFLOW | Self-evident. | † | DEXP |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by a zero, or by an addition, subtraction, multiplication, or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 119 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | DHYP=(DCOSH) |
| 120 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | DHYP=(DSINH) |
| 121 | I A | ARGUMENT INDEFINITE | †† | † | DTANH |
| 122 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | DTAN |
| 123 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT .GT. 1.0. | †† | † | DASNCS(DASIN) |
| 124 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT .GT. 1.0. | †† | † | DASNCS(DACOS) |
| 125 | I A | ARGUMENT INDEFINITE | †† | † | ERF(ERF) |
| 126 | I A | ARGUMENT INDEFINITE | †† | † | ERF(ERFC) |
| 127 | I A | ARGUMENT TOO LARGE | †† | † | ERF(ERFC) |
| 128 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT .GE. 1.0. | †† | † | ATANH |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by a zero, or by an addition, subtraction, multiplication, or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

TABLE B-5. EXECUTION DIAGNOSTICS (CONT'D)

| No. | Class | Message | Significance | Action | Issued By |
|---|---|---|---|---|---|
| 129 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | SIND |
| 130 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE | †† | † | COSD |
| 131 | I A | ARGUMENT INFINITE<br>ARGUMENT INDEFINITE<br>ARGUMENT TOO LARGE<br>ARGUMENT ODD MULTIPLE OF 90. | †† | † | TAND |
| 132-164 | | Reserved for FTN5 | | | |
| 165 | F A | INVALID SEQUENCE | SMKEY call specified a col-seq parameter without specifying a coding identifier of DISPLAY. SMKEY call specified a invalid col-seq parameter or an invalid code identifier. | 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 |
| 167-217 | | Reserved for FTN5 | | | |

† Check for undefined argument; if argument is calculated, check for undefined or illegal operand.

†† Infinites can be generated by dividing a non-zero number by a zero, or by an addition, subtraction, multiplication, or division whose result was greater than $10^{322}$ in absolute value. Indefinites are usually generated by dividing zero by zero.

## REPRIEVE PROCESSOR DIAGNOSTICS

Execution time diagnostics are issued by the FORTRAN reprieve processor. Object time reprieve occurs in TS and OPT=0 modes unless ER=0 has been specified on the FTN control statement, and in OPT=1 and OPT=2 modes if ER has been specified on the FTN control statement. The error message is written to the job dayfile and, under NOS 1 only, to file OUTPUT. Execution terminates following object time reprieve. The messages are listed in table B-6.

TABLE B-6. OBJECT TIME REPRIEVE DIAGNOSTICS

| Message | Significance | Action |
|---|---|---|
| TIME LIMIT EXCEEDED | Program execution time has exceeded the allowed time limit. | Check for infinite loop; use OPT=1 or 2 compilation; increase time limit with ETL statement (NOS/BE 1), SETTL statement (NOS 1), or on job statement. |
| MODE=nn | An invalid arithmetic computation has occurred at or near the indicated line. | Refer to appendix D. |
| MEMORY PARITY | A hardware error has occurred. | Rerun program. If error recurs, consult systems analyst. |
| INDEFINITE VALUE | An illegal arithmetic operation was performed at or near the indicated line. | Check for an uninitialized variable or a O/O division. |
| INFINITE VALUE | An illegal arithmetic operation was performed at or near the indicated line. | Check for division by zero or an arithmetic operation whose result was greater than $10^{322}$ or less than $10^{-294}$ in absolute value. |
| BAD ADDRESS | The program referenced an inaccessible area of memory. | Check for any situation that has caused valid addresses to be overwritten, such as DO loop that has exceeded array boundaries. Check for use of initialized variables. Check for I/O statements that exceed buffer size. |
| PP ABT | A peripheral processor (system function call) has aborted during reprieve processing. | Consult systems analyst. |
| ERROR CODE nnn | An arithmetic mode error has occurred at or near the indicated line. | Refer to appendix D. |
| BAD PP | A bad peripheral processor call was made. | Consult systems analyst. |

TABLE B-6. OBJECT TIME REPRIEVE DIAGNOSTICS (CONT'D)

| Message | Significance | Action |
|---|---|---|
| DROPPED | The job has been terminated by the operator. | Consult operator or dayfile for message from operator. |
| KILL | The job has been terminated by the operator. | Consult operator or dayfile for message from operator. |
| RERUN | The operator has rerun the job. | Consult operator or dayfile for message from operator. |
| ECS PAR | A hardware error has occurred while accessing ECS. | Rerun program. If error recurs, consult systems analyst. |
| AUTORCL | Automatic recall error. | Consult systems analyst. |
| MS LIMIT EXCEEDED | A mass storage output operation has exceeded available mass storage. | Decrease the amount of data being written to mass storage. |
| NULL PP | Program attempted to call a nonexistent peripheral processor (system function). | Consult systems analyst. |
| I/O LIMIT EXCEEDED | I/O operations exceeded limits established by the site. | Check for runaway I/O or consult systems analyst. |
| LCM LIMIT EXCEEDED | A write operation (WX or WL) has exceeded available LCM. | Decrease the amount of data being written to LCM. |
| PARITY ERROR | Probable hardware error. | Consult systems analyst. |
| OVER-INDEXED ARRAY | Program referenced a location outside array bounds. | Use C$ DEBUG, CYBER Interactive Debug, or Post Mortem Dump to locate error. |
| UNSATISFIED EXT | Unsatisfied external reference. A call was made to a non-existent function at subroutine. | Check for missing subprogram or misspelled sub-program name. See load map or dayfile. |
| BAD RESULT | Program attempted an illegal mathematical operation. | Use C$ DEBUG, CYBER Interactive Debug, or Post Mortem Dump to locate the error. |

# STATEMENT FORMS C

The following symbols are used in the descriptions of FORTRAN Extended statements:

| | |
|---|---|
| v | variable or array element |
| sn | statement label |
| iv | integer variable |
| m | unsigned integer or octal constant or integer variable |
| name | symbolic name |
| u | input/output unit:<br>1- or 2-digit decimal integer constant, integer variable with value of: 0-99, or an integer variable containing a Hollerith value which is the filename in L format |
| fn | format designator |
| iolist | input/output list |

Other forms are defined individually in the following list of statements.

| ASSIGNMENT STATEMENTS | Page Numbers |
|---|---|
| v = arithmetic expression | 2-15 |
| logical v = logical or relational expression | 2-18 |
| v = masking expression | 2-19 |

## MULTIPLE ASSIGNMENT

| | |
|---|---|
| $v_1 = v_2 = \ldots v_n$ = expression | 2-19 |

## FLOW CONTROL STATEMENTS

| | |
|---|---|
| GO TO sn | 4-1 |
| GO TO ($sn_1, \ldots, sn_m$),iv | 4-2 |
| GO TO ($sn_1, \ldots, sn_m$)iv | 4-2 |
| GO TO ($sn_1, \ldots, sn_m$),expression | 4-2 |
| GO TO ($sn_1, \ldots, sn_m$)expression | 4-2 |

| | Page Numbers |
|---|---|
| GO TO iv, $(sn_1, \ldots, sn_m)$ | 4-4 |
| GO TO iv $(sn_1, \ldots, sn_m)$ | 4-4 |
| ASSIGN sn TO iv | 4-3 |
| IF (arithmetic or masking expression) $sn_1, sn_2, sn_3$ | 4-5 |
| IF (arithmetic or masking expression) $sn_1, sn_2$ | 4-5 |
| IF (logical or relational expression) stat | 4-6 |
| IF (logical or relational expression) $sn_1, sn_2$ | 4-7 |
| DO sn iv = $m_1, m_2, m_3$ | 4-7 |
| DO sn iv = $m_1, m_2$ | 4-7 |
| CONTINUE | 4-12 |
| PAUSE | 4-13 |
| PAUSE n | 4-13 |
| PAUSE ≠c...c≠ | 4-13 |
| STOP | 4-14 |
| STOP n | 4-14 |
| STOP ≠c...c≠ | 4-14 |
| END | 4-14 |

## TYPE DECLARATION

| | |
|---|---|
| INTEGER $name_1, \ldots, name_n$ | 3-1 |
| TYPE INTEGER $name_1, \ldots, name_n$ | 3-1 |

| | Page Numbers |
|---|---|
| REAL $name_1, \ldots, name_n$ | 3-2 |
| TYPE REAL $name_1, \ldots, name_n$ | 3-2 |
| COMPLEX $name_1, \ldots, name_n$ | 3-2 |
| TYPE COMPLEX $name_1, \ldots, name_n$ | 3-2 |
| DOUBLE PRECISION $name_1, \ldots, name_n$ | 3-2 |
| DOUBLE $name_1, \ldots, name_n$ | 3-2 |
| TYPE DOUBLE PRECISION $name_1, \ldots, name_n$ | 3-2 |
| TYPE DOUBLE $name_1, \ldots, name_n$ | 3-2 |
| LOGICAL $name_1, \ldots, name_n$ | 3-3 |
| TYPE LOGICAL $name_1, \ldots, name_n$ | 3-3 |
| IMPLICIT $type_1 (ac_1, \ldots, ac_m), \ldots, type_n (ac_1, \ldots, ac_m)$ | 3-3 |

ac is one or more single alphabetic characters or ranges of characters separated by commas. A range is represented by the first and last character separated by a minus sign.

## EXTERNAL DECLARATION

| | |
|---|---|
| EXTERNAL $name_1, \ldots, name_n$ | 3-14 |

## STORAGE ALLOCATION

| | |
|---|---|
| type $name_1 \; (d_i)$ | 3-1 |
| TYPE type $name_1 \; (d_i)$ | 3-1 |
| DIMENSION $name_1 \; (d_1), \ldots, name_n \; (d_n)$ | 3-4 |

| | |
|---|---|
| $d_i$ | array declarator, one to three integer constants; or if name is a dummy argument in a subprogram, one to three integer variables or constants |
| type | INTEGER, REAL, COMPLEX, DOUBLE, DOUBLE PRECISION or LOGICAL |

| | Page Numbers |
|---|---|
| COMMON $v_1, \ldots, v_n$ | 3-5 |
| COMMON/$blkname_1/v_1, \ldots, v_n \ldots /blkname_n/v_1, \ldots, v_n$ | 3-5 |
| COMMON// $v_1, \ldots, v_n$ | 3-5 |

blkname — symbolic name or 1 - 7 digits

// — blank common

| | |
|---|---|
| DATA $vlist_1/dlist_1/, \ldots, vlist_n/dlist_n/$ | 3-15 |
| DATA $(vlist_1=dlist_1), \ldots, (vlist_n=dlist_n)$ | 3-15 |

$vlist_i$ — list of array names, array elements, variable names, and implied DO loops, separated by commas

$dlist_i$ — one or more of the following forms separated by commas:

constant
(constant list)
rf*constant
rf*(constant list)
rf(constant list)

constant list — list of constants separated by commas

rf — integer constant. The constant or constant list is repeated the number of times indicated by rf

| | |
|---|---|
| EQUIVALENCE $(glist_1), \ldots, (glist_n)$ | 3-8 |
| LEVEL n, $a_1, \ldots, a_n$ | 3-12 |

n — unsigned integer 1, 2 or 3

$a_i$ — variable, array element, array name

| | Page Numbers |
|---|---|
| **MAIN PROGRAMS** | |
| PROGRAM name | 7-2 |
| PROGRAM name($fpar_1$, $fpar_2$, . . . , $fpar_k$) | 7-2 |
| **SUBPROGRAMS** | |
| FUNCTION name ($p_1$, . . . , $p_n$) | 7-8 |
| type FUNCTION name ($p_1$ , . . . , $p_n$) | 7-8 |
| type — INTEGER, REAL, COMPLEX, DOUBLE, DOUBLE PRECISION or LOGICAL | |
| SUBROUTINE name ($p_1$,. . . , $p_n$) | 7-6 |
| SUBROUTINE name | 7-6 |
| SUBROUTINE name ($p_1$,. . . , $p_n$), RETURNS ($b_1$,. . . ,$b_m$) | 7-6 |
| SUBROUTINE name,RETURNS ($b_1$,. . . ,$b_m$) | 7-6 |
| **ENTRY POINT** | |
| ENTRY name | 7-18 |
| **STATEMENT FUNCTIONS** | |
| name ($p_1$,. . . , $p_n$) = expression | 7-10 |
| **SUBPROGRAM CONTROL STATEMENTS** | |
| CALL name | 7-16 |
| CALL name ($p_1$,. . . , $p_n$) | 7-16 |
| CALL name ($p_1$,. . . , $p_n$),RETURNS ($b_1$,. . . ,$b_m$) | 7-16 |
| CALL name,RETURNS ($b_1$,. . . ,$b_m$) | 7-16 |
| RETURN | 4-15 |
| RETURN i | 4-15 |
| i — is a dummy argument in a RETURNS list | |

| | Page Numbers |
|---|---|
| **SPECIFICATION SUBPROGRAMS** | |
| BLOCK DATA | 7-5 |
| BLOCK DATA name | 7-5 |
| **INPUT/OUTPUT** | |
| PRINT fn,iolist | 5-2 |
| PRINT fn | 5-2 |
| PRINT (u,fn) iolist | 5-3 |
| PRINT*,iolist | 5-10 |
| PRINT (u,fn) | 5-3 |
| PRINT (u,*) iolist | 5-10 |
| PUNCH fn,iolist | 5-3 |
| PUNCH fn | 5-3 |
| PUNCH (u,fn) iolist | 5-4 |
| PUNCH*,iolist | 5-10 |
| PUNCH (u,fn) | 5-4 |
| PUNCH (u,*) iolist | 5-10 |
| WRITE (u,fn) iolist | 5-4 |
| WRITE (u,fn) | 5-4 |
| WRITE fn,iolist | 5-5 |
| WRITE fn | 5-5 |
| WRITE (u) iolist | 5-7 |
| WRITE (u) | 5-7 |
| WRITE (u,*) iolist | 5-10 |
| WRITE*,iolist | 5-10 |

| | Page Numbers |
|---|---|
| READ (u,fn)iolist | 5-5 |
| READ (u,fn) | 5-5 |
| READ fn,iolist | 5-5 |
| READ fn | 5-6 |
| READ (u) iolist | 5-7 |
| READ (u) | 5-7 |
| READ (u,*) iolist | 5-8 |
| READ*,iolist | 5-8 |
| BUFFER IN (u,p) (a,b) | 5-19 |
| BUFFER OUT (u,p) (a,b) | 5-20 |

| | |
|---|---|
| a | first word of data block to be transferred |
| b | last word of data block to be transferred |
| p | integer constant or integer variable.<br>zero = even parity, nonzero = odd parity |

| | |
|---|---|
| NAMELIST/group name$_1$/$a_1$,...,$a_n$/.../group name$_n$/$a_1$,...,$a_n$ | 5-11 |
| READ (u,group name) | 5-13 |
| READ group name | 5-13 |
| WRITE (u,group name) | 5-14 |
| WRITE group name | 5-15 |
| PRINT (u, group name) | 5-15 |
| PRINT group name | 5-15 |
| PUNCH (u, group name) | 5-15 |
| PUNCH group name | 5-15 |

| | |
|---|---|
| $a_i$ | array names or variables |
| group name | symbolic name identifying the group $a_1, \ldots, a_n$ |

| | Page Numbers |
|---|---|
| **INTERNAL TRANSFER OF DATA** | |
| ENCODE (c,fn,v) iolist | 5-21 |
| DECODE (c,fn,v) iolist | 5-24 |
| v — starting location of record. Variable or array name | |
| c — length of record in characters. Unsigned integer constant or simple integer variable | |
| **FILE MANIPULATION** | |
| REWIND u | 5-26 |
| BACKSPACE u | 5-26 |
| ENDFILE u | 5-26 |
| **FORMAT SPECIFICATION** | |
| sn FORMAT ($fs_1$, . . . , $fs_n$) | 6-5 |
| $fs_i$ — one or more field specifications separated by commas and/or grouped by parentheses | |
| **DATA CONVERSION** | |
| srEw.d — Single precision floating point with exponent | 6-9 |
| srEw.dEe — Floating point with specified exponent length | 6-9 |
| srEw.dDe — Floating point with specified exponent length | 6-9 |
| srFw.d — Single precision floating point without exponent | 6-13 |
| srGw.d — Single precision floating point with or without exponent | 6-14 |
| srDw.d — Double precision floating point with exponent | 6-16 |
| rIw — Decimal integer conversion | 6-7 |
| rIw.z — Integer with specified minimum digits | 6-7 |
| rLw — Logical conversion | 6-21 |
| rAw — Alphanumeric conversion | 6-18 |
| rRw — Alphanumeric conversion | 6-20 |
| rOw — Octal integer conversion | 6-17 |

| | | Page Numbers |
|---|---|---|
| rOw.z | Integer with specified minimum digits | 6-17 |
| rZw | Hexadecimal conversion | 6-18 |

s optional scale factor of the form: nP

r optional repetition factor

w integer constant indicating field width

d integer constant indicating digits to right of decimal point

e integer indicating digits in exponent field

z integer specifying minimum number of digits

| | | |
|---|---|---|
| nX | Intraline spacing | 6-24 |
| nH . . . / * . . . * / ≠ . . . ≠ | Hollerith | 6-25 |
| / | Format field separator; indicates end of FORTRAN record | 6-28 |
| Tn | Column tabulation | 6-32 |
| V | Display code substitution | 6-34 |
| = | Numeric substitution | 6-34 |

## OVERLAYS

CALL OVERLAY (fname,i,j,recall,k) — 7-23

| | |
|---|---|
| fname | name of file or overlay in H format |
| i,j | octal with a B or decimal equivalent overlay numbers |
| recall | if 6HRECALL is specified, the overlay is not reloaded if it is already in memory |
| k | L format Hollerith constant: name of library from which overlay is to be loaded |
| | any other non-zero value: overlay loaded from global library set |

OVERLAY (fname,i,j,Cn) — 7-22

| | |
|---|---|
| fname | name of file |
| i,j | overlay numbers |
| Cn | n is a 6-digit octal number indicating start of load relative to blank common |

| | | Page Numbers |
|---|---|---|

## DEBUG

| | | |
|---|---|---|
| C$ DEBUG | | 9-22 |
| C$ DEBUG ($name_1$ , . . . , $name_n$) | | 9-22 |
| C$ AREA $bounds_1$ , . . . , $bounds_n$<br>C$ DEBUG | within program unit | 9-23 |
| C$ AREA/$name_1$/$bounds_1$ , . . . , $bounds_n$ , . . . /$name_n$/$bounds_1$ , . . . , $bounds_n$<br>C$ DEBUG ($name_1$ , . . . , $name_n$)<br>or<br>C$ DEBUG | external debug deck | 9-23 |

| | | |
|---|---|---|
| bounds | ($n_1$,$n_2$) | $n_1$ initial line position<br>$n_2$ terminal line position |
| | ($n_3$) | $n_3$ single line position to be debugged |
| | ($n_1$,*) | $n_1$ initial line position<br>* last line of program |
| | (*,$n_2$) | * first line of program<br>$n_2$ terminal line position |
| | (*,*) | * first line of program<br>* last line of program |

| | | |
|---|---|---|
| C$ ARRAYS ($a_1$, . . . , $a_n$) | | 9-3 |
| C$ ARRAYS | | 9-3 |
| $a_i$ | array names | |
| C$ CALLS ($s_1$, . . . , $s_n$) | | 9-5 |
| C$ CALLS | | 9-5 |
| $s_i$ | subroutine names | |
| C$ FUNCS ($f_1$, . . . ,$f_n$) | | 9-7 |
| C$ FUNCS | | 9-7 |
| $f_i$ | function name | |
| C$ GOTOS | | 9-14 |

Page Numbers

C$ NOGO 9-17

C$ STORES ($c_1,\ldots,c_n$) 9-10

$c_i$ variable name

variable name .relational operator. constant

variable name .relational operator. variable name

variable name .checking operator.

checking operators:

RANGE out of range
INDEF indefinite
VALID out of range or indefinite

C$ TRACE (lv) 9-15

C$ TRACE 9-15

lv level number:

0 tracing outside DO loops

n tracing up to and including level n in DO nest

C$ OFF 9-26

C$ OFF ($x_1,\ldots,x_n$) 9-26

$x_i$ any debug option

## COMPASS SUBPROGRAM IDENTIFICATION

IDENT name (in column 11) 17-2

END indicates end of COMPASS subprogram 17-2

## LISTING CONTROL DIRECTIVES

C/ LIST, NONE 12-2

C/ LIST, ALL 12-2

# ARITHMETIC D

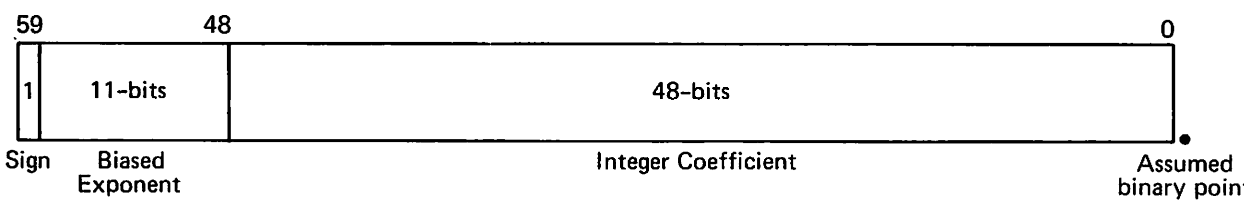
This section explains the internal format of numbers used in FORTRAN programs and the kinds of arithmetic performed on them. It is intended primarily to aid in reading octal dumps and interpret operating system mode error messages. The actual instructions generated for any sequence of code depend on the context of the code as well as the optimization level selected.

## INTERNAL FORMATS FOR VALUES

The internal format used to store a FORTRAN variable, array element, constant, or expression depends strictly on the type.

### REAL NUMBERS

Real numbers are stored in 60-bit floating point format as shown in figure D-1.




Bits 47 through 0 contain the coefficient of the number (equivalent to about 14 decimal digits). The binary point is considered to be at the right of bit 0. The exponent is biased by 2000 octal; that is, the exponent is represented by an 11-bit quantity (one's complement notation is used for negative numbers), 2000 octal is added to this quantity, and the low order 11 bits are used.

Additionally, real numbers are normalized. A normalized number is one in which bit 47 is the most significant bit; that is, bit 47 is different from bit 59. The special case of a word of all zero bits (positive zero) is also a normalized number. For every bit position that the coefficient is shifted to the left to achieve normalization, the exponent is reduced in value by one.

The sign of the number is represented by bit 59; the number is positive if bit 59 is 0 and negative if bit 59 is 1. Negative numbers are represented in one's complement form.

Minus zero (a word of all 1 bits) is considered to be equal to positive zero (a word of all zero bits) when the relational operators are used; minus zero is not considered less than positive zero. Minus zero is considered zero for arithmetic IF statements.

Table D-1 summarizes the configurations of bits 58 and 59 and the exponent and coefficient signs resulting from each combination.

TABLE D-1. BITS 58 AND 59 COMBINATIONS

| Bit 59 | Coefficient Sign | Bit 58 | Exponent Sign |
|---|---|---|---|
| 0 | Positive | 1 | Positive |
| 0 | Positive | 0 | Negative |
| 1 | Negative | 0 | Positive |
| 1 | Negative | 1 | Negative |

Some examples of floating point numbers, as they would appear in octal format, are as follows:

| Number | Octal Representation |
|---|---|
| +1. | 1720 4000 0000 0000 0000 |
| +100. | 1726 6200 0000 0000 0000 |
| -100. | 6051 1577 7777 7777 7777 |
| 1.E64 | 2245 6047 4037 2237 7733 |
| -1.E-64 | 6404 2570 0025 6605 5317 |
| 0. | 0000 0000 0000 0000 0000 |

## DOUBLE PRECISION

Double precision numbers occupy two consecutive words, each in the floating point format shown in figure D-1.

The first word contains the more significant part of the number, and the second word contains the less significant part.

Although complete arithmetic instructions using double precision arguments are not provided by the hardware, the FORTRAN Extended compiler generates code for true double precision by using instructions that give upper and lower half results with single precision arguments.

Some examples of double precision numbers, as they would appear in octal format, are as follows:

| | Octal Representation | |
|---|---|---|
| Number | First Word | Second Word |
| 7.834926843D137 | 26334153710320065255 | 25530330560116025671 |
| -2348585858574758224D12 | 57120455517237124716 | 57721332647630553777 |
| 0.0D0 | 00000000000000000000 | 00000000000000000000 |
| 1928374651928374650D256 | 35347102140424723427 | 34542402521200675526 |
| -0.0D0 | 77777777777777777777 | 77777777777777777777 |
| 1.234234234234234234234234234234 D0 | 17204737554312750737 | 16405543127507375543 |

## COMPLEX NUMBERS

Complex numbers are of the form a + bi, where the real part (a) occupies the first word, and the imaginary part (b) occupies the second word. Both words contain real numbers in the format shown in figure D-1. The formulas used for arithmetic for complex numbers are as follows:

$$(a + bi) \pm (c + di) = (a \pm c) + (b \pm d)\, i$$

$$(a + bi) * (c + di) = (ac - bd) + (ad + bc)\, i$$

$$(a + bi) / (c + di) = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}\, i$$

Some examples of complex numbers, as they would appear in octal format, are as follows:

| | Octal Representation | |
|---|---|---|
| **Number** | **First Word** | **Second Word** |
| (5.7,6.3) | 17225546314631463146 | 17226231463146314632 |
| (764E45,-12.2E-45) | 21604135136170411021 | 63013513153413026764 |
| (36.567985456983,2E-110) | 17254444263576544542 | 11436006220176715720 |

## INTEGERS

The full 60-bit word is used for internal representation of integers. Bit 59 is the sign bit (0 for a positive number, 1 for a negative number), and the other 59 bits represent the magnitude of the number. Only the lower 48 bits are used for multiplication and division in most cases, as well as for conversion from an integer to a real number; the full 60 bits are used for addition and subtraction. In the case of multiplication, division, and conversion, the upper 12 bits might be disregarded (except for the sign bit) without diagnostic if the operation takes place at execution time. Where constants are involved, the compiler might issue a diagnostic at compile time.

Some examples of integers, as they would appear in octal format, are as follows:

| Number | Octal Representation |
|---|---|
| $2^{47} - 1$ | 00003777777777777777 |
| 12131214121312 | 00000260420455254540 |
| -2 | 77777777777777777775 |

## LOGICAL VALUES

There are only two logical values: .TRUE. and .FALSE. .TRUE. is represented internally by a negative number (bit 59 is 1) and .FALSE. is represented internally by a positive number (bit 59 is 0).

## TYPELESS OPERANDS

Typeless operands include octal and Hollerith constants, masking expressions, and the values returned by the intrinsic functions AND, OR, XOR, COMPL, SHIFT, and MASK. Typeless operands are never converted but assume the type of the expression in which they occur. If they are used in an expression, the user must be aware of value associated with the bit pattern of a typeless operand in the context it is used (see Section 2).

Some examples of typeless operands, and their internal representation, are as follows:

| Operand | Octal Representation |
|---|---|
| 9ROR NOT 2B | 00172255161724553502 |
| 234 .OR. 4LYECH | 31050310000000000352 |
| SHIFT(COMPL(MASK(12)),4) | 00177777777777777760 |

## OVERFLOW

Overflow of the floating point range is indicated by a word whose upper 12 bits are $3777_8$ for a positive result and $4000_8$ for a negative result. These are the largest values that can be represented in floating point format, as shown in Table D-2. If the result of a computation has exactly $3777_8$ or $4000_8$ in the upper 12 bits, no error results immediately, but if the number is used subsequently, an error condition results. This situation is known as partial overflow.

Complete overflow occurs when an operand whose upper 12 bits would be larger than $3777_8$ or $4000_8$ is generated. Complete overflow also occurs when the result of a computation has a mathematically infinite value; for example, division by zero. Certain library functions return an infinite operand when called with invalid arguments. In the case of complete overflow, the upper 12 bits of the operand are set to $3777_8$ or $4000_8$ and the coefficient is set to all zero bits. The sign of the operand is the same as if the number had not exceeded the floating point range. Further action depends on the computer being used.

On a CYBER 70 Model 71, 72, 73, or 74, CYBER 170 Model 171, 172, 173, 174, or 175, or 6000 series computer, no action is taken unless the operand is used again. In this case, the error mode 2 flag (see below) is set. The program aborts and an error message is listed unless this error has been disabled by a MODE control statement or by installation option.

On a CYBER 70 Model 76, CYBER 170 Model 176, or 7000 series computer, the overflow condition flag is set in the Program Status Designator register as soon as complete overflow occurs. This flag causes an overflow message to be listed and the program to abort. This condition also results from the use of an operand that was not generated by an arithmetic operation.

## UNDERFLOW

Underflow occurs when the result of a computation would have a value less than $0000_8$ or $7777_8$ in the upper 12 bits. In this case, the word is set to all zeros.

On a CYBER 70 Model 71, 72, 73, or 74, CYBER 170 Model 171, 172, 173, 174, or 175, or 6000 series computer, no further action is taken.

On a CYBER 70 Model 76, CYBER 170 Model 176, or 7000 series computer, no action is taken unless underflow has been selected as a mode error by a MODE control statement or by installation default. In this case, the underflow condition flag is set in the Program Status Designator register as soon as the underflowed result is generated. This flag causes an underflow message to be listed and the program to abort.

TABLE D-2. FLOATING POINT REPRESENTATION

| | Positive Operand | Negative Operand |
|---|---|---|
| OVERFLOW | Complete Overflow 37770 . . .$0_8$<br>Partial Overflow 3777X . . .$X_8$ | 4000 . . .$0_8$<br>4000X . . .$X_8$ |
| LARGEST ABSOLUTE VALUE | ≅1.265014083171E+322 = 37767 . . .$7_8$<br>≅1.265014083171E+322 = 37767 . . .$7_8$ | ≅-1.265014083171E+322 = 40010 . . .$0_8$<br>≅-1.265014083171E+322 = 40010 . . .$0_8$ |
| SMALLEST NORMALIZED ABSOLUTE VALUE | ≅3.131513062514E-294 = 000140 . . .$0_8$ | ≅-3.131513062514E-294 = 777637 . . .$7_8$ |
| ZERO | 0 . . .$0_8$ | 7 . . .$7_8$ |
| INDEFINITE | 17770 . . .$0_8$ | 60007 . . .$7_8$ |

If underflow is selected as a mode error on the CYBER 70 Model 76, it is unlikely that a FORTRAN program will compile or execute successfully. Therefore, this error condition should not be enabled when FORTRAN programs are compiled or executed.

## INDEFINITE OPERANDS

An indefinite result is generated when a calculation cannot be resolved, such as a division operation when the divisor and dividend are both zero. The internal representation of an indefinite operand does not correspond to any number; the operand is represented by a minus zero exponent and a zero coefficient (17770 . . .$0_8$). Further action depends on the computer being used.

On a CYBER 70 Model 71, 72, 73, 74, CYBER 170 Model 171, 172, 173, 174, or 175, or 6000 series computer, no action is taken unless the indefinite operand is used again. In this case, the error mode 4 flag (see below) is set. The program aborts and an error message is listed unless this error has been disabled by a MODE control statement or by installation option.

On a CYBER 70 Model 76, CYBER 170 Model 176, or 7600 series computer, the indefinite flag is set in the Program Status Designator register as soon as the operand is generated. This flag causes a message to be listed and the program to abort.

## COMPUTATION WITH NON-STANDARD OPERANDS

If any mode error conditions have been disabled by the MODE control statement or by installation option, computations with these operands, which would normally cause the program to abort, can continue.

The following tables (D-3 through D-6) show the results of arithmetic operations using non-standard operands in all possible combinations. In these tables, W represents any value except infinite or indefinite, and N represents any positive value except infinite, indefinite, or zero.

TABLE D-3. NON-STANDARD ADD
X1 = X2 + X3

X3

| X2 | W | +∞ | -∞ | ±IND |
|---|---|---|---|---|
| W | - | +∞ | -∞ | IND |
| +∞ | +∞ | +∞ | IND | IND |
| -∞ | -∞ | IND | -∞ | IND |
| ±IND | IND | IND | IND | IND |

TABLE D-4. NON-STANDARD SUBTRACT
X1 = X2 - X3

X3

| X2 | W | +∞ | -∞ | ±IND |
|---|---|---|---|---|
| W | - | -∞ | +∞ | IND |
| +∞ | +∞ | IND | +∞ | IND |
| -∞ | -∞ | -∞ | IND | IND |
| ±IND | IND | IND | IND | IND |

TABLE D-5. NON-STANDARD MULTIPLY
X 1 = X2 * X3

X3

| X2 | +N | -N | +0 | -0 | +∞ | -∞ | ±IND |
|---|---|---|---|---|---|---|---|
| +N | - | - | 0 | 0 | +∞ | -∞ | IND |
| -N | - | - | 0 | 0 | -∞ | +∞ | IND |
| +0 | 0 | 0 | 0 | 0 | IND | IND | IND |
| -0 | 0 | 0 | 0 | 0 | IND | IND | IND |
| +∞ | +∞ | -∞ | IND | IND | +∞ | -∞ | IND |
| -∞ | -∞ | +∞ | IND | IND | -∞ | +∞ | IND |
| ±IND | IND | IND | IND | IND | IND | IND | IND |

TABLE D-6. NON-STANDARD DIVIDE
X1 = X2 / X3

X3

| X2 | | +N | -N | +0 | -0 | +∞ | -∞ | ±IND |
|---|---|---|---|---|---|---|---|---|
| | +N | - | - | +∞ | -∞ | 0 | 0 | IND |
| | -N | - | - | -∞ | +∞ | 0 | 0 | IND |
| | +0 | 0 | 0 | IND | IND | 0 | 0 | IND |
| | -0 | 0 | 0 | IND | IND | 0 | 0 | IND |
| | +∞ | +∞ | -∞ | +∞ | -∞ | IND | IND | IND |
| | -∞ | -∞ | +∞ | -∞ | +∞ | IND | IND | IND |
| | ±IND | IND | IND | IND | IND | IND | IND | IND |

## ARITHMETIC MODE ERRORS

Arithmetic mode errors occur when the central processor encounters an instruction whose execution is impossible or meaningless. The errors recognized, and the format of the message that is issued, vary depending on the operating system.

### NOS/BE 1 AND NOS 1 ERROR CONDITIONS

The following mode errors are issued under NOS/BE 1 and NOS 1:

| Mode | Error |
|---|---|
| 00 | Program stop (CYBER 70 series and CYBER 170 series only). Might result from attempting to execute a word of zeros or from a bad assigned GOTO statement or missing EXTERNAL statement. |
| 01 | Address out of range. A storage location outside the user's field length has been referenced. This error could be the result of an illegal array subscript, a call to an undefined subprogram, or a subroutine call with an incorrect number of arguments. |
| 02 | Infinite operand (defined above) |
| 03 | Infinite operand and address out of range |
| 04 | Indefinite operand |
| 05 | Indefinite operand and address out of range |
| 06 | Infinite or indefinite operand |
| 07 | Infinite operand and indefinite operand |

When executing on a CYBER 170 computer, the first digit might be nonzero, indicating a hardware error (as described in the appropriate operating system reference manual).

When an arithmetic mode error occurs, a message of the following type is issued:

time ERROR MODE = n.        ADDRESS = xxxxxx

where n is the error type and xxxxxx is the address in octal of the relative location where the error occurred.

## SCOPE 2 ERROR CONDITIONS

When an arithmetic error occurs under SCOPE 2, the following type of message appears in the dayfile under the headings shown below:

14.30.36*00012.059*SYS.        SC006 -        SCM DIRECT RANGE

CODE xxnnn

xx SC or JM — SC indicates System Control; JM, Job Management. System Control provides system overlay loaders and some communication between operating system overlays. Job Management controls user program input/output, and prepares user programs for execution.

nnn — Index number of the message.

MESSAGE AND MEANING — The message and an interpretation (if necessary) are printed.

LEVEL — Indicates the level of severity of the error as follows:

- X — Job terminates. No EXIT processing occurs.
- F — Job terminates. EXIT processing occurs.
- W — Warning is printed, and error is ignored. Processing continues, although the portion of the program containing the error may not be executed.
- I — Informative message is printed.

| CODE | MESSAGE AND MEANING | LEVEL |
|---|---|---|
| SC001 | LCM PARITY | F |
| SC002 | SCM PARITY | F |
| SC003 | LCM BLOCK RANGE | F |
| SC004 | SCM BLOCK RANGE | F |
| SC005 | LCM DIRECT RANGE | F |
| SC006 | SCM DIRECT RANGE | F |
| SC007 | PROGRAM RANGE | F |
| SC008 | BREAKPOINT | F |
| SC009 | STEP CONDITION | F |
| SC010 | INDEFINITE CONDITION | F |
| SC011 | OVERFLOW CONDITION | F |
| SC012 | UNDERFLOW CONDITION | F |
| SC040 | JOB MAKING 6000 REQUEST IN RAS+1; RAS+1 of user area is non-zero. | F |

# GLOSSARY E

This glossary does not include terms defined in the ANSI standard for FORTRAN, X3.9-1966.

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.)

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. 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.

BOI (Beginning-of-Information) – 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, exist prior to beginning-of-information.

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.

CALL BY NAME – A method of referencing a subprogram in which the addresses of the actual arguments are passed.

CALL BY VALUE – A method of referencing a subprogram in which only the values of the actual arguments are passed.

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 1 and NOS/BE 1 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 Extended 4, Sort/Merge 4, ALGOL 4, and the DMS-170 products is implemented through CRM. Neither the input/output of the NOS 1 and NOS/BE 1 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.

In this manual, the term CRM (or CYBER Record Manager) refers to the versions of Record Manager supported by NOS 1 and NOS/BE1; the term Record Manager refers to these versions plus the SCOPE 2 Record manager.

EOF(End-of-File) – A particular kind of boundary on a sequential file, recognized by 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

ENTRY POINT – A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

EOI (End-of-information) – 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.

EQUIVALENCE CLASS – A group of variables and arrays whose position relative to each other is defined as a result of an EQUIVALENCE statement.

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. Starts at beginning-of-information and ends at end-of-information.

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.

FIT (File Information Table ) – A table through which a user program communicates with 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.

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 to seven letters or digits, the first a letter. Files used in standard FORTRAN Extended input/output statements must be defined in the PROGRAM statement, and 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.

OPTIMIZING MODE – One of the compilation modes in the FORTRAN Extended compiler, indicated by the control statement options OPT=0, 1, and 2, or by omission of the TS option.

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 – 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.

| Device | RT | BT | Physical Representation |
|---|---|---|---|
| PRU device | W | I | A short PRU of level 0 containing one-word deleted record pointing back to last I block boundary, followed by a control word with flag indicating partition boundary. |
| | W | C | A short PRU of level 0 containing a control word with a flag indicating partition boundary. |
| | D,F,R,S,T,U,Z | C | A short PRU of level 0 followed by a zero-length PRU of level 17. |
| S or L format tape | W | I | 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 | 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,S,U,Z | C,K,E | Tapemark. |
| Any other tape format | Undefined. | | |

Notice that in a file with W type records a short PRU of level 0 terminates both a section and a partition.

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.

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 1 and NOS/BE 1, 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 no user data is called a zero-length PRU.

| Device | Size in Number of 60-bit Words |
|---|---|
| Mass storage | 64 |
| Tape in SI format with coded data† | 128 |
| Tape in SI format with binary data | 512 |
| Tape in X† or I format | 512 |
| Tape in other format | Undefined |

†Not supported under NOS 1

PRU DEVICE – A mass storage device or a tape in SI (NOS 1 and NOS/BE 1), I (NOS 1 and NOS/BE 1), or X (NOS/BE 1 only) format, so called because records on these devices are written in PRU's.

RECORD – 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 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 MANAGER – A generic term relating to the common products AAM and BAM that run under the NOS 1 and NOS/BE 1 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 Extended 4, Sort/Merge 4, ALGOL 4, and the DMS-170 products is implemented through CRM. Neither the input/output of the NOS 1 and NOS/BE 1 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.

In this manual, the term CRM (or CYBER Record Manager) refers to the versions of Record Manager supported by NOS 1 and NOS/BE 1; the term Record Manager refers to these versions plus the SCOPE 2 Record manager.

RECORD TYPE – The term record type can have one of several meanings, depending on the context. Record Manager defines eight record types established by an RT field in the file information table.

REFERENCE MAP – 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.

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 boht of 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.

| Device | RT | BT | Physical Representation |
|---|---|---|---|
| PRU device | W | I | Deleted one-word record pointing back to 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 | 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 | Short PRU with level less than 17 octal. |
| | D,F,R,T,U,Z | K | Undefined. |
| | S | Any | 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 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 | Any | Undefined. |
| Any other tape format | | | Undefined. |

The NOS 1 and NOS/BE 1 operating systems equate a section with a system-logical-record of level 0 through 16 octal.

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.

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.

SYSTEM-LOGICAL-RECORD - Under NOS/BE 1, 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.

TIME-SHARING MODE - One of the compilation modes in the FORTRAN Extended compiler, indicated by the TS control statement option.

UNIT DESIGNATOR - An integer constant, or an integer variable with a value of either 0 to 99 or an L format logical file name. In input/output statements, indicates on which file the operation is to be performed. It is linked with the actual file name by the PROGRAM 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 terminal or printed on a line printer. The image of cards input through the card reader or terminal also has such a terminator.

# INDEX

COMMENT SHEET

MANUAL TITLE: FORTRAN Extended Version 4 Reference Manual

PUBLICATION NO.: 60497800 REVISION: J

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

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE 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**

P.O. BOX 3492

Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD FOLD