5487

Burroughs

RELATIVE TO

~~III.0~~

III.1

B 7000/B 6000 Series

FORTRAN

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A.

January 1978

5001506

Burroughs

B 7000/B 6000 Series

FORTRAN

REFERENCE MANUAL

PRICED ITEM

Printed in U.S.A. January 1978 5001506

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

57-53

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001506-001 Date: June 1978

Publication Title: B 7000/B 6000 Series FORTRAN Reference Manual

Other Affected Publications: None

Supersedes:

Description:

Pages Changed	Pages Added
iii	PCN Page
v	6-2A
vii	7-2A
3-1	
5-3	
7-1	
7-5	
7-7	
10-1	
11-3	
11-5	
11-7	
12-1	
12-3	
12-21	
15-15	
18-3	
18-5	
18-9	
19-5	
19-7	
19-9	
19-17	
19-19	
20-1	
20-3	
B-5	
Index-3	
Index-7	
Index-11	

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue
Title thru ii	Original	12-22	PCN-001
iiA thru iiB	PCN-001	12-23 thru 12-24	Original
iii	Original	13-1 thru 13-16	Original
iv thru v	PCN-001	14-1 thru 14-5	Original
vi thru vii	Original	14-6	Blank
viii	PCN-001	15-1 thru 15-14	Original
ix thru x	Original	15-15	PCN-001
1-1 thru 1-4	Original	15-16	Blank
2-1 thru 2-2	Original	16-1 thru 16-7	Original
3-1 thru 3-2	PCN-001	16-8	Blank
3-3	Original	17-1 thru 17-6	Original
3-4	Blank	18-1 thru 18-3	Original
4-1 thru 4-10	Original	18-4 thru 18-6	PCN-001
5-1 thru 5-2	Original	18-7 thru 18-8	Original
5-3	PCN-001	18-9	PCN-001
5-4	Original	18-10 thru 18-11	Original
6-1 thru 6-2	Original	18-12	Blank
6-2A	PCN-001	19-1 thru 19-4	Original
6-2B	Blank	19-5 thru 19-7	PCN-001
6-3 thru 6-5	Original	19-8 thru 19-9	Original
6-6	Blank	19-10	PCN-001
7-1	Original	19-11 thru 19-16	Original
7-2 thru 7-2A	PCN-001	19-17 thru 19-19	PCN-001
7-2B	Blank	19-20	Blank
7-3 thru 7-5	Original	20-1 thru 20-2	PCN-001
7-6 thru 7-8	PCN-001	20-3	PCN-001
8-1 thru 8-12	Original	20-4	Original
9-1 thru 9-3	Original	A-1 thru A-3	Original
9-4	Blank	A-4	Blank
10-1	PCN-001	B-1 thru B-4	Original
10-2 thru 10-4	Original	B-5	PCN-001
11-1 thru 11-2	Original	B-6 thru B-9	Original
11-3	PCN-001	B-10	Blank
11-4	Original	C-1 thru C-4	Original
11-5 thru 11-8	PCN-001	Index-1 thru index-3	Original
11-9 thru 11-11	Original	Index-4	PCN-001
11-12	Blank	Index-5 thru index-6	Original
12-1	Original	Index-7	PCN-001
12-2 thru 12-3	PCN-001	Index-8 thru index-11	Original
12-4 thru 12-21	Original	Index-12	PCN-001

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001506-002 Date: November, 1979

Publication Title: B 7000/B 6000 Series FORTRAN Reference Manual

Other Affected Publications: None

Supersedes: N/A

Description:

Revisions to the text are indicated by a black vertical bar on the affected pages.

Replace pages

2-1
4-9
5-1
6-2A
10-1
13-5
13-13
15-15
17-3
19-15

Retain this PCN as a record of changes made to the basic publication.

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue	Page	Issue
Title thru ii	Original	10-1 thru 10-2	PCN-002	18-4 thru 18-6	PCN-001
iiA thru iiB	PCN-002	10-3 thru 10-4	Original	18-7 thru 18-8	Original
iii	Original	11-1 thru 11-2	Original	18-9	PCN-001
iv thru v	PCN-001	11-3	PCN-001	18-10 thru 18-11	Original
vi thru vii	Original	11-4	Original	18-12	Blank
viii	PCN-001	11-5 thru 11-8	PCN-001	19-1 thru 19-4	Original
ix thru x	Original	11-9 thru 11-11	Original	19-5 thru 19-7	PCN-001
1-1 thru 1-4	Original	11-12	Blank	19-8 thru 19-9	Original
2-1 thru 2-2	PCN-002	12-1	Original	19-10	PCN-001
3-1 thru 3-2	PCN-001	12-2 thru 12-3	PCN-001	19-11 thru 19-14	Original
3-3	Original	12-4 thru 12-21	Original	19-15 thru 19-16	PCN-002
3-4	Blank	12-22	PCN-001	19-17 thru 19-19	PCN-001
4-1 thru 4-8	Original	12-23 thru 12-24	Original	19-20	Blank
4-9 thru 4-10	PCN-002	13-1 thru 13-4	Original	20-1 thru 20-2	PCN-001
5-1 thru 5-2	PCN-002	13-5 thru 13-6	PCN-002	20-3	PCN-001
5-3	PCN-001	13-7 thru 13-12	Original	20-4	Original
5-4	Original	13-13 thru 13-14	PCN-002	A-1 thru A-3	Original
6-1 thru 6-2	Original	13-15 thru 13-16	Original	A-4	Blank
6-2A	PCN-002	14-1 thru 14-5	Original	B-1 thru B-4	Original
6-2B	Blank	14-6	Blank	B-5	PCN-001
6-3 thru 6-5	Original	15-1 thru 15-14	Original	B-6 thru B-9	Original
6-6	Blank	15-15	PCN-002	B-10	Blank
7-1	Original	15-16	Blank	C-1 thru C-4	Original
7-2 thru 7-2A	PCN-001	16-1 thru 16-7	Original	Index-1 thru Index-3	Original
7-2B	Blank	16-8	Blank	Index-4	PCN-001
7-3 thru 7-5	Original	17-1 thru 17-2	Original	Index-5 thru Index-6	Original
7-6 thru 7-8	PCN-001	17-3 thru 17-4	PCN-002	Index-7	PCN-001
8-1 thru 8-12	Original	17-5 thru 17-6	Original	Index-8 thru Index-11	Original
9-1 thru 9-3	Original	18-1 thru 18-3	Original	Index-12	PCN-001
9-4	Blank				

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO–West.

Rdok 12329

Burroughs

PUBLICATION CHANGE NOTICE

PCN No.: 5001506-003 Date: March 1981

Publication Title: B 7000/B 6000 Series FORTRAN Reference Manual (January 1978)

Other Affected Publications: None

Supersedes: N/A

Description

Major changes and implementations documented in this PCN include:

1. Deimplementation of the LEVEL parameter compiler option.
2. The FORTRAN library facility, including the BLOCK GLOBALS, EXPORT, CALL FREEZE, LIBRARY, and IN LIBRARY statements.

Revisions to the text are indicated by a black vertical bar on the affected pages.

Replace these pages		Add this page
v	18-3	12-20A
ix	19-3	
1-3	19-5	
6-3	19-7	
6-5	19-9	
8-1	19-11	
12-7	20-1	
12-9	20-3	
12-13	21-1	
12-15	21-3	
12-19	21-5	
13-3	B-1	
13-5	Index-1	
15-1	Index-3	
15-9	Index-5	
15-13	Index-7	
17-3	Index-9	
17-5		

Retain this PCN as a record of changes made to the basic publication.

The above pages covering
PCN 5001506-003

LIST OF EFFECTIVE PAGES

Page	Issue	Page	Issue	Page	Issue
Title thru ii	Original	10-1 thru 10-2	PCN-002	15-16	Blank
iiA thru iiB	PCN-003	10-3 thru 10-4	Original	16-1 thru 16-7	Original
iii thru iv	PCN-001	11-1 thru 11-2	Original	16-8	Blank
v thru vi	PCN-003	11-3 thru 11-8	PCN-001	17-1 thru 17-2	Original
vii thru viii	PCN-001	11-9 thru 11-11	Original	17-3 thru 17-6	PCN-003
ix thru x	PCN-003	11-12	Blank	18-1 thru 18-2	Original
1-1 thru 1-2	Original	12-1 thru 12-4	PCN=001	18-3 thru 18-4	PCN-003
1-3 thru 1-4	PCN-003	12-5 thru 12-6	Original	18-5 thru 18-6	PCN-001
2-1 thru 2-2	PCN-002	12-7 thru 12-10	PCN-003	18-7 thru 18-8	Original
3-1 thru 3-2	PCN-001	12-11 thru 12-12	Original	18-9 thru 18-10	PCN-001
3-3	Original	12-13 thru 12-16	PCN-003	18-11	Original
3-4	Blank	12-17 thru 12-18	Original	18-12	Blank
4-1 thru 4-8	Original	12-19 thru 12-20	PCN-003	19-1 thru 19-2	Original
4-9 thru 4-10	PCN-002	12-20A	PCN-003	19-3 thru 19-12	PCN-003
5-1 thru 5-2	PCN-002	12-20B	Blank	19-13 thru 19-14	Original
5-3 thru 5-4	PCN-001	12-21 thru 12-22	PCN-001	19-15 thru 19-16	PCN-002
6-1 thru 6-2	Original	12-23 thru 12-24	Original	19-17 thru 19-19	PCN-001
6-2A	PCN-002	13-1 thru 13-2	Original	19-20	Blank
6-2B	Blank	13-3 thru 13-6	PCN-003	20-1 thru 20-4	PCN-003
6-3 thru 6-5	PCN-003	13-7 thru 13-12	Original	21-1 thru 21-6	PCN-003
6-6	Blank	13-13 thru 13-14	PCN-002	A-3	Original
7-1 thru 7-2	PCN-001	13-15 thru 13-16	Original	A-4	Blank
7-2A	PCN-001	14-1 thru 14-5	Original	B-1 thru B-2	PCN-003
7-2B	Blank	14-6	Blank	B-3 thru B-4	Original
7-3 thru 7-4	Original	15-1 thru 15-2	PCN-003	B-5 thru B-6	PCN-001
7-5 thru 7-8	PCN-001	15-3 thru 15-8	Original	B-7 thru B-9	Original[
8-1 thru 8-2	PCN-003	15-9 thru 15-10	PCN-003	B-10	Blank
8-3 thru 8-12	Original	15-11 thru 15-12	Original	C-1 thru C-4	Original
9-1 thru 9-3	Original	15-13 thru 15-14	PCN-003	Index-1 thru 10	PCN-003
9-4	Blank	15-15	PCN-002	Index-11 thru 12	PCN-001

Correspondence regarding this document should be addressed directly to Burroughs Corporation, P.O. Box 4040, El Monte, California 91734, Attn: Publications Department, TIO—West.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

TABLE OF CONTENTS (Cont)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

GUIDE TO EFFICIENT USE

The purpose of this document is to present information about the FORTRAN programming language as implemented on the B 7000 and B 6000 series of computer systems. Specifically, this document describes both the programming language accepted by the B 7000/B 6000 series FORTRAN compiler and the various features of this compiler.

This manual is user-oriented in that it is designed to provide the FORTRAN programmer with a source of reference information pertinent to the manner in which the B 7000/B 6000 series system can be used.

This manual is logically divided into two distinct portions. The first portion consists of a discussion of the features of the B 7000/B 6000 series FORTRAN language, while the second discusses the B 7000/B 6000 series FORTRAN compiler. Chapters 2 through 17 describe the language, and it is to these chapters that the user is referred for programming information. Chapters 18 through 20 are devoted to a description of the compiler and contain information as to the manner in which FORTRAN programs are submitted to the compiler and the extent to which the user may control compiler operation.

Due to the fact that the basic features of the FORTRAN (FORmula TRANslation)[1] language have been standardized and documented elsewhere, this document is concerned primarily with providing information concerning the B 7000/B 6000 series implementation of this language. The present manual is not a primer in the language and should not be used as such. The first (and largest) portion of this document is devoted to a description of the constructs of this language as implemented on the B 7000/B 6000 series system. The discussion proceeds from basic language elements to general FORTRAN program structures. The pertinent chapters contain the following information:

Chapter 2 discusses source input format and program structure.

Chapters 3 through 5 discuss character sets, literals (i.e., constants and strings), and variables, the primary units of which the language is constructed.

Chapters 6 through 10 discuss FORTRAN expressions and statements, the computation-directed elements of the language.

Chapter 11 examines the manner in which storage is allocated for program data items.

Chapters 12 through 14 discuss I/O-related constructs.

Chapter 15 is concerned with FORTRAN subprograms and intrinsics.

Chapter 16 contains a discussion of language facilities allowing separate compilation and binding of program units and initiation of independent processes.

Chapter 17 discusses debugging and timing aids.

In addition, a number of appendices are provided at the end of this manual consisting of reference aids for the programmer.

[1]FORTRAN was originally developed for International Business Machines equipment.

B 7000/B 6000 SERIES FORTRAN LANGUAGE

The B 7000/B 6000 series FORTRAN language was designed so that many of the FORTRAN programs already in existence would be compatible with it with minimal changes. In particular, this language was designed to be compatible with FORTRAN IV, H level, and to contain ANSI X3.9-1966 Standard FORTRAN as a subset.

LANGUAGE DESCRIPTION CONVENTIONS

Throughout this document there are illustrations of the general formats of various language constructs. These illustrations follow certain conventions to be explained here.

The general format of a construct is presented with the user specified portions of the construct represented by lower-case letters and defined following the specification of the construct's format. The remaining characters in the specification are required portions of the construct being defined.

The proper format for an Example Item is:

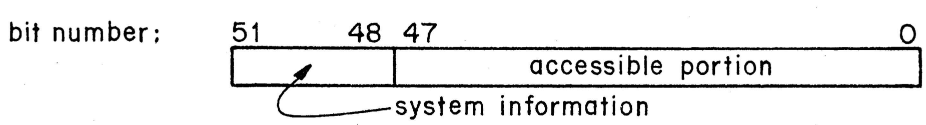
EX(s)MP,LE
where s is a string of from one to three characters, each of which is the character A.

This example presents the general format of a fictitious language construct called an "Example Item.". The portion of the construct over which the programmer has control is denoted by the lower-case letter s and is defined below the construct. The balance of the characters in the example format (i.e., all upper-case letters, the parentheses, and the comma) are required portions of the construct. As defined, valid examples of this sample construct would be:

```
EX(A)MP,LE
EX(AA)MP,LE
EX(AAA)MP,LE
```

Another convention employed in this document concerns the manner in which internal data words are described. The basic B 7000/B 6000 series data word is 52 bits in length and is described in detail in the B 6700 Hardware Handbook, Form No. 5000276. The term "48 accessible bits" is employed in this manual in relation to data word descriptions; this term refers to the 48 bits of the data word which are accessible to the user. The 52 bits of a data word are numbered 51 to 0, from left to right; the portion of the word which the user may directly access consists of bits 47 through 0.

The B 7000/B 6000 series data word is constructed in the following manner:



When it is necessary to illustrate data word fields, this document will display the 48 accessible bits in the following manner:

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

This data word layout is used since it facilitates the interpretation of the contents of the data word in terms of 4-bit hexadecimal digits. This is convenient since internal alphabetic processing on this machine is based on 8-bit EBCDIC characters.

BASIC FORTRAN CONCEPTS

Certain basic concepts concerning the FORTRAN language are presented here preliminary to the following discussion of the B 7000/B 6000 series implementation of this language.

A problem-solving procedure written in the FORTRAN language is called a FORTRAN source program.

Each program consists of one or more program units, including a required main program and possible additional subprograms. Some subprograms are provided by the system software and may be referenced by the source program.

Each program unit is constructed of a series of statements. These statements specify the arithmetic operations which are to be executed, control the order in which program statements are to be performed, accomplish various program input and output functions, describe program data items, or provide other program information.

Each program statement consists of a string of characters which is contained on one or more physical records (e.g., punched cards). The group of physical records containing an executable source program constitutes a source deck. This deck may be input as a file to a special computer program called a compiler which processes the input characters, generating an object program which the machine may execute.

RELATED DOCUMENTS

The information in the following Burroughs documents pertains to the material discussed in the present manual.

Publication	Form No.
B 7000/B 6000 Command and Edit (CANDE) Language Reference Manual	5011398
B 7000/B 6000 Input/Output Subsystem Information Manual	5001779
B 7000/B 6000 Mathematical Intrinsics Section, System Software Operational Guide, Volume No. 1	5011661
B 7000/B 6000 System/Binder Reference Manual	5001456
B 6700/B 7700 System Software Handbook	5000722

2. PROGRAM STRUCTURE

Every executable FORTRAN program consists of a sequence of statements, with each statement physically contained on one or more lines, or card images.

SOURCE INPUT FORMAT

Unless the compiler option FREE is set (see chapter 19), FORTRAN statements must be input to the compiler upon card images which have the following format.

Source input cards are in general free form, with these exceptions:

a. Columns 1 through 5 of a card image may contain a statement label. (See chapter 6.) This field is recognized as a label only on the first card image of an executable or FORMAT statement. On all nonexecutable statements other than the FORMAT statement and on all continuation card images, this field is ignored. A label without an associated statement is ignored. Blanks and preceding zeros in the label are ignored.
b. Column 6 of the first card image of a statement must be blank or contain a zero. A statement may be continued on several card images by placing any nonblank and nonzero character in column 6 of the continuation card images. As many continuation card images as necessary may be used.
c. Columns 7 through 72 of a card image contain the FORTRAN statement.
d. Columns 73 through 80 are normally not used by the FORTRAN compiler and may contain identification or sequencing information. This field is interpreted as a sequence number when the compiler options CHECK, INCLUDE, MERGE, NEWSEQERR or SEQERR are set (see chapter 19).

Two or more executable statements may be put on one card image if the statements are separated by a semicolon (;).

If a card image contains a percent sign (%) not in a string, the remainder of the card image (except the sequence number) is ignored and may be used for comments.

If a card image contains a "C" in column 1, the card image is considered a comment and is not interpreted except for the sequence number.

A card image containing a "$" in column 1 or a blank in column 1 and a "$" in column 2 is a compiler control card image as discussed in chapter 19.

Blanks are significant only in column 6 of a card image and in string literals. With these exceptions, blanks may be used freely or left out altogether without affecting the meaning of the FORTRAN program.

PROGRAM UNITS

Every executable FORTRAN program consists of exactly one main program which may be preceded or followed by as many subprograms as desired.

Each program unit consists of a sequence of statements terminated by an END statement.

END Statement

The nonexecutable END statement is provided for use as the terminal statement of a program unit.

The proper format for the END statement is:

```
END
```

Every program unit must contain exactly one END statement.

If an END statement is encountered during execution of a subprogram, a RETURN is implied; if an END statement is encountered in a main program, a STOP is implied. An END statement may be continued, but not labeled.

Main Program

The main program is the first (and perhaps only) program unit to be executed.

The main program may invoke SUBROUTINE and FUNCTION subprograms via the CALL statement and function references, respectively (subprograms are discussed in chapter 15). It is recommended that subprograms appear before their first reference, if possible. This will facilitate argument checking at compile-time and may improve efficiency.

Statement Ordering

The order of appearance of statements in a main program or subprogram body is determined by the following rules:

a. File declarations must precede the first FORTRAN statement of the program (see appendix B).
b. An IMPLICIT statement, when used, must be the first statement of the main program or the second statement of a subprogram, disregarding comments and file declarations.
c. DIMENSION, COMMON, EQUIVALENCE, and type statements must appear before any affected variables or arrays are referenced by a DATA statement.
d. If a DIMENSION, COMMON, or type statement contains any dimension information for an array, then the statement must precede any EQUIVALENCE statement containing that array.
e. FORMAT statements may appear at any point in a program unit.
f. Statement function declarations must precede the first reference to the function.

If optimization is requested (see Chapter 20), then the following additional restrictions apply:

a. DIMENSION, COMMON, type, and EQUIVALENCE statements must precede an executable statement referencing any of the affected variables or arrays.
b. An EXTERNAL statement must precede a function or subroutine reference containing the affected subprogram name as an actual parameter.

Thus, the recommended order in a program unit is as follows:

- File declarations.
- SUBROUTINE, FUNCTION, or BLOCK DATA statement, if appropriate.
- IMPLICIT statement.
- DIMENSION, COMMON, or type statements in any order.
- DATA statements.
- Statement function declarations.
- Remainder of program unit.
- END statement.

An example of a B 7000/B 6000 series FORTRAN program is given in appendix A.

3. CHARACTER SET

Characters are the elements of which a language is constructed. The B 7000/B 6000 series FORTRAN language is based upon a prescribed character set which is described in this chapter.

The B 7000/B 6000 series FORTRAN compiler accepts source input coded in EBCDIC, BCL, or BCD characters, though internal processing is carried out in EBCDIC.

EBCDIC INPUT

For EBCDIC input, the B 7000/B 6000 series FORTRAN character set may be described as consisting of these types of characters:

a. Digits
   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits
b. Letters
c. Basic FORTRAN characters
d. EBCDIC characters.

Digits

Octal digits consist of the characters 0, 1, 2, 3, 4, 5, 6, and 7. These digits are used to represent program values in terms of the octal (radix 8) number system.

Decimal digits consist of the characters in the octal digit set plus the characters 8 and 9. These digits are used to represent program values in terms of the decimal (radix 10) number system, and when the term "digit" is used in this manual it will refer to a member of the set of decimal digits.

Hexadecimal digits consist of the characters in the decimal digit set plus the characters A, B, C, D, E, and F. These digits are used to represent program values in terms of the hexadecimal (radix 16) number system, where A is equivalent to 10 in the decimal system, B is equivalent to 11 in the decimal system, etc.

Letters

Letters are defined as consisting of these 27 characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z $
```

The dollar sign ($) is included in the letter set (which is often referred to as the set of alphabetic characters) since it may be employed anywhere any other letter may be used.

Basic FORTRAN Characters

The basic FORTRAN characters consist of all digit and letter characters plus the characters:

```
" . , = + - * / ) ( & ' : blank
```

These characters are called the basic FORTRAN characters since they are the characters of which the FORTRAN constructs are built.

EBCDIC Characters

The EBCDIC (Extended Binary Coded Decimal Interchange Code) characters consist of the characters in the basic FORTRAN character set for EBCDIC input plus the other characters in the standard EBCDIC character set.

BCL INPUT

For BCL input, the B 7000/B 6000 series FORTRAN character set may be described as consisting of these types of characters:

a. Digits
   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits
b. Letters
c. Basic FORTRAN characters
d. BCL characters

Digits

Digit characters are defined in the same manner as for EBCDIC input.

Letters

Letter characters are defined in the same manner as for EBCDIC input.

Basic FORTRAN Characters

The basic FORTRAN characters are defined in the same manner as for EBCDIC input except that the apostrophe character ('), is replaced by the quote mark ('').

BCL Characters

The BCL (Burroughs Common Language) characters consist of the characters in the basic FORTRAN character set for BCL input plus the characters:

] ; % [@ ? #

BCD INPUT

For BCD input, the B 7000/B 6000 series FORTRAN character set consists of these types of characters:

a. Digits
   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits
b. Letters
c. Basic FORTRAN characters
d. BCD characters

Digits

Digit characters are defined in the same manner as for EBCDIC input

Letters

Letter characters are defined in the same manner as for EBCDIC input.

Basic FORTRAN Characters

The basic FORTRAN characters are defined in the same manner as for EBCDIC excepting that the apostrophe character ('), is replaced by the quote mark ('') and the ampersand (&) is not provided.

BCD Characters

The BCD (Binary Coded Decimal) characters (also called Hollerith characters) consist of the characters in the basic FORTRAN character set for BCD input plus the characters:

?] ;

CHARACTER CARD CODES

For EBCDIC-, BCL-, and BCD-coded punch card inputs to the FORTRAN compiler, each basic FORTRAN character has a corresponding card code. These character card codes are given in table 3-1. The codes for digit and letter characters are not presented in this table, since the codes involved are identical for EBCDIC, BCL, and BCD.

Table 3-1. Card Codes for Basic FORTRAN Characters

Basic FORTRAN Character	Character Name	Card Rows Punched		
		EBCDIC	BCL	BCD
=	Replacement Operator	8,6	8,5,0	8,3
+	Plus	12,8,6	12,0	12
–	Minus	11	11	11
*	Asterisk	11,8,4	11,8,4	11,8,4
/	Slash	1,0	1,0	1,0
(	Left Parenthesis	12,8,5	12,8,5	8,4,0
)	Right Parenthesis	11,8,5	11,8,5	12,8,4
,	Comma	8,3,0	8,3,0	8,3,0
.	Decimal Point	12,8,3	12,8,3	12,8,3
'	Apostrophe	8,5	N	N
''	Quote Mark	7,8	8,7,0	8,4
&	Ampersand	12	12	N
blank	blank	none	none	none

N = Not a basic FORTRAN character.

4. LITERALS

At the next level of language complexity are the literals: numeric constants and strings. These items are formed from the basic FORTRAN characters according to prescribed rules.

Literals function as FORTRAN data items used in problem solving and related operations such as input/output (I/O). The rules governing the use of these constructs are discussed here.

Literals may be divided into "numeric" (constant) and "non-numeric" (string) literals.

NUMERIC LITERALS

A constant numeric data item may be represented in the following ways:

a. integer constants.
b. real (floating-point) constants.
c. double precision constants.
d. complex constants.
e. octal constants.
f. hexadecimal constants.
g. logical constants.

These seven constant data representations are discussed in the following paragraphs; their internal storage requirements are discussed in chapter 11.

Integer Constants

The proper format for an integer constant is:

sn
where s is an optional sign character (+ or –) and n is a string of decimal digits representing an integer not greater than 549755813887.

The following are examples of valid integer constants:

```
   0          99999999999
  +0          03770
  -0          1254
17711         -5 7 (interpreted as -57)
```

The following are examples of invalid integer constants:

1.0	(decimal point not permitted; interpreted as a real constant)
3,000	(no commas or other punctuation permitted)
1548009000000	(maximum value exceeded; interpreted as a double precision constant)

Real Constants

The proper formats for a real constant are:

1. sm.n 2. smpnEx
where s is an optional sign character (+ or –), m and n are strings of decimal digits with a combined total not exceeding 11 significant digits (leading zeros are not significant digits), one of which (but not both) may be omitted, p is an optional decimal point, and x is an integer constant.

The second form of a real constant is called the scientific or exponential form. The value represented is the number preceding the E multiplied by 10 raised to the power denoted by the integer constant following the E. Thus, the real constant 3E2 represents the value 300, which is 3 times 10 to the second power. If the decimal point is omitted, it is assumed to immediately precede the E.

If the number of significant digits exceeds 11, approximately 11 significant digits will be retained (see chapter 11).

If either form of real constant has no sign, it will be interpreted as representing a non-negative value.

The maximum absolute value of a real constant that is permitted is approximately

4.3135914667 E68 = ((8**13)-1)*8**63

and the smallest nonzero absolute value permitted is approximately

8.7581154021 E-47 = 8**(-51).

Zero is also permitted.

The following are examples of valid real constants:

```
3.1415926535              2.5E+07 (same as 250000000)
0.                        1.023342E+8
0.0                       23456789012E-22
2E3  )                    0000000000007.
2.E3 } (equivalent)       .5  3E-4  2    (interpreted as .53E-42)
2E+3 )                    -253.
.075                      -.075
6.02E23                   2.9979E08
```

The following are examples of invalid real constants:

-1597	(no decimal point or E portion; interpreted as an integer constant)
6.2E+68	(exceeds maximum size limit)
6.2E-48	(smaller than minimum size limit)
E22	(exponent part alone not permitted; interpreted as a variable name)
2.7E1.2	(exponent part must be an integer)
1E2E3	(only one E portion allowed per constant)
2,765,987.	(no commas or other punctuation, except decimal point, permitted)

The internal machine representation of a real constant is similar to the internal representation of an integer constant. Thus, real and integer constants are often interchangeable in programs; however, care should be taken when intermingling such constants in expressions because the types employed if an expression will affect the type of the outcome of the evaluation of the expression (see chapter 9, ASSIGNMENT STATEMENTS).

Double Precision Constants

The proper formats for a double precision constant are:

1. si 2. sm.n 3. supvDx
where s is an optional sign character; i is a string of decimal digits representing a number greater than 549755813887; m and n are strings of decimal digits with a combined total of more than 11 significant digits (leading zeros are not significant digits), one of which (but not both) may be omitted; u and v are strings of decimal digits, one of which (but not both) may be omitted; p is an optional decimal point; and x is an integer constant.

If the number of significant digits in a double precision constant exceeds 22, approximately 22 significant digits will be retained (see chapter 11).

The maximum absolute value of a double precision constant that is permitted is approximately 1.94882838205028079124469 D+29603 = ((8**26)–1)*8**32754 and the smallest nonzero absolute value permitted is approximately 1.9385458571375858335564 D-29581 = 8**(–32755). Zero is also permitted.

The following are examples of valid double precision constants:

```
3.1415926535897932
3.1415926535897932D0      (equivalent)
3.1415926535897932D-0
1D3
+1D+003     (equivalent)
+1.D+3
1.D0003
1234567890.1234567890123 D 29594
1  2D-2 4259
6.63D-034
9.80665D+0
2.7182818284590452353603
27182818284590452353603
```

The following are examples of invalid double precision constants:

3.14159265	(less than 12 digits and no D portion; interpreted as a real constant)
2.7 D29604	(exceeds maximum size limit)
2.7 D-29585	(smaller than minimum size limit)
444	(smaller than 549755813887; interpreted as integer constant)
1.234,567,890,123.	(commas not permitted)
1.3E456	(no D in exponent part)

Complex Constants

The proper format for a complex constant is:

(m,n)
where m and n are the real and imaginary part, respectively, each of which may be an integer or real constant.

The complex constant (m,n) represents the quantity m+ni, where $i = \sqrt{-1}$.

The following are examples of valid complex constants with the equivalent mathematical expressions beside them:

Complex Constant	Equivalent Mathematical Expression
(123,456)	123 + 456i
(3.14159265,1597)	3.14159265 + 1597i
(0,.05)	0 + 0.05i (or 0.05i)
(100.,100.00)	100 + 100i
(1E2,0)	100 + 0i (or 100)
(0,-1)	0 + (-i) (or -i)

The following are examples of invalid complex constants:

(3,)	(missing imaginary part)
(12345678901234.,0)	(neither part may be a double precision constant)
(12.0,72.E68)	(imaginary part exceeds size limit for real constants)
(3,4,5)	(excessive number of parts - more than two)

Octal Constants

In some contexts, program values may be represented in other than decimal form in B 7000/B 6000 series FORTRAN. One of these alternate representations is the octal constant literal which corresponds to notation in a number system with a radix of 8.

The proper formats for an octal constant are:

1. sOm 2. Osm
where s is an optional sign character (+ or –) and m is a string of one to 16 octal digits.

Since an octal constant may contain up to 16 octal digits (each of which represents 3 bits of storage), this constant may specify the contents of all of the 48 accessible bits of the memory word used to store the constant value. When the minus (–) is used as the sign character, then the mantissa sign bit [46:1] of the word involved is set to the value opposite that specified by the digits of the constant. When the sign character is + or omitted, then the mantissa sign bit remains as specified by the octal digits contained in the constant.

For example, the sign bit of a storage word is set to 1 when its contents are specified by the octal constant O-200; this word then contains the internal representation of the integer constant –128 (see chapter 11, STORAGE ALLOCATION). Similarly, the sign bit of the word with contents specified by O-2000000000000000 is 0 since the minus sign in the octal constant returns bit 46 to zero after the 2 in the upper digit position of the constant sets it to one.

The following are examples of valid octal constants with their decimal equivalents:

Octal Constant	**Decimal Equivalent**
O0	0
O000000000000100	64
O100	64
O–77	–63
–O77	–63
O+00 01	1

The following are examples of invalid octal constants:

77763	(missing O)
O1234567812	(contains the decimal digit 8 which is illegal for an octal constant)
O0000000000005555	(contains more than 16 digits)
O 333.33	(contains a decimal point which is illegal for an octal constant)

An octal constant may only be used for data initialization in DATA and type statements.

Hexadecimal Constants

Another alternate representation of program values is the hexadecimal constant literal which corresponds to notation in a number system with a radix of 16.

The proper formats for a hexadecimal constant are:

```
1.  sZn
2.  Zsn
```

where s is an optional sign character (+ or –) and n is a string of one to 12 hexadecimal digits.

The optional sign character performs the same function as it performs for the octal constant, affecting only the mantissa sign bit (bit 46) of the word storing the specified value.

Each hexadecimal digit corresponds to a unique pattern of four bits. A list of these 4-bit patterns is given here with their corresponding hexadecimal ("hex") digits:

Hex Digit	Bit Pattern	Hex Digit	Bit Pattern
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Thus, for example, the string of hexadecimal digits 1A2 corresponds to this pattern of twelve bits: 000110100010.

The following are examples of valid hexadecimal constants with their decimal equivalents:

Hexadecimal Constant	Decimal Equivalent
Z0	0
Z10	16
Z100	256
-Z1000	-4096
Z-10000	-65536
ZF F	255

The following are examples of invalid hexadecimal constants:

FFF60	(missing Z)
Z-ABCDEFG	(contains the character G which is not a hexadecimal digit)
-Z100200300400A0	(contains more than 12 digits)
Z333.33 003	(contains a decimal point which is illegal for a hexadecimal constant)

A hexadecimal constant may only be used for data initialization in DATA and type statements.

Logical Constants

Two logical constants are provided to represent the logical values "true" and "false".

These two logical constants have the form:

.TRUE.

and

.FALSE.

The use of these logical constants is restricted to certain types of expressions (see discussion of logical expressions in chapter 6). The internal machine representations of these two constants are such that the data words corresponding to the constant .TRUE. and the integer constant 1 are identical and the data words corresponding to the constant .FALSE. and the integer constant 0 are identical.

NON-NUMERIC LITERALS: STRINGS

Constant non-numeric data items may be represented in B 7000/B 6000 series FORTRAN by string literals. This literal construct has two different configurations: the Hollerith string and the proper string. These two configurations may be used interchangeably and are distinguishable only by the differences in their formats.

The proper format for a Hollerith string is:

wHs
where w is a positive integer constant denoting the width of the string in characters and s is a string containing no more than 264 legal EBCDIC, BCL, or BCD characters, depending upon the input character format.

The proper format for a proper string is:

1. 's' 2. ''s''
where s is a string of no more than 264 valid EBCDIC characters. For BCL and BCD input, this string is delimited by quotes ('') and must consist of BCL or BCD characters respectively.

For EBCDIC, BCL and BCD input, the Hollerith string may contain any valid EBCDIC, BCL or BCD character respectively (see chapter 3, CHARACTER SET).

For EBCDIC input, the proper string may contain any valid EBCDIC character including the apostrophe (') and quote (''), but these characters are handled in a special manner. If the string delimiter is the apostrophe, then an apostrophe is interpreted as part of the string only if two adjacent apostrophes appear in the proper string in its place; these two apostrophes will be treated as representing one apostrophe character. If the string delimiter is the quote, then a quote is interpreted as part of the string only if two adjacent quotes appear in the proper string in its place; these two quotes will be treated as representing one quote character. Thus, the proper strings ''ABC'DE'' and 'ABC''DE' each represent the same string of six characters, ABC'DE.

For BCL and BCD input, the proper string may contain any valid BCL or BCD character, respectively, including the quote ("), but this character is handled in a special manner. A quote mark is interpreted as part of the string only if two adjacent quotes appear in the proper string in its place; these two quotes will be interpreted as representing one quote character. Thus, the proper string "ABC""DE" is interpreted as representing the quantity: ABC"DE.

Any blank characters within the character string will be considered part of the string.

The following are examples of strings valid for EBCDIC input (b represents a blank character):

```
2HbQ
4H"C:"
4H'C:'
'bSIDURibbb5'
020HHE SAID, "DON'T GO."    } 
'HE SAID, "DON''T GO."'     } (equivalent)
"HE SAID, '"DON'T GO.'""    }
'"b" IS A BLANK'
'%%%6%%%'
"ABCDIFGHJKL"
```

The following are examples of strings invalid for EBCDIC input:

3HbbbAB	(character string may not be longer than the integer preceding the Hh indicates)
-6HABCDEF	(string width specification may not be a negative integer)
0HA	(string width specification may not be zero)
'POIU'b'YT'	(this construct will be interpreted as two separate strings rather than one string containing an apostrophe since the two inner apostrophes are not adjacent)

The following are examples of strings valid for BCL or BCD input (b represents a blank character):

```
2HbQ
4H"C:"
"bSIDURIbbb5"
"bb'""ABCD"  } (equivalent)
7Hbb"ABCD    }
001Hb        } (equivalent)
"b"          }
"(b)  IS A BLANK"
"(MSG)  HERE"
```

The following are examples of strings invalid for BCL or BCD input:

3HbbbAB	(character string may not be longer than the integer preceding the H indicates)
-1HW	(string width specification may not be a negative number)
OHQ	(string width specification may not be zero)
"ABC"EFGH"	(a quote character in a BCL or BCD string is represented by a double quote)
"POIU"b"YT"	(this construct will be interpreted as two separate strings rather than one string containing a quote since the two inner quotes are not adjacent)

Strings are employed primarily in two ways: as representations of the value of a data item and as a format specification in a FORMAT statement (refer to chapter 13). When a string represents the value held by a data word, the string must contain: no more than the number of characters specified by the CHARS compiler option for single precision values and no more than twice the number of characters specified by the CHARS compiler option for double precision values (see the discussion of the CHARS compiler option in Chapter 19).

5. VARIABLES

A FORTRAN variable is an entity which has a symbolic name and which represents a datum. The value of the datum represented by a variable may be altered during program execution.

A variable identifies one or more storage locations for purposes of data storage and retrieval. The contents of these storage locations are accessed by referencing the associated variable name.

The internal handling of variables is discussed in chapter 11.

VARIABLE NAMES

A FORTRAN variable name is an identifier which consists of one to six alphanumeric characters (i.e., letter or digit characters), with the leading character being a letter (including $). If the variable name is more than six characters long, then only the first six characters of this identifier will be retained internally. Embedded blanks are ignored.

Variables may be classified into five fundamental types:

a. integer
b. real
c. double precision
d. complex
e. logical

The value represented by a variable of each of these types may be expressed by a constant of the same type. The values represented by each variable type must, therefore, obey the magnitude and significant-digit restrictions governing the corresponding type of constant.

Unless declared otherwise in a specific type statement or an IMPLICIT statement (see chapter 7), a variable will be assigned a type according to the initial character of its name. If this initial character is the letter I, J, K, L, M, or N, then the variable will by default be of type integer. If this initial character is any other letter (including $), then the variable will by default be of type real. No such defaults exist for types double precision, complex, or logical; variables of these types must be declared as such by appropriate type statements or IMPLICIT statements.

The following are examples of valid variable names with the type assigned to them when they are not affected by type or IMPLICIT statements:

AMK599	Type real
IF	Type integer (Note that this variable name is valid since there are no reserved words in B 7000/B 6000 series FORTRAN.)
$64000	Type real
END$Q	Type real
OF TEN	Type real; interpreted as OFTEN
LOOP3	Type integer
REALNUMBER	Type real; truncated to six characters and stored internally as REALNU.
ZIP WITH	Type real; truncated and stored as ZIPWIT

The following are examples of invalid variable names:

3LOOP	(Variable name may not begin with a digit character)
BE-GIN	(Characters other than letters, digits, or blanks are not allowed in a variable name)

ARRAY ELEMENTS

FORTRAN variables may be divided into simple variables (which are denoted by a variable name only) and array elements. An array element is a member of a data structure called an array, which is a nonempty ordered set of data organized into dimensions.

An array is referenced by an array name which is of the same form as a variable name. A variable name is designated as an array name by its appearance in an array declarator (see Chapter 7). All of the elements of an array are of the same type, and this type is indicated by the array name in the same manner as for the variable names.

Each member of an array is an array element. The proper format for an array element is:

a (s)
where a is an array, (s) is the subscript, and s is a list of subscript expressions separated by commas.

The number of subscript expressions must be equal to the number of dimensions in the array declarator except in an EQUIVALENCE statement (see Chapter 7).

A subscript expression is an integer, real or double precision expression. Subscript expressions which are not integer will be truncated to an integer value (see Chapter 11).

An array name may appear without a subscript list in:

a. The dummy argument list of a subprogram
b. The actual argument list of a subprogram reference
c. An I/O list
d. A COMMON, DATA, EQUIVALENCE, or type statement
e. A READ, WRITE, PRINT, or PUNCH statement as the format designator
f. A READ or WRITE statement as the file designator
g. A NAMELIST statement
h. A DEBUG MONITOR or DEBUG DUMP statement

Under certain circumstances, the subscript value of a subscript in an array element may fall outside the range declared for the array. The rule is: any subscript for an array is allowed which references an element of a COMMON block containing the array or an element of an array EQUIVALENCEd to the array or an element of an array associated with the array through argument passing.

Example:

```
COMMON /X/ A(10), Q, B(19)
COMMON T, C(10)
DIMENSION D(10)
EQUIVALENCE (C(8),D(3))
CALL S(A(5))
END
SUBROUTINE S(F)
COMMON /X/ R, E(9)
DIMENSION F(99)
RETURN
END
```

Array Name	Allowable Subscripts
A	1 TO 30
B	-10 TO 19
C	0 TO 15
D	-5 TO 10
E	0 TO 29
F	-3 TO 26

The following are examples of valid array elements:

`B(I)`	
`AMK 599(6)`	(interpreted as AMK599(6))
`ZIPWITH(3)`	(interpreted as ZIPWIT(3) when used where an array element may appear)
`J5(J5(3))`	(the subscript is itself an array element)
`ARRAY2 (1,1,1,1)`	
`A(R*T)`	
`Q  (3.7)`	(interpreted as Q(3))
`INT(1+9*SIN(I*PI10))`	

The following are examples of invalid array elements:

`A(A)`	(a subscript must be a valid arithmetic expression; an array name does not constitute such an expression)
`3ARRAY(6)`	(An array name must be a variable name)

A detailed discussion of the internal handling of FORTRAN arrays is contained in chapter 11, STORAGE ALLOCATION.

6. EXPRESSIONS AND STATEMENTS

This chapter discusses the manner in which expressions are constructed and evaluated. In addition, this chapter lists general features of the statements which form the basis of the FORTRAN language.

EXPRESSIONS

The purpose of expressions is to express a computation.

An expression consists of operands, operators, and parentheses. An operand is a valid constant, variable, function or intrinsic reference, or expression enclosed in parentheses. The expression represents the value obtained when the indicated operations are performed on the indicated values.

Expressions may be divided into two basic types according to the type of constant which may represent the value of the expression: arithmetic expressions and logical expressions.

Operators

The operators which may be used in a FORTRAN expression are listed in the following table with the relative precedence assigned to each operator by the compiler.

Operator	Type	Relative Precedence	Function Represented
**	Arithmetic	9	Exponentiation
unary -	Arithmetic	8	Change of sign
/	Arithmetic	7	Division
*	Arithmetic	7	Multiplication
-	Arithmetic	5	Subtraction
+	Arithmetic	5	Addition
.NE.	Relational	4	Not equal to
.GE.	Relational	4	Greater than or equal to
.GT.	Relational	4	Greater than
.IS.	Relational	4	48- or 96-bit identical to
.EQ.	Relational	4	Equal to
.LE.	Relational	4	Less than or equal to
.LT.	Relational	4	Less than
.NOT.	Logical	3	Logical negation
.AND.	Logical	2	Logical conjunction
.OR.	Logical	1	Logical disjunction

Operations of equal precedence are performed from left to right, except exponentiation which is carried out from right to left. The unary + operator is ignored. Parentheses may be used to override operator precedence in the usual manner.

Arithmetic Expressions

An arithmetic expression represents a rule for computing a value corresponding to a constant of type integer, real, double precision, or complex.

An arithmetic expression may contain only arithmetic operators, constants (except logical, octal, and hexadecimal constants), non-logical variables, non-logical function and intrinsic references, and grouping parentheses. In general, mixed operand types are permitted, with the exception that exponents may not be of type complex.

Immediately adjacent operators are not permitted; parentheses should be used to avoid adjacent operators. For example:

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

It is also permitted to use strings as operands, provided that such strings represent single precision values. If the string has fewer than the number of characters specified by the CHARS option, it will be stored left justified within the character area of the data word specified by the CHARS option and followed by blanks. Therefore, the expression 'ABCDEF' +4.7 is valid (for EBCDIC coded input when CHARS=6). The string represents a data word with the hexadecimal value ZC1C2C3C4C5C6, and 4.7 is stored as a real constant as explained in chapter 11. The words are added together as two numeric values.

Expression Types

The types of the operands in an arithmetic expression determine the type of the value obtained from the evaluation of the expression.

The following tables illustrate the resultant types of arithmetic operations depending upon the types of the operands and the operator involved. "DOUBLE" indicates type double precision.

For the operators +, –, *, and /, the result of the operation will be of the following type:

Type of First Operand	Type of Second Operand			
	Integer	**Real**	**Double**	**Complex**
Integer	INTEGER	REAL	DOUBLE	COMPLEX
Real	REAL	REAL	DOUBLE	COMPLEX
Double	DOUBLE	DOUBLE	DOUBLE	COMPLEX
Complex	COMPLEX	COMPLEX	COMPLEX	COMPLEX

For exponentiation (**) the result of the operation will be of the following type:

Type of Base	Type of Exponent			
	Integer	**Real**	**Double**	**Complex**
Integer	INTEGER	REAL	DOUBLE	ILLEGAL
Real	REAL	REAL	DOUBLE	ILLEGAL
Double	DOUBLE	DOUBLE	DOUBLE	ILLEGAL
Complex	COMPLEX	COMPLEX	COMPLEX	ILLEGAL

In the case of division involving two integer operands, the result will be truncated to an integer value. Thus, the expression 3/2 represents the value 1 and (–8)/3 represents the value –2, while the expression 3.0/2 represents the value 1.5.

The meaning of exponentation (**) depends on the values of the operands. The various meanings of Y**Z are as follows:

Type of First Operand (Y)	Type of Second Operand (Z)					
	Integer			Real or Double		
Value	**Z>0**	**Z=0**	**Z<0**	**Z>0**	**Z=0**	**Z<0**
Y>0	Note 1	1	Note 2	Note 3	1	Note 3
Y<0	Note 1	1	Note 2	Note 4	1	Note 4
Y=0	0	Note 4	Note 4	0	Note 4	Note 4

Note 1: Y**Z = Y*Y*Y...*Y (Z times)
Note 2: Y**Z = the reciprocal of Y*Y*Y...*Y (ABS(Z) times)
Note 3: Y**Z = EXP(Z*ALOG(Y))
Note 4: Value of expression is undefined

Examples

The following are examples of valid arithmetic expressions (all entities are non-logical):

```
6
1+6
SIN(3.14159*(-A))+2
AND(A(1),A(2))
20**FCN1(END(1,2),12)
-B*A
-(-P)
```

The following are examples of invalid arithmetic expressions:

`B**-A`	(illegal adjacent operators)
`Q,R`	(illegal character)
`2+A.AND.B`	(illegal logical operator)
`(A+B)   (C+D)`	(illegal implied multiplication)

Logical Expressions

A logical expression represents a rule for computing a value corresponding to .TRUE. or .FALSE. and consists of any valid combination of logical operands, logical operators, and parentheses. A logical operand may be a logical constant, logical variable, logical function reference, a relational expression, or a logical expression in parentheses.

Relational Expressions

A relational expression consists of two arithmetic expressions separated by one of the seven relational operators and represents the value .TRUE. or .FALSE.. The relation .IS. is true only if the data words referenced on either side of the operator are bit-by-bit identical. Only .EQ. and .NE. are permitted with complex items.

If a relational operator is used to compare strings, the bits representing the strings are treated as numerical data. This may produce unexpected results if a character (including a space) occupies bit positions used to represent the sign or exponent of numerical data (see chapter 11).

Logical Operators

Logical negation is expressed by the operator .NOT.. The operator .NOT. will complement the value of a logical operand.

The .AND. operator is used to express the logical product of two logical operands. The operation A.AND.B, will yield the value .TRUE. if and only if both A and B are .TRUE.. Otherwise the value of A.AND.B is .FALSE..

The .OR. operator is used to express the logical sum of any two logical operands. The operation A.OR.B, will yield the value .FALSE. only if both A and B are .FALSE.. Otherwise, the value of A.OR.B is .TRUE..

Examples

The following are examples of valid logical expressions (simple variables A, B, C, D and array L are logical):

```
A
B2.GT.B3
A.OR.L(3)
A.AND.Q.LE.SIN(R)+3.7
C.AND..NOT.D
.NOT.(A.OR.B).AND..NOT.(Q.IS.R)
J+1.EQ.  FCN(1,3,R,A)
```

The following are examples of invalid logical expressions:

`A+B`	(illegal operator)
`AND(Q,R)`	(the AND intrinsic returns a real value)
`J+1.OR.B+6`	(an arithmetic expression is not a logical primary)

STATEMENTS

Every executable FORTRAN program consists of a sequence of statements. These statements may be classified into executable and nonexecutable statements.

Executable Statements

An executable statement is a statement that will cause action to be taken whenever that statement is reached during execution of a program.

The B 7000/B 6000 series FORTRAN executable statements and the chapters in which they are discussed are:

Statement	Chapter
Assignment (including ASSIGN statement)	9
BACKSPACE	12
CALL	8
CALL EXIT	8
CALL FREEZE	21
CHANGE	12
CLOSE	12
CONTINUE	8
DO	8
ENDFILE	12
FIND	12
GO TO	8
IF	8
INQUIRE	12
LOCK	12
OPEN	12
PAUSE	8
PRINT	12
PUNCH	12
PURGE	12

Statement	Chapter
READ	12
RETURN	8
REWIND	12
STOP	8
WRITE	12
ZIP WITH	16

Nonexecutable Statements

A nonexecutable statement is a statement which gives information to the compiler regarding storage allocation, data initialization, I/O editing specifications, and program units.

The B 7000/B 6000 series FORTRAN nonexecutable statements and the chapters in which they are discussed are:

Statement	Chapter
BLOCK DATA	15
BLOCK GLOBALS	21
COMMON	7
DATA	10
DIMENSION	7
END	1
ENTRY	15
EQUIVALENCE	7
EXPORT	21
EXTERNAL	7
FILE	21
FORMAT	13
FUNCTION	15
IMPLICIT	7
IN LIBRARY	21
LIBRARY	21
NAMELIST	14
Statement function definition	15
SUBROUTINE	15
Type	7

Statement Labels

A statement may be optionally labeled so that it may be referred to in a DO statement, a GO TO statement, an IF statement, a READ, WRITE, PRINT, or PUNCH statement, or an actual argument list.

A statement label consists of a one- to five-digit unsigned integer constant. No statement label may appear on more than one statement in the same program unit. A statement label must appear anywhere in the first through fifth character positions of the initial card image of a statement (see chapter 2).

7. SPECIFICATION STATEMENTS

The nonexecutable specification statements are employed to supply compile-time information about the types, initial values, and storage allocation of variables and arrays, and to allow subprograms to be used as actual arguments.

The specification statements include the following:

Type statements
COMMON statement
DIMENSION statement
EQUIVALENCE statement
EXTERNAL statement
IMPLICIT statement

TYPE STATEMENTS

The type statements allow the type of a variable to be explicitly specified for a program unit. Such type specifications override any implicit typing due to the initial character of the symbolic name of the variable.

Variables may be assigned the following types:

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL

The proper formats for a type statement are:

```
1.  t m
2.  t m/v/ ...
```

where t is one of the types listed; m is a list of one or more simple variables, array names, array declarators, or function names separated by commas; v is an initial value list; and the ellipsis indicates as many repetitions of the form m/v/ or m as desired, with each repetition preceded by a comma.

Type statements must precede the use of affected variables in DATA statements and, if OPT=1, in executable statements. If the type statement contains an array declarator, then the type statement must precede any EQUIVALENCE statement referencing that array.

Array Declarators

An array declarator declares the size and number of dimensions of an array.

The proper format for an array is:

a (d)
where a is an array name and d is a list of dimension declarators separated by commas.

The number of dimensions in the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is 31.

The proper format for a dimension declarator is:

[d1:] d2
where d1 is the lower dimension bound and d2 is the upper dimension bound. The brackets indicate that the enclosed portion of the dimension declaration is optional.

The lower and upper dimension bounds are real or integer constants or simple variables.

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

An array declarator must precede the first use of that array in an executable, DATA, or EQUIVALENCE statement.

Each array declarator is either a constant array declarator or a variable array declarator. A constant array declarator is an array declarator in which each of the dimension declarators is a constant. A variable array declarator is an array declarator in which one or more of the dimension declarators are simple variables. Those bounds that are variables are called variable bounds. Variable bounds must be either declared in COMMON statements preceding the array declarator or be dummy arguments.

In addition, each array declarator is either an actual array declarator or a dummy array declarator. An actual array declarator is an array declarator in which the array name is not a dummy argument. An actual array declarator must be a constant array declarator and is permitted in DIMENSION, COMMON, and type statements. A dummy array declarator is an array declarator in which the array name is a dummy argument. A dummy array declarator may be either a constant array declarator or a variable array declarator. A dummy array declarator is permitted in DIMENSION and type statements, and may appear only in function or subroutine subprograms.

The size of a dimension is equal to the value of the dimension declarator for that dimension. The size of an array is equal to the number of elements in the array and is equal to the product of the sizes of the dimensions specified by the array declarator for that array. The size of an array must not exceed 65,535 elements.

Dimension declarators which are not integer are truncated to an integer value before being used to calculate the size of a dimension or the subscript value of a subscript.

Examples:

The array AR declared by

```
DIMENSION AR(3.78:7.21)
```

has 5 elements: AR(3), AR(4), AR(5), AR(6), and AR(7).

The array R declared by:

```
DIMENSION R(X,Y)
```

Where X=2.8 and Y=3.7 has 6 elements: R(1,1), R(2,1), R(1,2), R(2,2), R(1,3), and R(2,3).

Optional Size Specification

The items in the list of a REAL type statement may optionally be followed by the specification: *8. Such items will be assigned the type double precision.

For example, the type statement:

```
REAL I, J*8, K
```

will assign the type real to I and K and the type double precision to J.

The word REAL in a REAL type statement may optionally be followed by the specification: *8. The REAL*8 specification assigns the type double precision to all items in the list.

For example, the type statement

```
REAL*8 R,S,T
```

assigns the type double precision to R, S, and T.

Initial Value Lists

The type statement may be used instead of the DATA statement for variable initialization (see chapter 10). Type statements may optionally contain initial value lists constructed in the same fashion as the initial value list in a DATA statement.

The default initial value of a variable of type integer, real, or double precision is 0. The default initial value of a complex variable is (0,0). The default initial value of a logical variable is .FALSE..

Any number of simple variables may be initialized in one type statement initial value list. If an array is to be initialized, it must be the last declared item preceding the initial value list, which may not contain more values than the number of elements in the array.

If there are more simple variables in the type statement than there are values in the initial value list, then the latter portion of the simple variables will be initialized. Arrays are initialized from the beginning of the array.

For example, when this type statement is encountered during compilation:

```
REAL C, A(3)/1,"MNP"/, INT
```

the variables C and INT will retain their default initial values of zero, the array element A(1) will be initialized to 1, the array element A(2) will be initialized to MNPbbb if CHARS=6, and the array element A(3) will retain its default initial value of zero.

This type statement is illegal:

```
REAL X,Y(3)/1,2,3,4/
```

since the initial value list contains more values than there are elements in Y.

This type statement is valid:

```
REAL X,Y,Z/1,2/, A(2)/2*3/, B(3)/3*1/
```

In this example, Y and Z will be initialized to 1 and 2, respectively, each element of A will be initialized to 3, each element of B will be initialized to 1, and X will retain its default initial value of zero.

Examples:

```
REAL I,J,JTEST,D*8/1.2D3/
LOGICAL L1,L2/T/
DOUBLE PRECISION D
COMPLEX CC(N,6)
INTEGER A(20,20,4)/1600*1/
REAL*8 DOUBLE, DFCN1, $FCN/2*3.6D20/
```

COMMON STATEMENT

The nonexecutable COMMON statement associates simple variables and arrays with blocks of storage which may be shared among several program units.

The proper format for the COMMON statement is:

1. COMMON/n/v ... 2. COMMON v ...
where n is an optional common block name; v is a list of simple variables, array names, or array declarators; and the ellipsis indicates as many repetitions of the form /n/v as desired. No dummy arguments may appear in the list v.

Common Block Names

A symbolic name is associated with each block of common storage; this name is called a common block name. Any program unit may access the block of storage associated with this name via a common statement employing this name. Common storage associated with a common block name is called named common.

A common block name is constructed in the same manner as a variable name. This name may not be a dummy argument. A common block name is considered as such only in the context of a COMMON statement.

A common block need not be named; common storage associated with no name is called blank common and is assigned an internal identifier. These two statements are equivalent and define a blank common block:

```
COMMON//A,B(10)
COMMON A,B(10)
```

Storage Assignments

Simple variables and arrays are assigned contiguous locations in common storage in the order in which they appear in a COMMON statement.

Entire arrays but not individual array elements may be assigned storage locations in common storage. No common block may exceed 65,535 elements.

If the same common name appears more than once in a program unit, the common elements associated with a second or subsequent appearance are considered as extensions to the list of the previous appearances.

A discussion of the manner in which locations are allocated to elements of common storage is contained in chapter 11. Data initialization may be performed in a BLOCK DATA program unit or in the main program. (See chapter 15.)

A double precision or complex variable in a common block must be positioned such that the first of the two words containing the simple variable or each array element is always located at an odd-numbered word location in the block, where the first word of the block is number one.

Simple variables and array names may not be duplicated in COMMON statements. One simple variable or array may not be assigned to more than one block of common storage within a program unit.

Examples:

```
COMMON/BLOCK1/A,B(10),C//G,HOLD/BLOCK2/Q(3)
COMMON C
COMMON T1/CMN/T2,T3
```

DIMENSION STATEMENT

The nonexecutable DIMENSION statement specifies the size and number of dimensions of an array.

The proper format for the DIMENSION statement is:

DIMENSION d
where d is a list of array declarators separated by commas.

Each array referenced in a program unit must have its array bounds specified exactly once in an array declarator in that program unit (See TYPE STATEMENT in this chapter). An actual array declarator or a dummy array declarator may appear in a DIMENSION statement. An array declarator specifies exactly the amount of internal storage to be allocated to the array. For a dummy array declarator the total number of words of storage required by the array may not exceed the size of the actual array associated with it. Array storage is discussed in chapter 11.

Examples:

```
DIMENSION A(10)
DIMENSION B(N,2),C(6)
DIMENSION Q(J)
```

EQUIVALENCE STATEMENT

The nonexecutable EQUIVALENCE statement causes two or more simple variables or arrays referenced in the same program unit to share the same memory locations.

The proper format for the EQUIVALENCE statement is:

EQUIVALENCE (k) ...
where k is a list of two or more simple variables, array names, or subscripted array names separated by commas; and the ellipsis indicates as many repetitions of the form (k) as desired, with each repetition preceded by a comma.

No dummy argument may appear in an EQUIVALENCE statement. The items appearing in the list k will share some or all of the same storage locations. The subscript expressions of subscripted array names must be integer constants and must be either equal in number to the number of dimensions declared for the array or be one in number. When a subscripted array name in an EQUIVALENCE statement has as many subscript expressions as the array has dimensions, the subscripted array name denotes an array element.

Example:

If the array A is declared by

DIMENSION A(–5:5)

the statement

EQUIVALENCE (A(2),X)

equivalences X and the 8th element of the array A, which is A(2).

When the name of a multidimensional array appears in an EQUIVALENCE statement subscripted by only one subscript expression, that subscript expression denotes the linearized subscript value of the array being equivalenced.

Example:

If the array A is declared by

DIMENSION A(0:5, –5:–1)

the statement

EQUIVALENCE (A(2),X)

equivalences X and the second element of the linearized form of array A, which is A(1,–5) (see the discussion of how arrays are stored internally in Chapter 11).

An array name without a subscript denotes the first element of that array.

When two or more subscripted or unsubscripted array names appear in the same list k, the EQUIVALENCEd arrays are to overlap and be aligned in such a manner that the indicated elements share storage locations. No list may specify more than one element of the same array.

The EQUIVALENCE statement may be used to associate elements with a common block. This association may extend the common block beyond its former terminal point, thus increasing the size of the common block. Such an association may not extend a common block backwards to locations preceding the inital point of a common block. Two elements of common storage may not be made equivalent to one another, either directly or indirectly, by an EQUIVALENCE statement.

An EQUIVALENCE statement must precede any executable statement employing items EQUIVALENCEd.

Storage considerations are discussed in chapter 11.

Examples:

```
EQUIVALENCE (A,B,C) ,(A(3) ,R(9) ,G
EQUIVALENCE (D,E)
```

EXTERNAL STATEMENT

When an actual argument list of a function reference or a subroutine call contains a function or subroutine name, that name must appear in an EXTERNAL statement in the same program unit in which that reference or call appears.

The proper format for the nonexecutable EXTERNAL statement is:

EXTERNAL v
where v is a list of function or subroutine names separated by commas.

The EXTERNAL statement must appear before the function or subroutine reference. A statement function name may not appear in an EXTERNAL statement.

If a CALL statement or function reference appears in a subroutine or function that passes itself as an actual argument, then the subroutine or function must name itself in an EXTERNAL statement.

Any FORTRAN in-line intrinsic specified in an EXTERNAL statement will not be expanded in the code file and will be assumed to denote subprograms in the executable program (see chapter 15 for a list of in-line intrinsics).

The following are examples of valid EXTERNAL statements:

```
EXTERNAL EQUIV
EXTERNAL SUBA, FCN1
```

IMPLICIT STATEMENT

The nonexecutable IMPLICIT statement is an auxiliary type statement which allows the default types assigned to variable names according to their initial characters to be altered.

The proper format for the IMPLICIT statement is:

IMPLICIT t(c) ...
where t is a type name (see TYPE STATEMENTS), c is an initial character list, and the ellipsis indicates as many repetitions of the form t(c) as desired, with a comma preceding each repetition.

Only one IMPLICIT statement is allowed in a program unit, and any program unit may contain an IMPLICIT statement. If used, the IMPLICIT statement must be the first statement of a main program or the second statement of a subprogram (except for comments and file declarations). The IMPLICIT statement applies only to symbolic names in the program unit in which the statement appears, including function and entry names and dummy arguments.

Initial Character Lists

Each element of the list of initial characters may be either a letter or a pair of letters separated by a minus sign. If the element is a letter, a name beginning with that letter is assigned the specified default type. If the element is a letter pair, then names beginning with letters between and including the two letters specified are assigned the default type. The second of the letters in a letter pair must follow the first in the alphabet. The $ is assumed to follow Z.

Examples:

The following are valid IMPLICIT statements:

```
IMPLICIT REAL (L-N)     } These statements have identical effect.
IMPLICIT REAL (A-$)     }
IMPLICIT DOUBLE PRECISION(D)
IMPLICIT LOGICAL(A-C,L), REAL (D-F), COMPLEX(X)
```

8. CONTROL STATEMENTS

The executable control statements are used to alter the normal flow of program execution. These statements may transfer control to another part of the program, terminate or suspend execution, or control iterative processes.

The control statements consist of the following:

CALL statement
CALL EXIT statement
CONTINUE statement
DO statement
GO TO statement
IF statement
PAUSE statement
RETURN statement
STOP statement
ZIP WITH statement (see chapter 16).

CALL STATEMENT

The executable CALL statement causes the specified subroutine to be executed.

The proper formats for the CALL statement are:

1. `CALL s` 2. `CALL s (a)`
where s is the name of a subroutine subprogram or an subroutine entry name in this program and a is a list of actual arguments separated by commas.

The elements of the list of actual arguments must agree in number, order of appearance, and type with the dummy arguments appearing in the SUBROUTINE statement or in the ENTRY statement appearing in the called subroutine. These actual arguments are evaluated in left-to-right order before the subroutine is entered.

If a dummy argument is real, an integer actual argument may be used, and vice versa. An actual argument in a subroutine reference may be one of the following items:

a. A string.
b. A simple variable.
c. An array element.
d. An array name.
e. An expression.
f. A FUNCTION or SUBROUTINE name (see Chapter 15 for exceptions).
g. An item of the form: &n, where n is a statement label.

Execution of a CALL statement results in an association of actual arguments with all appearances of dummy arguments in the subroutine body.

Following these associations, control is transferred to the first executable statement following the called SUBROUTINE or ENTRY statement.

If an actual argument is an array element with an arithmetic expression as a subscript expression, the subscript expression is evaluated at the time of the execution of the CALL statement, and the resulting array element is associated with the corresponding dummy argument in the subroutine.

Array Handling

If a dummy argument of a subroutine is an array name, the corresponding actual argument in the CALL statement may be an array name or an array element. The actual argument may also be a string if the corresponding dummy argument is a single precision array name. The following correspondence is established between the actual array and the dummy array at the time the CALL statement is executed.

If the actual argument is an array name, the entire array may be accessed in the subroutine. The dummy array may have as many dimensions as desired. The correspondence between the internal storage locations assigned to the actual array and to the dummy array will be established as described in chapter 11. The dummy array declarator in the subroutine serves two purposes. It determines the correspondence between actual array elements and dummy array elements; it also determines the number of array elements referenced when the dummy array name appears without a subscript.

For example, if the array A is declared in the statement:

```
REAL A(20)
```

and is passed as an actual argument to the dummy argument B which is declared in the subroutine by the statement:

```
REAL B(5)
```

then all twenty elements of A may be referenced through B (for example B(18) is not an invalid array element), but the statement:

```
WRITE(6,1) B
```

will cause only five items to be output.

If the actual argument is an array element, the array from that element on may be accessed in the subroutine. Considering the preceding example, if the array element A(11) had been passed to the subroutine instead of A, then the last ten elements of this array (A(11) through A(20)) could be referenced through the dummy array B.

The following program illustrates the correspondence between a two-dimensional actual argument array A and a corresponding two-dimensional dummy argument B. The storage locations for B consist of the first 15 storage locations of A as described in chapter 11.

```
      INTEGER A(4,6)
      DO 2 I=1,4
      DO 1 J=1,6
    1 A(I,J)=10*I+J)
    2 PRINT/,(A(I,J),J=1,6)
      CALL S(A)
      END

      SUBROUTINE S(B)
      INTEGER B(3,5)
      DO 1 K=1,3
    1 PRINT/,(B(K,J),J=1,5)
      RETURN
      END
```

output:

```
      11, 12, 13, 14, 15, 16,
      21, 22, 23, 24, 25, 26,
      31, 32, 33, 34, 35, 36,
      41, 42, 43, 44, 45, 46,
      11, 41, 32, 23, 14,
      21, 12, 42, 33, 24,
      31, 22, 13, 43, 34,
```

Subroutine Returns

If the subroutine does not cause termination of the program, it may return control to the next executable statement following the CALL statement or to an executable statement whose label appears in the actual argument list preceded by an ampersand (&). Refer to the RETURN statement in this chapter for a further discussion of subroutine returns.

Examples:

```
      CALL SUBA(A,B(2), 3HABC, 'TITLE', J+20)
      CALL DATE(3)
```

CALL EXIT STATEMENT

The executable CALL EXIT statement is provided to allow the termination of an executing program.

The proper format for the CALL EXIT statement is:

```
CALL EXIT
```

The execution of this statement within a program which does not contain a subroutine named EXIT produces exactly the same result as the execution of a STOP statement.

CONTINUE STATEMENT

The executable CONTINUE statement is provided primarily as a program documentation aid as the execution of this statement produces no action.

The proper format for the CONTINUE statement is:

```
CONTINUE
```

Examples:

```
      .
      .
      .
      DO 1 I=2,10,2
      A(I)=I/M
      PRINT 100,A(I)
      IF (A(I)) 3,1,1
    1 CONTINUE
      M=-M
    3 CONTINUE
      .
      .
      .
```

In this example, the CONTINUE statement labeled 1 is used as the final statement of a DO loop, and the CONTINUE statement labeled 3 is used as a transfer point for an arithmetic IF statement. The use of the first CONTINUE allows the DO loop to be clearly delimited. The second CONTINUE statement allows transfer to an arbitrary point in the program, the location of which may be changed merely by changing the location of the CONTINUE statement.

DO STATEMENT

The executable DO statement is a control statement that allows a series of statements to be repeatedly executed while the value of a specified variable is varied between specified limits. The number of iterations is dependent upon this control variable.

The proper formats for the DO statement are:

1. DO s v=m,n,i 2. DO s v=m,n
where s is the label of an executable statement following the DO statement in the same program unit, v is the integer or real control variable, and m, n, and i are the initial parameter, terminal parameter, and incrementation parameter, respectively, each of which is an integer or real arithmetic expression.

The DO statement causes repeated execution of the statements in its range. The range of a DO statement is defined to be the first executable statement following the DO statement up to and including the statement whose label is specified in the DO statement.

If the optional incrementation parameter is omitted, a value of 1 is assumed.

After each execution of the range of the DO, and after the control variable has been incremented by the value of the incrementation parameter, a test is made to determine if the value of the control variable is greater than the value of the terminal parameter. If so, the DO is considered satisfied and control is transferred to the first executable statement after the range of the DO. If not, the range of the DO is executed again. In any case, the range of a DO is executed at least once. If the increment is negative, the range of the DO will be executed exactly once or will be executed an infinite number of times.

The following program prints out the cosine of the numbers –5.4, –5.3, –5.2, and –5.1.

```
   DO 8 X = -5.4, -5.05, .1
8  PRINT/, COS(X)
   END
```

The following program initializes A(1) through A(10) and B(1,1),..., B(10,10) to the value 5.

```
   J=10
   DO 2 I=1,J
   A(I)=5.
2  B(I,I)=5.
   END
```

The control variable is available for use within the range of the DO statement and may be modified as desired. The incrementation parameter and terminal parameter are reevaluated prior to their use and may also be modified by any statement within the range. There is no restriction on transfers out of or into the range of a DO. If a transfer is made into the range of a DO, the programmer is responsible for the appropriate assignment of a value to the control variable.

Example:

```
          K=100
          DO 24 INTGR=2,K,2
          SUM=SUM + INTGR
24        K=K-2
```

In this example, the variable K is diminished in value by 2 each time that the DO loop is executed. This will cause termination of the DO loop when INTGR=52 rather than 100 as was initially declared.

Nesting

Any number of DO statements may be nested one within another with the following restriction: if a DO statement occurs in the range of another DO, the range of the former must be completely contained within the range of the latter. Both may, however, specify the same statement as the last statement in their ranges. For example:

```
      DO 10 I=1,23
      DO 10 J=3,I-1
10    C(I,J+1) = I*J
```

When several DO statements share the last statement of their ranges, the control variable of one DO statement is not incremented and tested until each DO statement within its range is satisfied. When the outermost DO statement is satisfied, control continues with the next executable statement following its range.

GO TO STATEMENT

The executable GO TO statement transfers control from one point of an executing program to another point in the same program unit.

The proper formats for the GO TO statement are:

1. GO TO n 2. GO TO v 3. GO TO v,(m) 4. GO TO (m),a
where n is the label of an executable statement in the same program unit, v is an integer simple variable, m is a list of statement labels separated by commas, and a is an arithmetic expression. In forms 3 and 4, the comma is optional.

Unconditional GO TO

Form 1 represents the unconditional GO TO statement. Execution of this statement causes the executable statement in this program unit bearing the indicated label to be the next statement executed.

For example, the statement GO TO 23 causes the statement labeled 23 to be the next statement executed.

Assigned GO TO

Forms 2 and 3 represent the assigned GO TO statement. The execution of this statement causes control to be transferred to the statement whose label was last assigned to the simple integer variable v by an ASSIGN statement (see chapter 9). The list of labels which may optionally appear in this statement may be used as a documentation aid in listing the labels to which transfer may be made. This list is ignored during execution.

Thus, execution of the statements:

```
ASSIGN 23 TO K
GO TO K
```

will cause control to be transferred to the statement labeled 23.

Failure to assign a statement label to the variable v with an ASSIGN statement will cause program termination when the assigned GO TO is executed.

Computed GO TO

Form 4 represents the computed GO TO statement. Execution of this statement causes control to be transferred to a statement whose label appears in the list or to the next executable statement following the GO TO statement. How control is transferred depends on the value of the arithmetic expression following the list. It is evaluated, truncated if not an integer, and is used to select one of the labels in the list. If the expression has the value n, control passes to the n-th label in the list. If there are fewer than n labels in the list or if n is less than or equal to zero, control passes on to the next executable statement following the GO TO statement.

An example of a computed GO TO statement is:

```
GO TO (1,25,3,6,1,17), J+1
```

At execution time, the value of J+1 is computed. If J+1 = 4, then control will pass to the statement labeled 6, the fourth label in the list. If J+1 = 1 or 5 in this example, control will pass to statement number 1. If J+1 is less than 1 or greater than 6, control will pass to the next executable statement after the GO TO statement.

Notice also that the statement:

```
GO TO (3), J
```

is not the same as:

```
GO TO 3
```

The latter case is unconditional, but in the former case control will pass to statement number 3 only if J=1. If J has any other value, control will pass on to the next executable statement, which may or may not be statement number 3.

IF STATEMENT

The executable IF statement transfers control from one point to another in the same program unit depending upon an arithmetic value, or conditionally executes a statement.

The proper formats for the IF statement are:

1. IF(a) n,z,p 2. IF(e) s
where a is an arithmetic expression; n, z, and p are the labels of executable statements in the same program unit, e is a logical expression; and s is any executable statement except a DO statement or a form 2 IF statement.

Arithmetic IF

Form 1 of the IF statement represents an arithmetic IF statement. The arithmetic IF statement is a three-way branch. The arithmetic expression inside the parentheses is evaluated, and control is transferred to the statement bearing the first, second, or third label, depending on whether the value of the expression is less than zero, zero, or greater than zero, respectively.

Since complex numbers are not ordered as negative or positive, the expression in an arithmetic IF may not be of complex type.

An example of an arithmetic IF is:

```
IF (I-J) 3, 986, 17
```

If I-J is negative, control will be transferred to the statement labeled 3; if zero, to the statement labeled 986; and if positive, to the statement labeled 17. Notice that this is actually a test of whether J is greater than, equal to, or less than I.

The three statement labels in an arithmetic IF need not be different. For example:

```
IF((A-2)*(B-3)) 10, 10, 3
```

Here control will pass to statement labeled 3 only if (A-2)*(B-3) is greater than zero. Otherwise, control will pass to statement number 10.

Logical IF

Form 2 of the IF statement represents a logical IF statement. The logical IF will conditionally execute a statement. The logical expression enclosed in parentheses is evaluated. If this logical expression is .TRUE., then the statement following the logical expression is executed. If it is .FALSE., then the statement following the logical expression is ignored and control passes on to the next executable statement following the IF statement.

The statement following the logical expression may be any executable statement except a DO statement or another logical IF.

Examples of logical IF statements:

```
IF (A.EQ.B.OR.C.EQ.D) G=G+1
```

If A equals B or C equals D (or both), then G will be incremented by 1. Otherwise, G will remain unchanged. In any event, control will then pass on to the next statement.

```
IF (L1) GO TO 97
```

If L1 is .TRUE., then control will pass to statement number 97. If L1 is .FALSE., then control will pass on to the next statement.

```
IF (A.LE.97) IF (B) 12,12,13
```

If A is less than or equal to 97, then the arithmetic IF will be executed, and control will pass on to statement number 12 or 13, depending on the value of B. If A is greater than 97, control will pass on to the next statement.

PAUSE STATEMENT

The executable PAUSE statement allows an executing program to be suspended indefinitely.

The proper format for the PAUSE statement is:

1. PAUSE 2. PAUSE p
where p is a one- to eight-digit unsigned integer constant or a string containing up to 17 characters.

The execution of the PAUSE statement causes the unconditional suspension of the program being executed pending operator action. The statement may not appear in a BLOCK DATA subprogram. In addition to suspending the program, the execution of this statement causes the optional integer or string following the PAUSE to be displayed at the Operator Display Terminal.

The program may be resumed at the first executable statement following the PAUSE statement by a system input message from the Operator Display Terminal of the form:

```
mOK
```

where m is the mix number of the job. Alternately, the job may be discontinued at this point by a system input message of the form:

```
mDS
```

where m is the mix number of the job. If the program is run with CANDE, the ?OK and ?DS CANDE control commands may be used for the mOK and mDS system input messages, respectively.

Examples:

```
PAUSE
PAUSE 2
PAUSE 'BEFORE PUNCHING'
PAUSE "BEFORE PUNCHING"
```

RETURN STATEMENT

The executable RETURN statement specifies the manner in which the calling program unit is to be reentered following the execution of a subprogram.

The proper formats for the RETURN statement are:

1. RETURN 2. RETURN a
where a is an arithmetic expression.

A RETURN statement in a main program is interpreted as a STOP statement.

Standard Return

Form of the 1 RETURN statement causes a standard return to the calling program unit when executed. For a subroutine, the next executable statement following the CALL statement invoking the subroutine will be the next statement executed. For a function, the calling program unit will continue executing from the point of the function reference.

If the END statement of the subprogram is encountered before a RETURN statement is executed, a standard return will be performed.

Alternate Return

Form 2 of the RETURN statement returns control to a specified labeled executable statement in the calling program unit when executed. An alternate return from a function will not return a value for the function reference.

When an alternate return is executed, the arithmetic expression is evaluated. If the value of this expression is not an integer, the value is truncated to an integer. The resulting final value, n, is used to select the n-th asterisk (*) in the dummy argument list of the subprogram. If n is greater than the number of asterisks in this list or if n is less than or equal to zero, execution is terminated. If the expression is an integer constant which is zero, negative, or larger than the number of asterisks, a syntax error results.

The statement label in the actual argument list corresponding to the selected asterisk identifies the statement to which control is to be returned.

Example of Alternate Return

The following program excerpt illustrates a calling program unit able to handle alternate returns from the called subprogram.

```
      CALL SUBA(A1,&33,A2,&20,11)
15    Z=A1 + N
20    Z=Z + A2
33    Z=Z + A1
```

The called subroutine could be the following:

```
      SUBROUTINE SUBA (A,*,B,*,J)
      IF (A) 5,6,7
5     RETURN 1
6     RETURN A+2
7     RETURN
      END
```

First suppose that when the IF statement in the subroutine is executed, A is greater than zero. Then there will be a branch to the statement labeled 7, which is a standard return. The subroutine will return to the calling program unit at the statement following the CALL statement, the statement labeled 15.

Second, suppose that A is less than zero. When the IF statement in the subroutine is executed, there will be a branch to the statement labeled 5. This is an alternate return to the first label in the actual argument list. Return will be to the statement labeled 33 in the calling program unit.

If A is equal to zero when the IF statement is executed, there will be a branch to the statement labeled 6. The arithmetic expression evaluates to 2, causing a return to the statement labeled 20 in the calling program unit.

Had a RETURN 3 or RETURN 0 been attempted in SUBA, execution would have been terminated.

STOP STATEMENT

The executable STOP statement allows termination of an executing program.

The proper formats for the STOP statement are:

1. STOP 2. STOP s
where s is a one-to six-digit unsigned integer constant.

Execution of the STOP statement causes unconditional termination of the program being executed. This statement may not appear in a BLOCK DATA subprogram.

The optional integer following the STOP is ignored.

Examples:

```
STOP
STOP 99
```

ZIP WITH STATEMENT

The executable ZIP WITH statement is discussed in chapter 16.

9. ASSIGNMENT STATEMENTS

The executable assignment statements allow an arithmetic value, logical value, or label to be assigned to a variable.

The proper formats for the assignment statement are:

1. v=e 2. ASSIGN n TO k
where v and e are either a non-logical variable and an arithmetic expression or a logical variable and a logical expression, respectively; n is a statement label, and k is an integer simple variable.

ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement is represented by form 1 of the assignment statement, with v being a non-logical variable and e being an arithmetic expression.

When such a statement is executed, the arithmetic expression is evaluated and the value obtained is placed into the storage word or word pair allocated to the variable, v.

In general, the variable and the arithmetic expression need not be of the same type. If the types are different, the expression is first evaluated and then converted to the type of the variable. This conversion proceeds according to the rules indicated in the following table.

The following notation is employed in the conversion table:

- n - indicates no conversion is performed;
- fix - indicates conversion to integer with truncation;
- fl - indicates conversion from integer internal form to real internal form (which generally requires no action);
- real - indicates conversion by taking the real part of a complex item;
- rnd - indicates conversion from double precision to real with rounding;
- ext - indicates conversion from real to double precision, with the second word set to zero;
- i=0 - indicates that an imaginary part of zero is supplied.

Type of v	Type of Expression			
	Integer	Real	Double Precision	Complex
Integer	n	fix	rnd, fix	real, fix
Real	fl	n	rnd	real
Double Precision	fl, ext	ext	n	real, ext
Complex	fl, i=0	i=0	rnd, i=0	n

The internal storage formats of the various data types are discussed in chapter 11.

If a real value is to be converted to integer and the value lies outside the allowable range for integer constants, execution will be terminated.

Because the system hardware automatically handles mixed-mode arithmetic, it is unimportant whether a value is stored in integer or floating-point format. The preceding table represents the situation when the parameter compiler option OPT is assigned the value –1 (see chapter 20). When this option has the default value of 0, program values are stored in integer format when possible as a code optimization feature. Thus, relatively small floating-point values with a zero fraction part are stored as integers.

Examples:

```
G(IROW+2,JCOL)=IROW-N
A=J+12+B/3.6
C=6.2+(1,2.3)
```

LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is represented by form 1 of the assignment statement, with v being a logical variable and e being a logical expression.

When such a statement is executed, the logical expression is evaluated and the logical value obtained is placed into the storage word allocated to the variable v.

Examples:

```
L(2,4)=L(1,1).AND.L(1,2)
D6=G.GT.H.OR.B.IS.C
L=.TRUE.
```

ASSIGN STATEMENT

An ASSIGN statement is represented by form 2 of the assignment statement. This statement assigns a statement label to an integer simple variable so that the variable may be used in an assigned GO TO statement (see chapter 8).

Example:

```
ASSIGN 999 TO LABEL
GO TO LABEL, (1,2,999)
```

When the GO TO statement is executed, control will be transferred to the executable statement bearing the label 999 in that program unit.

Until it has been reassigned with an arithmetic value, an assigned variable should only be used in assigned GO TO statements since unpredictable results could occur.

10. DATA STATEMENTS

The non-executable DATA statement allows compile-time initialization of variables.

The proper format for the DATA statement is:

DATA k/d/ ...
where k is a variable list (see text), d is an initial value list (see text), and the ellipsis indicates as many repetitions of the form k/d/ as desired, with each repetition optionally preceded by a comma.

STATEMENT USE

A DATA statement may appear at any point in a program unit with the following exception. A DATA statement may not precede any specification statement affecting a variable initialized by the DATA statement.

The DATA statement is processed at compilation time. Variables and arrays appearing in a DATA statement in a subprogram will retain their values between references to that subprogram. A DATA statement appearing in a subroutine or function subprogram may not initialize dummy arguments or elements of COMMON storage. Elements of a COMMON block may appear in DATA statements only in a BLOCK DATA subprogram or in a main program.

VARIABLE LISTS

The variable list denotes the variables and arrays that are to be initialized in the DATA statement.

A variable list may be a simple variable list, a DO-implied variable list, or two variable lists separated by a comma. A simple variable list consists of a variable or an array name. A DO-implied variable list consists of a variable list followed by a comma and a DO-implied specification, all of which is enclosed within parenthesis.

The proper format for a DO-implied specification is:

1. v = m,n,i 2. v = m,n
where v is the integer or real control variable and m, n, and i are the initial parameter, terminal parameter, and incrementation parameter, respectively, each of which is an optionally signed integer constant.

The control variable in a DO-implied variable list must appear only as a subscript expression in the variable list, and only array elements employing the control variable and integer constants as subscript expressions may appear in DO-implied variable lists.

Example of a variable list:

```
K, M, (A(J,2), B(J),J=1,10),X
```

THE INITIAL VALUE LIST

The list of values to which the elements of the variable list are to be initialized consists of a list of constants and strings separated by commas. Each element of the variable list should be initialized.

The characters "T" and "F" may be used in the initial value list to indicate the logical constants .TRUE. and .FALSE., respectively.

The constants and strings may be optionally preceded by a repeat count of the form n*, where n is an unsigned nonzero integer constant. Preceding a constant or string by n* is equivalent to listing the constant or string n times.

Values are assigned to elements of the variable list in the order in which they occur. For example, the DATA statement:

```
DATA A,B/2,3/, C,D/2*4/
```

initializes the variables A and B to the values 2 and 3, respectively, and initializes C and D to 4.

The initial value list may contain strings up to 264 EBCDIC characters long. A "primary" string is defined to be a string consisting of at most the number of characters specified by the CHARS compiler option that initializes a single precision variable, or a string consisting of at most twice the number of characters specified by CHARS that initializes a double precision variable. A "long" string is defined to be a string consisting of more than the number of characters specified by CHARS that initializes a single precision variable, or a string consisting of more than twice the number of characters specified by CHARS that initializes a double precision variable (see chapter 19 for a discussion of the CHARS compiler option).

A primary string is stored left-justified within the character area of the variable as defined by CHARS with blank fill to the right if necessary. For example, if D is a double precision variable and CHARS=6, the DATA statement:

```
DATA D/'bABCDEFG'/
```

(where b represents a blank) initializes D to the value: bABCDEFGbbbb.

Any long string initializing a simple variable or array element or preceded by a repeat count greater than 1 is truncated to a primary string, with the leftmost characters of the string retained.

A long string initializing an array name is transferred to the array, beginning with the indicated element, until either the end of the array or the end of the string is encountered. Character assignments are made to the array elements in the order in which these elements are stored internally (see chapter 11). If the string terminates before the entire array is initialized, the last initialized array element will be filled on the right with blanks, if necessary. The remaining elements of the array will be blanked if the last value in the initial value list were a string, or zeroed otherwise.

For example, this program excerpt:

```
REAL A(6)
DOUBLE PRECISION D(2)
DATA D,A/'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ,1,2,3,"ABCDEFG"/
```

will cause the following assignments:

```
D(1)='ABCDEFGHIJKL'
D(2)='MNOPQRSTUVWX'
A(1)=1
A(2)=2
A(3)=3
A(4)='ABCDEF'
A(5)='Gbbbbb'
A(6)='bbbbbb'
```

Double Precision Octal and Hexadecimal Constants

A double precision variable may be initialized by an octal constant containing 32 or fewer octal digits or by a hexadecimal constant containing 24 or fewer hexadecimal digits. Only double precision items may be initialized by these extensions to the octal and hexadecimal constants.

When 17 to 32 octal or 13 to 24 hexadecimal digits are specified in the constant, the digits beyond those necessary to specify the contents of the first word are stored right-justified in the second word with zero fill if necessary.

For example, if D is a double precision simple variable, the DATA statement:

```
DATA D/Z123456789ABCDEF/
```

places the hexadecimal value 123456789ABC000000000DEF into D.

ASSIGNMENT CONSIDERATIONS

If a simple variable which is not in COMMON storage or EQUIVALENCEd is initialized more than once, a warning will be given during compilation. The variable will be initialized to the last value encountered. If such a variable or array is in COMMON storage or is EQUIVALENCEd and is initialized more than once, the value of the item will be undefined.

Conversion During Assignment

The following table indicates the type of conversion to be performed on a constant or string appearing in an initial value list when it is assigned as the initial value of a variable.

The following notation is used in this table:

- n - no conversion, except as previously noted for strings and octal and hexadecimal constants;
- fix - truncated to integer if magnitude does not exceed maximum integer size, integer overflow error given otherwise;
- fl - convert integer to real;
- real - convert from complex to real by truncating imaginary part;
- rnd - convert from double precision to real with rounding;
- ext - extend single precision to double by adding a second word of zero;
- i=0 - supply zero imaginary part.

	Variable Type				
Constant Type	**Integer**	**Real**	**Double Precision**	**Complex**	**Logical**
Integer	n	fl	fl, ext	fi, i=0	n
Real	fix	n	ext	i=0	n
Double Precision	rnd, fix	rnd	n	rnd, i=0	i
Complex	real, fix	real	real, ext	n	i
Logical	i	i	i	i	n
String	n	n	n	n	n
Single Precision Octal or Hexadecimal	n	n	ext	i=0	n
Double Precision Octal or Hexadecimal	rnd	rnd	n	rnd, i=0	i

11. STORAGE ALLOCATION

Each B 7000/B 6000 series FORTRAN data item is allocated one or more units of storage depending upon the type of value the item represents. The unit of storage is the data word, 48 bits of which are accessible to the user.

The following FORTRAN features require storage configurations:

a. Simple Variables
b. Arrays
c. EQUIVALENCEd Data Items
d. Elements of COMMON Storage

The method of storage allocation for each of these features is discussed in this chapter.

NOTE

The notation [m:n] is used in this section to describe data word fields. The 48 accessible bits of a data word are considered to be numbered, with the left most bit being bit 47 and the right most bit being bit 0. In the notation [m:n], m denotes the number of the left most bit of the field being described, and n denotes the number of bits in the field. For example, the field shown here (bits 28 through 24) would be described by [28:5]:

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

Hexadecimal constants are used extensively in the present chapter to describe word contents. Hexidecimal constants can be visualized as columns in the preceding diagram of a data word.

SIMPLE VARIABLE

Each simple variable requires one or two words of storage depending upon the type of the variable. The simple variable types are:

a. integer simple variables
b. real simple variables
c. double precision simple variables
d. complex simple variables
e. logical simple variables

Each of these simple variable types has a unique internal storage configuration.

Integer Simple Variables

An integer simple variable requires one word of storage. The data word corresponding to an integer simple variable is partitioned as follows:

Field	Contents
[47:1]	Unused
[46:1]	Sign bit
[45:7]	Contains all 0's
[38:39]	Magnitude of data item

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word involved. This bit is 0 for positive values and 1 for negative values. The magnitude of the value is stored right-justified in bits 38 through 0 preceded by zeros.

For example, the internal representation of the integer 10 described using hexadecimal constant notation is:

```
Z00000000000A
```

The internal representation of –10 is:

```
Z40000000000A
```

The largest integer value that may be stored is Z007FFFFFFFFF = 549755813887.

Real Simple Variables

A real simple variable requires one word of storage. The data word corresponding to a real simple variable is partitioned as follows:

Field	Contents
[47:1]	Unused
[46:1]	Mantissa sign bit
[45:1]	Exponent sign bit
[44:6]	Magnitude of data item exponent
[38:39]	Magnitude of data item mantissa

Real (floating-point) values are represented internally in signed-magnitude mantissa-and-exponent notation. The sign of the mantissa is denoted by bit 46 while the sign of the exponent is denoted by bit 45 of the data word. A minus sign is denoted by a 1 in the appropriate bit. The magnitude of the exponent of the value is contained in bits 44 through 39; hence, the exponent may not exceed a magnitude of (2**6) –1 = 63. The magnitude of the mantissa of the value is contained in bits 38 through 0. This field is identical to the magnitude field of a data word containing an integer value. Thus, an integer data word may be considered to be a real data word with an exponent field containing all zeros.

The value represented by a real data word is:

```
(mantissa) * 8 ** (exponent)
```

The magnitude of the mantissa is usually stored left-normalized within its word field. Therefore, the internal representation of the real value 0.5 (or .5E0) in hexadecimal constant notation is Z26C000000000. The exponent in this example is –13 and the mantissa is 4*8**12.

Double Precision Simple Variables

A double precision simple variable requires two adjacent words of storage. The data words corresponding to a double precision simple variable are partitioned as follows:

Field		Contents
	(First Word)	
[47:1]		Unused
[46:1]		Mantissa sign bit
[45:1]		Exponent sign bit
[44:6]		Least significant part of magnitude of exponent
[38:39]		Most significant part of magnitude of mantissa
	(Second Word)	
[47:9]		Most significant part of magnitude of exponent
[38:39]		Least significant part of magnitude of mantissa

Double precision values are represented internally in signed-magnitude mantissa-and-exponent notation with a most significant part and a least significant part. The sign of the mantissa of the data item is contained in bit 46 of the first data word while the sign of the exponent is contained in bit 45 of the first word. A minus sign is denoted by a one in the appropriate bit. The magnitude of the exponent of the value is contained in a total of 15 bits. The first nine of these 15 bits (the most significant part of the magnitude of the exponent) are contained in bits 47 through 39 of the second data word. The remaining six of these 15 bits (the least significant part of the magnitude of the exponent) are contained in bits 44 through 39 of the first data word.

The magnitude of the mantissa of the value is contained in a total of 78 bits. The first 39 bits (the most significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the first data word. The remaining 39 bits (the least significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the second data word.

The value represented by a double precision data word pair may be obtained by the formula:

```
((most significant part of mantissa)
+(least significant part of mantissa) * 8 ** (-13))
*8**((least significant part of exponent)
    +((most significant part of exponent) * 2 ** 9))
```

The magnitude of the mantissa is usually stored left-normalized within the total 78-bit field.

As an example, the internal two-word representation of the double precision value 1D0 in hexadecimal constant notation is:

```
Z261000000000
Z000000000000
```

The contents of the first of these two data words are identical to the contents of a data word representing the real value 1E0. When a value which does not exceed the limits of a real simple variable is assigned to a double precision data word pair, then that data word pair can be represented in a form where the second data word is all zeros and the first data word is identical to the real representation of the value.

Complex Simple Variables

A complex simple variable requires two adjacent words of storage. The first data word contains the real part of the simple variable, and the second data word contains the imaginary part of the simple variable.

Each of these two data words conforms to the format of an integer or real simple variable. If the complex variable is assigned the value of a complex constant whose real and/or imaginary part is an integer constant, then the first and/or second data word of the complex constant will be partitioned as an integer data word. A real constant in the complex constant will similarly control the structure of the corresponding complex data word.

For example, if C is complex, the assignment statement:

```
C = (1,0.5E0)
```

places into the two data words corresponding to C these values:

```
Z000000000001
Z26C000000000
```

The first data word (the real part of C) contains a representation of the integer constant 1. The second data word (the imaginary part of C) contains a representation of the real constant 0.5E0. If the assignment statement were:

```
C = (1, 1)
```

then the contents of the data words would be:

```
Z000000000001
Z000000000001
```

Logical Simple Variables

A logical simple variable requires one word of storage. The data word corresponding to a logical simple variable is partitioned as follows:

Field	Contents
[47:47]	Unused
[0:1]	Value bit

The left most 47 accessible bits of the data word containing the value of a logical simple variable has no effect on the value, while the value represented by that variable is contained in bit 0 of the data word. When bit 0 is 1, the value of the variable is .TRUE. and when bit 0 is 0, the value of the variable is .FALSE.

Usually, the internal representation of a logical value is identical to the representation of an integer 1 or an integer 0. However, since only the state of bit 0 is taken into consideration, the remaining 47 bits may assume any states. Logical operators reference all 48 bits (see chapter 6).

ARRAYS

FORTRAN arrays are provided to allow the user to organize program storage locations into a convenient structure. Internally, an array is stored as a group of one or more contiguous storage units.

A FORTRAN array of any number of declared dimensions is represented internally by a one-dimensional array of storage locations. Each element of an array has storage requirements and partitioning identical to that of a simple variable of the same type as the array. Thus, each element of a real array requires one data word partitioned like a real simple variable and each element of a complex array requires two data words partitioned like a complex simple variable.

Each integer, real, and logical array is allocated as many data words as there are elements in the array. Each double precision and complex array is allocated twice as many internal data words as there are elements in the array.

The elements of an array are ordered into a sequence. The subscript of an array element has a subscript value which determines which element of the sequence is identified by the array element. The first element has a subscript value of 1.

If an array is declared by the array declarator

A(L1:U1, L2:U2, L3:U3, ..., Ln:Un)

the subscript value for the array element

A(S1,S2,S3,...,Sn)

would be

```
1 + (S1–L1)
  + D1*(S2–L2)
  + D1*D2*(S3–L3)
  + ...
  + D1*D2*...*D(n–1)*(Sn–Ln)
```

where

Dk is the size of the kth dimension, equal to Uk–Lk+1, where Uk and Lk are truncated to integers.

n is the number of dimensions.

Sk is the kth subscript expression, which is truncated to an integer.

All subscripts which resolve to subscript values between 1 and (D1*D2* ... *Dn) are valid. Therefore, individual subscript expressions are not compared to dimension bounds.

For an array declared by the declarator

A(6,6)

a reference to element A(10,1) is valid.

The offset of an array element within the sequence is its position in the sequence relative to the first element. The offset of an array element is equal to its subscript value minus 1. The offset of the first element of an array is zero.

Subscript expressions are truncated to an integer value before being used to calculate the subscripted value.

Example:

For an array declared

DIMENSION A(0:5,1:4)

the subscript value for A(3.73,2.79) and A(3,2) would be the same:

$$
\begin{aligned}
&= 1 + (3-0) + 6*(2-1) \\
&= 1 + 3 + 6 \\
&= 10
\end{aligned}
$$

The ordering of the array elements produced by the subscript value is identical to that obtained by listing all of the array elements by varying the first subscript expression most rapidly, the second subscript expression next most rapidly, etc.

Example:

For the array A declared by the declarator

DIMENSION A(0:2, 1:2, -2:-1)

the elements are stored in the following order:

```
A(0,1,-2)
A(1,1,-2)
A(2,1,-2)
A(0,2,-2)
A(1,2,-2)
A(2,2,-2)
A(0,1,-1)
A(1,1,-1)
A(2,1,-1)
A(0,2,-1)
A(1,2,-1)
A(2,2,-1)
```

Element A(0,1,–2) corresponds to the first element of the internal array. This order is the order in which array elements are considered when a multidimensional array is used in an EQUIVALENCE statement or when the name of a multidimensional array appears in an I/O list, DATA statement, argument list, COMMON statement, as a format designator, as a file designator, in NAMELIST output, in a DEBUG MONITOR statement, or in a DEBUG DUMP statement.

An array is stored internally in consecutive storage locations as long as the storage requirement of the array does not exceed 4095 words. Arrays exceeding this size will be segmented for overlay purposes unless the compiler option LONG is set (see chapter 19). Each segment will contain a maximum of 256 words and will automatically be overlayed and recalled as necessary during execution of the program. In no case may an array exceed 65,535 elements in length.

EQUIVALENCED DATA ITEMS

The EQUIVALENCE statement allows the user to assign a number of program data items to a single storage unit. Thus, more than one simple variable name or array element name may refer to one storage location.

Single Storage Locations

The least complicated use of the EQUIVALENCE statement involves the assignment of data items requiring a single word of storage to a mutual storage location. As an example, assume that the following statements are the first statements of an executable program.

```
IMPLICIT INTEGER (A), REAL(B), LOGICAL(L)
DIMENSION AR(2)
EQUIVALENCE (A, AR(2), B2), (AL,L)
```

The EQUIVALENCE statement causes the integer simple variable A, the integer array element AR(2), and the real variable B2 to all be assigned to the same data word. The first element of AR is not affected by this statement. A change in the value of any one of the three EQUIVALENCEd items produces a simultaneous, equivalent change in the value of the other two items.

The EQUIVALENCE statement also causes the integer simple variable AL and the logical simple variable L to be assigned to the same data word. As the simple variable L changes value, AL also changes value. For example, the assignment statement:

```
L = .TRUE.
```

places the integer value 1 into simple variable AL. Also, as the simple variable AL changes value, L changes value accordingly. As AL assumes even integer values, L assumes the value .FALSE., and as AL assumes odd integer values, L assumes the value .TRUE..

Multiple Storage Locations

EQUIVALENCE statements may also involve data items requiring more than one word of storage. As an example, assume that the following statements are the first statements of a program.

```
DOUBLE PRECISION D
REAL A(2)
EQUIVALENCE (A, D, B)
```

The EQUIVALENCE statement causes the real array A, the double precision simple variable D, and the real simple variable B to be assigned to identical data words. As A and D both require two data words, the first and second elements of A become equivalent to the first and second words, respectively, of the storage unit assigned to D. The simple variable B, requiring only one data word, will be assigned to the same word as A(1) and the first word of D.

Example, where CHARS=6:

```
      DOUBLE PRECISION STRING
      DIMENSION PART(2)
      EQUIVALENCE(STRING, PART)
      PART(1) = 'ABCDEF'
      PART(2) = 'GHIJKL'
      WRITE (6,6) STRING
6     FORMAT(X,A12)
      END
```

This program produces the following output:

```
ABCDEFGHIJKL
```

Array Handling

The EQUIVALENCE statement may be used to assign a group of contiguous storage locations to a number of program arrays. There are three types of possible array references in an EQUIVALENCE statement:

a. An array name.
b. A subscripted array name with the same number of subscript expressions as the array has dimensions.
c. A subscripted array name with one subscript expression where the array has more than one dimension.

Assume that the following statements are the first statements of a program unit.

```
REAL A(4), B(10, 10), C(100), D(50), E(3, 3), F(50)
DOUBLE PRECISION DE(2) / 2* 1D0/
EQUIVALENCE (A, DE), (B(1,1),C(1)),(D,F(26),E(2))
```

The first list in this EQUIVALENCE statement causes the real array A and the double precision array DE to share four storage words. The first two elements of A become equivalent to the two words of the first element of DE, and the last two elements of A becom equivalent to the two words of the second element of DE. The initial value list in the type statement declaring DE causes the double precision value 1D0 to be placed in DE(1) and DE(2) and in the word pairs A(1), A(2) and A(3), A(4).

The second list in the EQUIVALENCE statement causes the real arrays B and C to share 100 storage words. Each element in the 100-element array C, beginning with C(1), is assigned to the same storage unit as a unique element of B. The elements of the two-dimensional array B are stored internally in the manner described in the discussion of "ARRAYS" in this chapter. The internal storage units assigned to C occur in the same order as the elements of that array. Hence, each C(I) is equivalenced to the Ith internal element of B and it follows that if the following two PRINT statements occur in this program unit, identical output will be produced:

```
PRINT 1,(C(I), I=1,100)
PRINT 1,B
```

The third list in the sample EQUIVALENCE statement indicates that the elements of the arrays D, F, and E are to be equivalenced in such a manner that D(1), F(26), and the second internal element of the two-dimensional array E are to be assigned identical internal storage units. Therefore, the elements F(26) through F(50) become equivalent to the first 25 elements of D. Equivalencing is handled in the manner illustrated below. Each of the lines denotes a single storage location, and the array elements on a line are assigned to the same storage location.

	F(1) through F(25)	E(1,1)
D(1)	F(26)	E(2,1)
D(2)	F(27)	E(3,1)
D(3)	F(28)	E(1,2)
D(4)	F(29)	E(2,2)
D(5)	F(30)	E(3,2)
D(6)	F(31)	E(1,3)
D(7)	F(32)	E(2,3)
D(8)	F(33)	E(3,3)
D(9) through D(25)	F(34) through F(50)	
D(26) through D(50)		

The following DATA statement would initialize the elements E(1,2), E(2,2), F(28), F(29), D(3), and D(4) with the value 6:

```
DATA (E(J,2), J=1,2) / 2*6/
```

ELEMENTS OF COMMON STORAGE

The COMMON statement allows program values to be communicated between program units without employing argument lists and permits these data items to be referenced by the same or different symbolic names in each program unit. This statement also allows data initialization via a BLOCK DATA program unit (see chapter 15).

Each element of a common block is allocated storage in common storage once for an executable program. Each program unit may reference a common block (and hence each location in the block) through an appropriate COMMON statement.

The size of each block of common storage is either as large as the maximum size indicated by a COMMON statement referencing that block in any program unit or as large as the maximum length to which the block is extended by an EQUIVALENCE statement. In no event may a common block exceed 65,535 data words in length.

Assume that the following statements are the first statements of a program unit.

```
SUBROUTINE MSG
DOUBLE PRECISION D
LOGICAL FLAG(6)
COMMON WORD1,WORD2,D,FLAG,TEXT(20)
COUNT=1
```

The COMMON statement will be assumed to provide the maximum size of the blank common block in the executable program containing program unit MSG. The total size of this common block is then 30 words. These words are denoted in the MSG subprogram as the words assigned to the real variables WORD1 and WORD2, the word pair assigned to the double precision variable D (which begins as required at an odd-numbered word within the block), the six words assigned to the logical array FLAG, and the twenty words assigned to the real array TEXT. These data words are contained within the common block in the order listed.

The blank common block just described could be referenced, for example, by a COMMON statement within another program unit:

```
     SUBROUTINE DUMP
     COMMON T(10)
     PRINT 1,T
1    FORMAT(X,Z12)
     RETURN
     END
```

The elements of real array T would therefor correspond to the first ten data words contained in the common block. The last 20 data words of the common block are not accessed in the DUMP subprogram.

Additionally, the EQUIVALENCE statement may interact with the COMMON statement to extend the length of a common block beyond the maximum size specified for the block by COMMON statements.

As an example, assume that the following statements form two units of an executable program.

```
FUNCTION SUM(N)
COMMON /GR1/IT(3,3)
   .
   .
   .
RETURN
END

LOGICAL FUNCTION TEST(L)
LOGICAL X(6)
COMMON /GR1/K(9)
EQUIVALENCE (K(6),X)
   .
   .
   .
RETURN
END
```

The common block referenced by these two program units is named GR1 and will consist of 11 data words. The function SUM accesses the first nine words of this block through the two-dimensional integer array IT. The function TEST accesses the first nine words of the block through the integer array K. In addition, the next two locations of GR1 are referenced as the logical array elements X(5) and X(6) since X is equivalenced to K starting at the element K(6).

Array K and array IT both refer to the same area of common storage. Use of array K allows these locations to be referenced using only one subscript expression. Equivalencing portions of the X and K arrays allow some elements of GR1 to be handled as both integer and logical type items.

The common block could not have been extended backwards by the EQUIVALENCE statement. The following combination of statements is invalid:

```
LOGICAL X(6)
COMMON/GR1/K(9)
EQUIVALENCE (X(3),K)
```

12. INPUT/OUTPUT STATEMENTS

The executable Input/Output (I/O) statements handle communication of data between internal and external storage.

PROGRAM FILES AND FILE DESIGNATORS

A file is a collection of records. Files are contained on peripheral hardware devices such as disks, disk packs, magnetic tapes, line printers, card readers, card punches, or remote terminals.

The records contained in a file are each an integer number of data words in length and exist as a totally ordered set. The N records of a file are logically numbered in order from 1 through N; record 1 immediately follows the initial point of the file, while record N immediately precedes the endfile mark terminating the file.

A file designator is a means of referring to a file. No file designator may refer to more than one file at a time. A file designator on this system is an arithmetic expression with a value from 1 to 99. If the arithmetic expression is not of type integer, it will be truncated to an integer.

Each file is recognized by the system by two names: the internal name and the external name or the value of the TITLE attribute of the file (see Input/Output Subsystem, Form No. 5001779). The form of an internal file name is:

```
FILEn
```

where n is the file designator for the file with no leading zeros. The internal file names corresponding to the file designators 6 and 99 are FILE6 and FILE99, respectively.

The TITLE of a file consists of one to fourteen identifiers separated by slashes (/); each identifier is one to seventeen characters long. When the TITLE of a program file is unspecified, a default TITLE is assigned to the file which is identical to the internal file name. The following are examples of valid file TITLEs:

```
A/B
FILE7
PRINTER
DATA-FILE/ONE/FINAL
```

Care must be taken when employing file designators which are not constants. Such file designators are called variable file designators. A file which is referenced by a variable file designator must be either declared in a file declaration, not referenced explicitly as a constant file designator in another I/O statement, or not referenced implicitly by a READ, PRINT, or PUNCH statement. If not, the program may get a syntax error, or execution of the program will be terminated by the system and the message:

```
UNDEFINED VARIABLE FILE
```

will be displayed at the Operator Display Terminal (or the user's station if initiated through CANDE).

Each file designator has a default device type associated with it. File designators 5, 6, and 7 are assigned card reader, line printer, and card punch files, respectively, by default. Other designators are assigned tape files by default.

The attributes of a file may be specified by the user through file declarations. These declarations are described in appendix B. The FORTRAN compiler automatically assigns values to file attributes when a constant file designator is encountered in I/O statements in the source. The user need never employ file declarations if the default file specifications are satisfactory and consistent with the intended use of the file.

I/O LISTS

I/O lists are used in READ, WRITE, PRINT, and PUNCH statements to denote the program storage locations to or from which data is to be transferred.

An I/O list consists of one or more simple list elements or DO-implied I/O lists separated by commas.

A simple list element is a variable name, an array name, or an array element. The following are examples of valid I/O lists:

```
A, B(3)
XARRAY, XARRAY (2,4)
X(1+N), G
RESULT
```

A simple list element may also be an arithmetic expression or a logical expression when used in an output list. The following are examples of valid output lists:

```
N, 1+COS(N)
A.OR.B, X**3
```

A DO-implied I/O list is an I/O list immediately followed by a comma and a DO-implied specification, all of which is enclosed within parentheses.

The proper format for a DO-implied specification is:

1. v = m,n,i 2. v = m,n
where v is the integer real control variable, and m,n and i are the initial parameter, terminal parameter, and incrementation parameter respectively, each of which is an arithmetic expression.

The control variable need not appear in the I/O list. The following are examples of DO-implied I/O lists:

```
(A(J) , B , J , C(J,J) , J = 1,10,2)
(IPLOT , COUNT = 1,60)
(X, (Y(I,J), I=1,10), Z, J=1,2)
```

An I/O list may contain grouping parentheses. The following are examples of valid I/O lists:

```
A, (J,GX(J), B, J=3,4),10,X
XARRAY(3), XARRAY,(XARRAY(EL), EL = 1,20)
RESULT
(A(I), I=1,20) , (J(I), I=1,5)
(((Z(I,J,K),K=1,10),J=1,10),I=1,10),7,G
```

Transfer of data to and from storage locations occurs in the order these locations are named in the I/O list, from left to right. Items named in a DO-implied I/O list are referenced repeatedly until the implied DO is satisfied. The appearance in an I/O list of an array element specifies only that array element, while the appearance of an array name specifies every element declared for the array in that program unit. These array elements are transferred in the order in which they are stored internally (see discussion of array storage in chapter 11).

If an arithmetic or logical expression appearing in an I/O list invokes a function, that function must not cause execution of an I/O statement using the same file as the file employed by the statement being executed. Thus, if F is a function and this I/O statement is executed:

```
WRITE (6,1) A, F (20,X)
```

then F must not contain an I/O statement employing FILE6.

RESULT CLAUSE LISTS

Result clauses provide error handling for READ and WRITE statements. A result clause list consists of one or more result clauses separated by commas.

A result clause is either an action specifier or a result descriptor assignment.

Action Specifier

The proper format for an action specifier is:

1. DATA = s 2. END = s 3. ERR = s
where s is the label of an executable statement appearing in the same program unit as the action specifier.

If the DATA action specifier appears in a formatted READ statement, then control will pass automatically to the statement bearing the label s when the data read does not agree with the requirements specified by the format statement. If the DATA action specifier appears in an unformatted READ or WRITE statement, then control will pass automatically to the statement labeled s if the number of data words indicated by the I/O list exceeds the size of the accessed record. If these conditions occur and the DATA action specifier is missing, then the execution of the program will be terminated unless a result descriptor assignment is present.

If the END action specifier appears in a READ or WRITE statement, then control will automatically pass to the statement labeled s when an end-of-file condition occurs. End-of-file conditions occur when the following actions are attempted by a READ or WRITE statement.

Statement	Type of File	Action Causing End-of-File Condition
READ	CARD	Attempting to read a card with an invalid character in column 1.
READ	TAPE	Attempting to read an endfile record.
READ	DISK	Attempting to read a record beyond the last record of a file.
WRITE	DISK	Attempting to write beyond the end of the designated area of the disk file.
WRITE	LINE PRINTER	Attempting to write a record at end-of-page. The PAGESIZE file attribute must be assigned an appropriate value on a FILE system control card or file declaration (see appendix B).

If the conditions occur and the END action specifier is missing, then execution of the program will be terminated unless a result descriptor assignment is present.

If the ERR action specifier appears in a READ or WRITE statement, then control will automatically pass to the statement labeled s when a parity error occurs during the data transfer. If a parity error occurs and the ERR action specifier is missing, then execution of the program will be terminated unless a result descriptor assignment is present.

Examples

```
DATA = 12
END = 20006
ERR = 99
```

Result Descriptor Assignment

The proper format for a result descriptor assignment is:

RESULT = v
where v is a simple variable name or array element name.

A FORTRAN program may obtain information regarding the result of an I/O operation from the result descriptor word returned by the system I/O routines. The result descriptor assignment allows a variable or array element to be named as the recipient of this result descriptor. The result word returned contains the following information. The notation used here to describe data word fields is discussed in chapter 11.

Field	**Contents**
[0:1]	A 1 indicates an error has occurred during the I/O operation and that the field [16:16] is non zero. A 0 indicates a successful I/O operation.
[4:1]	A 1 indicates a data error has occurred.
[7:1]	A 1 indicates a parity error has occurred.
[9:1]	A 1 indicates an EOF or end-of-page has occurred.
[15:1]	A 1 indicates a time-out has occurred.
[16:1]	A 1 indicates a file security violation has occurred.
[24:8]	The number of characters in the last word of the record processed if unblocked.
[47:20]	Actual data size of the logical record in INTMODE units or words as controlled by the UNITS file attribute.

If the simple variable or array element specified in the result descriptor assignment is of type logical, then that variable will be set to .TRUE. if an I/O error has occurred.

If an I/O error condition occurs and no action specifier is present, the presence of a result descriptor assignment prevents termination of the program. Execution will continue with the next executable statement.

The following is an example of a result descriptor assignment:

```
RESULT = A(1,2)
```

Examples

```
RESULT = RSLT(2)
END = 99, ERR = 601
DATA = 100, RESULT = A
```

I/O STATEMENT SYNTAX

FORTRAN I/O statements may be grouped in the following manner:

a. Input statement (READ statement)
b. Output statements (WRITE, PRINT, and PUNCH statements)
c. Auxiliary I/O statements (BACKSPACE, CHANGE, CLOSE, ENDFILE, FIND, INQUIRE, LOCK, OPEN, PURGE, and REWIND statements)

READ Statement

The executable READ statement allows data to be input from peripheral storage or an array into the internal storage locations indicated by the I/O list or the format specified in the statement. The transferred data may be converted during the transfer process as indicated by a format specification item, or read without editing. Provisions may be made to handle errors incurred during the read.

The proper formats for the READ statement are:

```
 1.  READ f,m
 2.  READ (n,f)m
 3.  READ (n=r,f)m
 4.  READ (n,f,c)m
 5.  READ (n=r,f,c)m
 6.  READ f
 7.  READ (n,f)
 8.  READ (n=r,f)
 9.  READ (n,f,c)
10.  READ (n=r,f,c)
11.  READ (n)m
12.  READ (n=r)m
13.  READ (n,c)m
14.  READ (n=r,c)m
15.  READ (n)
16.  READ (n=r)
17.  READ (n,c)
18.  READ (n=r,c)
19.  READ (a,f)m
20.  READ (a,f,c)m
21.  READ (a,f)
22.  READ (a,f,c)
23.  READ (a)m
24.  READ (a,c)m
```

where n is a file designator; r is an arithmetic expression, the truncated value of which is a record number; f is either the label of a FORMAT statement in the same program unit, the name of an array containing format specifications, a namelist name (in forms 6-10), or a slash (/) for free-format input (except forms 6-10); c is a result clause list; m is an I/O list; and a is an array name.

File Referenced

Execution of a READ statement, forms 1 through 18, causes data to be read from one or more records of a file. The file referenced is either indicated implicitly through use of READ statement forms 1 or 6, or explicitly in the other forms by the file designator, n. READ statement forms 1 and 6 will reference the same file as a READ statement using a file designator having the value (when truncated) of 5 unless a file declaration card with the designator "READER" or a READER compiler control statement applies to the present program unit (see appendix B and chapter 19). If such a file declaration or compiler control statement appears, READ statement forms 1 and 6 will reference the file declared.

NOTE

If the file designator is that of a port file, only formatted statements are allowed. Violations of this rule result in program termination.

Record Numbers

A record number may optionally follow the file designator as shown in READ statement forms 3, 5, 8, 10, 12, 14, 16, and 18. The record number must have a value from 1 to the number of records in the designated file, and the "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote (for BCL-or BCD-coded source input). The presence of a record number causes the file to be positioned at the specified record before the read is performed. Otherwise, the file position is unchanged prior to the read. After the READ statement is executed, the file is positioned at the record immediately after the last record read. A nonpositive record number references the first record.

Use of Buffers

The intermediate storage between a file and internal storage consists of the file buffers. The size of each file buffer is equal to the number of words in each file block. The size of a file block and the number of buffers associated with a file are file attributes which may be specified.

When a READ statement is executed, the file buffers are checked to determine if any buffer contains the record to be read. If the record is found in a buffer, the data is accessed from the buffer and the file is not affected. If the record is not found in a buffer, then a read on the file block containing the record is initiated, placing the block in a buffer from which the data is then transferred to internal program storage.

Formatted READ Statement

A formatted READ statement is denoted by forms 1 through 10 an 19 through 22 of the READ statement. Such statements reference a FORMAT statement, an array containing format specifications, a namelist, or free-formatting. When a format is associated with a READ statement, the data to be transferred is scanned and edited according to the designated format specifications. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. The namelist I/O is discussed in chapter 14. Free-format is discussed later in this chapter.

The following are examples of valid formatted READ statements:

```
READ (5,1) AB
READ (FILEN, 75) (A(J), J=3,10)
READ 12
READ (23, 12345,DATA=12400) A,B,C
READ (23=12,AF)
READ 5,ARRAY
READ (5,ARRAY)
READ (5,B,RESULT=X,ERR=2)Y,Z(6),(A(J),J=1,6)
READ (I+2=IR-1,NMLST)
READ /,A,B
```

Unformatted READ Statement

An unformatted READ statement is denoted by forms 11 through 18, 23, and 24. No editing of transferred data occurs when an unformatted READ statement is executed.

Execution of an unformatted READ statement causes the input of one logical record from the designated file. If an I/O list is specified as part of the statement, data will be transferred to the specified locations from the record. Transfer occurs as full 48-bit words, and the records accessed should have been generated by unformatted WRITE statements. If no I/O list is specified on an unformatted READ, one record will be skipped in the indicated file (the file position will be advanced one record).

The following is an example of a program employing the unformatted READ statement.

```
FILE  4(TITLE="TEMP",KIND=DISK)
      .
      .
      .
      DO 1 J=1,10
      .
10    WRITE (4)A,B,C
      .
      .
      REWIND 4
      DO 60 J=1,10
      READ(4)X,Y,Z
      .
      .
60    CONTINUE
99    STOP
      END
```

Internal Data Transfer

If an array name is specified as the file designator in a READ statement, the character string contained in that array is read when the statement is executed. For example, the statements

```
  DIMENSION PACKED(5), UNPAKD(30)
  READ(PACKED,5)UNPAKD
5 FORMAT (30C1)
```

take characters, stored six characters per word in the array PACKED, and places them one character per word, right-justified with zero fill, into the array UNPAKD. Free format input is not allowed when performing internal data transfer.

WRITE Statement

The executable WRITE statement allows data to be output to peripheral storage or an array from the internal storage locations indicated by the I/O list or the format specified in the statement. The transferred data may be converted during the transfer process as indicated by a format specification, or written without editing. Provision may be made to handle errors incurred during the write.

The proper formats for the WRITE statement are:

```
 1.  WRITE (n,f)m
 2.  WRITE (n=r,f)m
 3.  WRITE (n,f,c)m
 4.  WRITE (n=r,f,c)m
 5.  WRITE (n,f)
 6.  WRITE (n=r,f)
 7.  WRITE (n,f,c)
 8.  WRITE (n=r,f,c)
 9.  WRITE (n)m
10.  WRITE (n=r)m
11.  WRITE (n,c)m
12.  WRITE (n=r,c)m
13.  WRITE (a,f)m
14.  WRITE (a,f,c)m
15.  WRITE (a,f)
16.  WRITE (a,f,c)
17.  WRITE (a)m
18.  WRITE (a,c)m
```

where n is a file designator; r is an arithmetic expression, the truncated value of which is a record number; f is either the label of a FORMAT statement in same program unit, the name of an array containing format specifications, a NAMELIST name (in forms 5-8), or a free-format designator (in forms 1-4 and 13-16); c is a result clause list; m is an I/O list; and a is an array name.

File Referenced

Execution of a WRITE statement causes data to be written from internal storage to one or more records of a file or an array. The file referenced is indicated by the file designator, n.

NOTE

If the file designator is that of a port file, only formatted statements are allowed. Violations of this rule result in program termination.

Record Numbers

A record number may optionally follow the file designator as shown in WRITE statement forms 2, 4, 6, 8, 10, and 12. The record number must have a value from 1 to the number of records in the designated file. The "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote mark (for BCL-or BCD-coded source input). The presence of a record number causes the file to be positioned at the specified record before the write is performed. Otherwise, the file position is unchanged prior to the write. After the WRITE statement is executed, the file is positioned at the record immediately following the last record written. A nonpositive record number references the first record.

Formatted WRITE Statement

A formatted WRITE statement is denoted by forms 1 through 8 and 13 through 16 of the WRITE statement. Such statements reference a FORMAT statement, an array containing format specifications, a namelist, or free-formatting. When a format is associated with a WRITE statement, the data to be transferred is converted and positioned within the record according to the designated format specifications. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. Namelist I/O is discussed in chapter 14. Free-format is discussed later in this chapter.

The following are examples of valid formatted WRITE statements:

```
WRITE(6,/) A,B
WRITE(6,1) AB
WRITE (FILEN,75) (A(J),J=3,10)
WRITE(23+J,12345,ERR=12400) 6,A,4,B,3,C
WRITE(23=12,AF)
WRITE(7,ARRAY)
WRITE(6,B,RESULT=X,ERR=2)Y,Z(6),(A(J),J=1,6)
WRITE (J+2 = (JR*2)+1,NMLST )
```

Unformatted WRITE Statement

An unformatted WRITE statement is denoted by forms 9 through 12, 17, and 18. No editing of transferred data is associated with these forms of the WRITE statement.

Execution of an unformatted WRITE statement causes one logical record to be output to the indicated file. The I/O list denotes the sequence of values to be contained in the record. The contents of the indicated storage locations are placed unchanged in the record as full 48-bit words intended to be read by unformatted READ statements. The I/O list may not be omitted on this type of WRITE statement. An unformatted WRITE statement may generate more than one physical record only if the file involved is a LINKWORD file (FILETYPE=6).

Internal Data Transfer

If an array name is specified as the file designator in a WRITE statement, the character string produced as the edited output is placed into the array. For example, the statements

```
          DIMENSION PACKED(5),UNPAKD(30)
          WRITE(PACKED,6) UNPAKD
      6   FORMAT(30C1)
```

take characters one character per word from the rightmost character position of each word of the array UNPAKD, and store them six characters per word in the array PACKED.

PRINT Statement

The executable PRINT statement allows data to be output to peripheral storage from the internal storage locations indicated by the I/O list or the format specified in the statement. This statement is a variation of the formatted WRITE statement; the file referenced is not explicitly specified, and no result clause list is allowed.

The proper formats for the PRINT statement are:

1. PRINT f 2. PRINT f,m
where f is either the label of a FORMAT statement in this program unit, the name of an array containing format specifications, a namelist name (in form 1), or a free-format designator (in form 2); and m is an I/O list.

File Referenced

Execution of PRINT statement causes data to be written from internal storage to one or more records of a file. Every PRINT statement appearing within an executable program references the same file. The file referenced is the same file that would be referenced by a file designator with a value of 6, unless a file declaration with the designator "PRINTER" or "PRINT" or a PRINTER compiler control statement applies to the present program unit (see appendix B and chapter 19). If such a file declaration or compiler control statement appears, the PRINT statement will reference the declared file.

The following is an example of a program using the PRINT statement.

```
FILE  6(TITLE="PRINTOUT/SINE")
      PRINT 1,(SIN(J),J=1,10)
1     FORMAT(F7.3)
      STOP
      END
```

This program generates output on a line printer file with the TITLE "PRINTOUT/SINE".

Record Access

The position of the file is unchanged prior to the execution of the PRINT statement. After execution of a PRINT statement, the file is positioned at the record immediately after the record(s) written.

Formatting

The format designator in a PRINT statement is either the label of a FORMAT statement, the name of an array containing format specifications, a namelist name, or a free-format designator. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. Namelist I/O is discussed in chapter 14. Free format is discussed later in this chapter.

PUNCH Statement

The executable PUNCH statement allows data to be output to peripheral storage from the internal storage locations indicated by the I/O list or the format specified in the statement. This statement is a variation of the formatted WRITE statement; the file referenced is not explicitly specified, and no result clause list is allowed.

The proper formats for the PUNCH statement are:

```
1.  PUNCH f
2.  PUNCH f,m
```

where f is either the label of a FORMAT statement in this program unit, the name of an array containing format specifications, a namelist name (in form 1), or a free-format designator (in form 2); and m is an I/O list.

Execution of a PUNCH statement causes data to be written from internal storage to one or more records of a file. Every PUNCH statement appearing within an executable program references the same file. The file referenced is the same file that would be referenced by a file designator with a value of 7, unless a file declaration with the designator "PUNCH" applies to the present program unit (see appendix B). If such a file declaration appears, the PRINT statement will reference the declared file.

The following are examples of valid PUNCH statements:

```
PUNCH NMLST
PUNCH 1,(A(I),I=2,7)
PUNCH ARRAYF, ARRAYX
PUNCH 12, I+2
PUNCH /,G,R2
```

BACKSPACE Statement

The executable BACKSPACE statement is an auxiliary I/O statement which allows a program file to be repositioned.

The proper format for the BACKSPACE statement is:

BACKSPACE n
where n is an arithmetic expression, the truncated value of which is a file designator.

When a BACKSPACE statement is executed, the indicated file is positioned one record before its present position, unless the file is presently positioned at its initial point, in which case no action is initiated.

Example:

```
FILE 4(TITLE="SOURCE/FILE4",KIND=DISK,MAXRECSIZE=4)
     READ(4,1) A
     PRINT 2,A
     BACKSPACE 4
     READ(4,2) A
     PRINT 2,A
1    FORMAT(A6)
2    FORMAT(2X,A6)
     STOP
     END
```

This program causes record 1 of the disk file "SOURCE/FILE4" to be read into the simple variable A using two different formats and to be printed out each time. The BACKSPACE statement cancels the change of file position caused by the first READ statement.

The BACKSPACE statement may not be used with the READER, PRINTER, or PUNCH file if such a file exists since this statement must employ a file designator. The BACKSPACE statement may not reference line printer, card punch, or remote files.

CHANGE Statement

The executable CHANGE statement is an auxiliary I/O statement which modifies the attributes of a file.

The proper format for the CHANGE statement is:

CHANGE (n,t)
where n is a file designator and t is a list of file attribute assignment clauses seperated by commas.

A file attribute assignment clause is of the form fa = v, where fa is a valid file attribute identifier as listed in Input/Output Subsystem, Form number 5001779, and v is either the VALUE intrinsic applied to an attribute mnemonic or an expression whose type and value are valid for that attribute. Refer to the I/O Subsystem manual for restrictions on when each attribute may be changed.

The CHANGE statement will cause the listed attributes to be assigned the given values for the indicated file if none of the applicable restrictions have been violated.

Any pointer-valued attribute, such as TITLE, must be set using an array. The array must contain left-justified EBCDIC characters, 6 characters per word, and the string of characters must be terminated with a period. No compiler translation of character strings will be performed, even if compiler option BCL, BCD, ASCII, B5500 or B5700 is set or the characters-per-word differs from 6 (through the CHARS compiler option).

NOTES

A warning is issued for INTMODE=BCL in CHANGE statements.

The BCD, B5500 and B5700 options will be deimplemented on the 3.3 release. A warning message is issued to this effect.

Example:

```
REAL A(2)/"MASTER/FILE."/
CHANGE (7,TITLE=A)
```

All logical-valued attributes must be explicitly set to a logical value; for example,

```
CHANGE (7,OPTIONAL=.TRUE.)
```

CLOSE Statement

The executable CLOSE statement is an auxiliary I/O statement which closes a file.

The proper formats for the CLOSE statement are:

1. CLOSE n 2. CLOSE (n) 3. CLOSE (n,DISP=DELETE) 4. CLOSE (n,DISP=KEEP) 5. CLOSE (n,DISP=CRUNCH)
where n is a file designator.

The execution of a CLOSE statement causes the designated file to be returned to system control. The program no longer has access to the file.

Because a CLOSE on a port file can have only one meaning, any other options specified in the CLOSE statement are ignored for port files.

If the file is a disk file generated by this program, the file will be closed and saved if the fourth form is used. If the fifth form is used, the file is closed and saved and the unused portion of the last row of disk space (beyond the end-of-file indicator) is returned to the system. Use of the other forms will cause the file to be immediately purged from the disk directory. A disk file created by the program will not remain on disk after execution of the program has terminated unless the file is closed by a form 4 or 5 CLOSE statement.

If the file is a card punch file, a card containing an endfile label is punched and the card punch device is released to the system.

If the file is a line printer file, the printer is skipped to channel 1, an ending label is printed, the printer is again skipped to channel 1, and the printer is released to the system.

If the file is a tape file and the third form is used, the file is closed and the tape is released to the system for other use. The other forms of the CLOSE statement cause a tape mark (and an ending label for labeled tape) to be written after the last block on the tape (output tapes only). In any case, the tape is released to the system.

The first two forms of the CLOSE statement are equivalent. The third form is equivalent to the PURGE statement, and the fourth form is equivalent to the LOCK statement.

Examples:

```
CLOSE TAPENO
CLOSE 6
CLOSE (15, DISP=KEEP)
```

ENDFILE Statement

The executable ENDFILE statement is an auxiliary I/O statement which writes an endfile record on an output file and closes the file.

The proper format for the ENDFILE statement is:

ENDFILE n
where n is a file designator.

The ENDFILE statement is intended primarily for use with tape files. When an ENDFILE statement follows an output statement on the file, the file is closed, system labels are written if LABELTYPE=STANDARD, and the tape is positioned such that the next record written will follow the file just created. When an ENDFILE statement follows a READ on the file, the tape is positioned at the start of the next file on the tape. When an ENDFILE statement follows a REWIND statement or another ENDFILE statement designating the same file, the ENDFILE statement is ignored.

When a program is creating multiple tape files on one physical reel, then each file on this tape must be closed by an ENDFILE statement before the next file may be written on the tape.

An ENDFILE statement executed on a file other than a tape file causes the file to be closed and rewound. The file is not released to the system by this statement.

The ENDFILE statement when used on a disk or pack file will set the EOF pointer for that file. Once the EOF pointer has been set for a disk or pack file by an ENDFILE statement, it cannot be set to a smaller value by another ENDFILE statement.

The following is an example of a program employing the ENDFILE statement.

```
FILE 4(TITLE="A60/B",KIND=TAPE9, MAXRECSIZE=20)
FILE 5(TITLE="A60/C",KIND=TAPE9, MAXRECSIZE=30)
     TAPENO=4
     WRITE(TAPENO,2,ERR=12)  FCNA(TIME(1))
     .
     .
     .
     ENDFILE TAPENO
     TAPENO=TAPENO + 1
     WRITE(TAPENO,36,DATA=70) G, R1, R2
     .
     .
     .
     STOP
     END
```

FIND Statement

The executable FIND statement is an auxiliary I/O statement which positions the designated file at the specified record and loads that record into an available buffer in anticipation of a read or write on that record.

The proper format for the FIND statement is:

FIND(n = r)
where n is a file designator and r is an arithmetic expression the truncated value of which is a record number.

The record number must have a value from 1 to the number of records in the designated file. The "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote (for BCL-or BCD-coded source input). A nonpositive record number references the first record.

If, when a FIND statement is executed, the indicated record is not already in any of the program buffers, then a read on the file block containing the appropriate record is initiated. The action initiated by the FIND statement may be carried out while the program performs further computations prior to the input or output statement. This conserves execution time in many cases since the input or output statement must otherwise initiate this record access itself if it finds that no buffer contains the desired record.

Error Handling

An error occuring during execution of a FIND statement will not cause the program to be terminated at that time. When a subsequent input or output statement is executed on the same file, the I/O statement will incur the I/O error.

A FIND statement may follow another FIND statement without an intermediate I/O statement. This would cause both records to be available in buffers if the file has enough buffers.

The following program is an example of the use of the FIND statement:

```
FILE 4(TITLE="A60",KIND=DISK, MAXRECSIZE=10)
     READ(4,20)A,B
     FIND(4=15)
     G=A*B**2
     READ(4,21,END=12,ERR=19) CNTRL
     .
     .
     .
     STOP
     END
```

This program causes data to be read into simple variables A and B from the first record of the disk file "A60", the 15th record of this file to be placed into a buffer while the value of the simple variable G is calculated, and a value to be read from this record into the simple variable CNTRL. Control would immediately pass to the statement labeled 19 when this latter read is attempted if an error occurred during the execution of the FIND statement.

INQUIRE Statement

The executable INQUIRE statement is an auxiliary I/O statement which may be used to determine the current status of file attributes.

The proper format for the INQUIRE statement is:

INQUIRE (n,t)
where n is a file designator and t is a list of file attribute clauses separated by commas.

A file attribute clause is of the form fa=v, where fa is a file attribute identifier as listed in Input/Output Subsystem, Form Number 5001779, and v is a variable whose type corresponds to that of the attribute.

All pointer valued attributes, such as TITLE, must be assigned to arrays into which the character strings will be transferred. The strings will be in EBCDIC characters, six characters per word, regardless of the value of compiler options B5500, B5700, BCD, BCL, ASCII, and CHARS. The string is terminated by a period. Any array used in this manner must be long enough to contain the character string it is to receive. All character positions in the array not needed to store the value of the attribute will be left unchanged.

Examples:

```
REAL KND,L(15),A(12)
LOGICAL OPN
INQUIRE (10,KIND=KND,OPEN=OPN,MAXRECSIZE=L(10),TITLE=A)
```

After execution of this INQUIRE statement, KND will contain the KIND of file 10, OPN will contain the value .TRUE. if file 10 is open or .FALSE. if file 10 is closed, L(10) will contain the maximum record size in characters or words (depending on the UNITS and EXTMODE attributes), and array A will contain the TITLE of the file.

Example:

```
FILE  70 (TITLE="MASTER/FYLE",KIND=DISK)
      REAL FNAME (3)/3*1/
      INQUIRE (70,TITLE=FNAME,KIND=KND)
```

After the execution of the INQUIRE statement, FNAME(1)="MASTER", FNAME(2) = "/FYLE.",FNAME(3)=1, and KND=VALUE (DISK).

LOCK Statement

The executable LOCK statement is an auxiliary I/O statement which closes and saves a file.

The proper format for the LOCK statement is:

LOCK n
where n is a file designator.

The LOCK statement is equivalent to CLOSE(n,DISP=KEEP).

Examples:

```
LOCK TAPENO - 1
LOCK 22
LOCK I+J
```

OPEN Statement

The executable OPEN statement is an auxiliary I/O statement which opens a file and which may assign values to attributes of the file.

The proper formats for the OPEN statement are:

```
1.  OPEN (n)
2.  OPEN (n,o,c,t)
```

where n is a file designator; o is an opentype assignment clause; c is a result clause; and t is a list of file attribute assignment clauses separated by commas.

A file attribute assign clause is of the form fa=v, where fa is a valid file attribute identifier as listed in the Input/Output Subsystem, Form No. 5001779, and v is either the VALUE intrinsic applied to an attribute mnemonic or an expression whose type and value are valid for that attribute. The listed attributes will be assigned the given values for the indicated file as it is opened. Explicit use of the file attribute OPEN within the OPEN statement causes a warning message.

An opentype assignment clause is of the form OPENTYPE=a, where a is an EBCDIC string whose value is either WAIT or AVAILABLE.

If the value WAIT is assigned, the open operation is initiated, and program execution is suspended until the open operation is complete.

If the value AVAILABLE is assigned and the file cannot be opened, the program is not suspended; otherwise, the file is opened.

If n specifies a port file and the opentype assignment clause is not used, the default is WAIT.

If the opentype assignment clause is used for a non-port file, this clause is ignored.

The open action is not required. Any READ, WRITE, PRINT, PUNCH, BACKSPACE or FIND statement will open a file if it is not already open. The file may not be already open when the OPEN statement is executed.

Any pointer-valued attribute, such as TITLE, must be set using an array. The array must contain left-justified EBCDIC characters, 6 characters per word, and the string of characters must be terminated with a period. No compiler translation of character strings will be performed, even if compiler option BCL, BCD, ASCII, B5500 or B5700 is set, or the characters-per-word differs from 6 (through the CHARS compiler option).

NOTES

A warning message is issued for INTMODE=BCL in OPEN statements.

The BCD, B5500, and B5700 options will be deimplemented on the 3.3 release. A warning message is issued to this effect.

Example:

```
REAL A(2)/"MASTER/FILE."/
OPEN (7,TITLE=A)
```

All logical-valued attributes must be explicitly set to a logical value, for example;

```
OPEN (7,OPTIONAL=.TRUE.)
```

Examples:

```
REAL A(2)/"SOME/FILE."/
OPEN (30, KIND=VALUE(DISK), TITLE=A,AREAS=2,MAXRECSIZE=10)
OPEN(9)
```

PURGE Statement

The executable PURGE statement is an auxiliary I/O statement which closes a file, purges it from the system, and releases its space for other use.

The proper format for the PURGE statement is:

PURGE n
where n is a file designator.

The PURGE statement is equivalent to CLOSE(n,DISP=DELETE).

Examples:

```
PURGE 10
PURGE FILENO
PURGE 9 +KFILE
```

REWIND Statement

The executable REWIND statement is an auxiliary I/O statement which positions a file at its initial point.

The proper format for the REWIND statement is:

REWIND n
where n is a file designator.

The REWIND statement may be executed at any point within a program to position a file at its initial point, causing the first record of the file to be the default record read or written when the next input or output statement is executed which references that file. The default record accessed by the first input or output statement in a program referencing a file is the first record of that file. Therefore, an initial REWIND statement is unnecessary.

If the file is positioned at its initial point when a REWIND statement is executed, no action is initiated.

When a REWIND statement is executed which references a file on a multi-file tape reel, the entire tape reel is rewound and positioned at beginning-of-tape.

The REWIND statement may not be used with line printer, card punch, card reader, or remote files.

The following is an example of a program employing the REWIND statement:

```
FILE  5(TITLE="A60/D",KIND=DISK,MAXRECSIZE=10)
      DO 1 J=1,100
      READ 5,A,B,C
      .
      .
      .
      CONTINUE
      FILENO=5
      .
      .
      .
      REWIND (FILENO)
      DO 601 COUNT=1,100
      .
      .
      .
601   WRITE(5,2)X,Y,Z
999   STOP
      END
```

This program reads the first 100 records of the disk file "A60/D", rewinds the file, and rewrites the first 100 records of the file.

FREE-FORMAT INPUT AND OUTPUT

The use of a free-format designator with READ, WRITE, PRINT, and PUNCH statements allows I/O to be performed with editing, but without using a FORMAT statement. The appropriate format is selected automatically, but the free-format designator gives the user some control over the editing.

The proper format for the free-format designator is:

ar/sw
where a is an asterisk (*), s is a slash (/), and r and w are integer or real arithmetic expressions enclosed in parentheses.

Input

On input, only the form consisting of a single slash (/) can be used. It allows input from records consisting of data items separated by commas.

Numeric data items can be integer, real, or double precision constants. A real or double precision constant data item may also consist of an explicit part alone. In addition, the exponent in a real or double precision data item that has a mantissa may be indicated by a plus or minus sign alone and the letter "E" or "D" may be left off.

Complex variables should be input from two single precision data items.

If a double precision variable is input from a single precision data item, the variable is assigned a double precision value.

Character strings must be enclosed in quotes.

Logical variables accept any string beginning with the letter "T" as the value .TRUE. and any string beginning with a letter other than "T" as the value .FALSE.. If a numeric data item is input to a logical variable, the variable will be assigned that numeric value.

If a blank or some character which does not make sense is found in a data item, all characters from that point until a comma or end of record is detected will be ignored.

Example:

If B is logical and C is complex and if the input record consists of (where b is a blank):

```
1b2b3b,T,2,3
```

then the statement READ/,A,B,C assigns 1.0 to A, .TRUE. to B, and (2.0,3.0) to C.

Output

On output, each value is edited into an appropriate format. An edited item is never split across a record boundary. If the record is too short to hold any reasonable representation of the item, a string of pound signs (#) is output in place of the item.

Data items are normally separated by a comma and a space. If the optional second slash (/) is used, they are separated by two spaces. Note that output produced in this manner cannot be read by a free-format input statement.

If the optional asterisk is used, the name of the data item and an equal sign (=) are output prior to the value of the data item. If the data item is not a simple variable name, then <EXP> is output as the name of the data item.

If the optional r is present, the value of the expression in parentheses, truncated to an integer if necessary, specifies the maximum number of items that will be placed in one record.

If the optional w is present, the value of the expression in parentheses, truncated to an integer if necessary, specifies the maximum width of the field into which the data item is edited.

Only the free-format designator / is available in optimized FORTRAN (see chapter 20).

Named output cannot correctly handle subscript expressions whose magnitude contains more than 7 decimal digits. The compiler will give a warning if such subscripts can be detected at compile time. The subscripts printed may be incorrect.

In the following examples, the output produced by each statement is shown after the statement.

```
PRINT/,123.456000,-.00123,1234,.TRUE.,'WOW'
123.456, -0.00123, 1234, T, WOW,

PRINT (2)//,1234.567,-2.3,4,-1.23456E9,6D7
     1234.567            -2.3
            4      -1.23456E+9
       6.0D+7

WRITE(6,//(10))-1.234567E-14,5.4321D12345,.TRUE.
-1.2346-14  5.43+12345           T

PRINT(3)/(6),-12.34,.NOT..TRUE.,'TOO LONG', '''WOW''',8.2D12345
     -12.34,       F, ######
      ''WOW'', D12346,

X=0.7
PRINT */(10),X,SIN(X)
     X=0.7, <EXP>=0.644,
```

FILE HANDLING CAPABILITIES

A B 7000/B 6000 series FORTRAN program may perform a large number of operations on program files through the available I/O statements. Files are discussed in the Input/Output Subsystem manual, Form No. 5001779. A brief description of the manner in which a FORTRAN program may manipulate files is presented here.

Line Printer Files

Line printer files may be manipulated by the WRITE, PRINT, CLOSE, ENDFILE, LOCK, OPEN, CHANGE, INQUIRE, and PURGE statements. Spacing of lines is controlled by the carriage control character contained in the leftmost character position of each output record (see chapter 13).

All printer files generated by a program are placed in backup disk files prior to their automatic printing and removal by the system if the run-time system option LPBDONLY is set. Line printer files may only be generated sequentially; that is, no record number may be used in the WRITE statement.

Card Reader Files

Card reader files may be manipulated by the READ statement and all auxiliary I/O statements except the ENDFILE statement. Card files are read by the card reader device and placed in paeudo card disk files by the system. These pseudo card files are the files which are accessed by a program, thus allowing the files to be positioned similar to disk files.

Card Punch Files

Card punch files may be manipulated by the WRITE, PUNCH, CHANGE, CLOSE, ENDFILE, IN-

QUIRE, LOCK, OPEN, and PURGE statements. The TITLE of a card punch file is punched on the initial label card of the file.

Card punch files may only be generated sequentially; that is no record number may be used in the WRITE statement.

Disk Files

Disk files may be manipulated by any I/O statement. Such files, once generated, may be used both for input and output and may be accessed serially or randomly.

More than one disk file may be open at one time.

A disk file generated by a program must be closed by the CLOSE (n,DISP=KEEP) or LOCK statement before the file will be retained in the disk directory.

Tape Files

Tape files may be manipulated by any I/O statement. Such files, once generated, may be used both for input and output (depending on the presence of a write ring), and may be accessed serially or randomly.

Tape files may be classified according to whether the file is the only file accessed from the reel on which it resides or is one of several files accessed from the same tape reel.

Single-File Tape Reels

The TITLE of a program file which is the only program file accessed from that tape may be any valid TITLE.

When an ENDFILE or REWIND statement is executed on such a tape file, the file is closed, or closed and rewound, respectively, and the tape can only be used by this file.

The tape file will be identified in system messages as "MT m", where m is the unit number of the tape device on which the reel is mounted.

Multi-File Tape Reels

One tape reel may contain more than one tape file used in the same program. Such files must have TITLEs consisting of two suitable identifiers separated by a slash, with the leftmost identifiers being identical.

Thus, the following could be TITLEs of files contained on one tape reel:

```
A60/ONE
A60/DATA
A60/CODE
```

As files are being generated on one tape reel, each file written must be closed by an ENDFILE statement before the next file may be written. The last file generated should be closed by a CLOSE or REWIND statement. When an ENDFILE statement is executed as a tape file is being written, the file is closed, system labels are written if LABELTYPE=STANDARD, and the tape is positioned at the next available space in anticipation of the generation of another tape file.

13. FORMAT SPECIFICATIONS AND STATEMENTS

This chapter discusses the format specifications provided for editing data during input and output and the method of associating these specifications with input and output statements through the FORMAT statement and arrays containing such specifications.

FORMAT SPECIFICATIONS

The following format specifications are provided:

```
 Aw
 Cw
 Dw.d
 Ew.d
 Fw.d
 Gw.d
wHs or 's' or "s"
 Iw
 Jw
 Lw
 Ow
fP
 Tt
 U or Uw or Uw.d
 Vw or Vw.d
wX or X
 Zw
```

In addition, there are two format specification modifiers:

```
K
$
```

In the list of specifications and in the following paragraphs, w and t represent unsigned nonzero integer constants denoting the width of the field in the external character string and a character position relative to the beginning of the external record, respectively. Item d represents an unsigned integer constant denoting the number of digits in the fractional part of the external string. Item f represents an integer constant scale factor. Item s represents a string of 264 or fewer characters.

The internal storage types referred to in this chapter are discussed in chapter 11.

A Format Specification

The format specification Aw transfers data to or from internal storage as EBCDIC characters.

Input Using Aw

On input, the Aw format specification causes the character string of width w in the external field to be assigned to the corresponding variable in the I/O list.

Characters are stored internally in EBCDIC, six characters per data word. Characters are placed left-justified into a data word with trailing blanks. If the field width, w, exceeds 6 (12 for double precision variables), the rightmost six (12) characters in the input field are stored into the variable and the remaining characters are ignored. The variable may be of type real, integer, or double precision.

Examples:

External String	Specification	Internal Value Real or Integer	Internal Value Double Precision
ABCD	A4	ABCDbb	ABCDbbbbbbbb
bABCDb	A6	bABCDb	bABCDbbbbbbb
ABCDEFGHI	A9	DEFGHI	ABCDEFGHIbbb
bABCDEFGHIJKb	A13	GHIJKb	ABCDEFGHIJKb

Output Using Aw

On output, the Aw format specification causes the characters contained in the variable in the I/O list to be placed into an external string of length w. If w is greater than 6 (12 for double precision variables), the 6 (12) characters contained in the variable will be placed right-justified in the field, preceded by blanks. If w is less than or equal to 6 (12 for double precision variables), the leftmost w characters in the variable will be placed in the field. If one of the character fields in the variable contains a bit pattern which does not correspond to an EBCDIC graphic, then a question mark will be placed in that position.

Examples:

Internal Value	Specification	External String
ANCbbX	A3	ABC
ABCbbX	A6	ABCbbX
ABCbbX	A9	bbbABCbbX
ABCDEFGHIJKb	A9	ABCDEFGHI
ABCDEFGHIJKb	A13	bABCDEFGHIJKb

C Format Specification

The Cw format specification has the same effect as the Aw format specification except for the following:

a. Characters are placed into and taken from the right-hand portion of the variable (rather than the left-hand portion).
b. On input the unused portion of the variable is filled with zeros (rather than blanks).

Examples of input (Z00 is represented by a ?):

External String	Specification	Internal Value Real or Integer	Internal Value Double Precision
ABCD	C4	??ABCD	????????ABCD
bABCDb	C6	bABCDb	??????bABCDb
ABCDEFGHI	C9	DEFGHI	???ABCDEFGHI
bABCDEFGHIJKb	C13	GHIJKb	ABCDEFGHIJKb

Examples of output:

Internal Value	Specification	External String
??ABCD	C4	ABCD
bABCDb	C4	BCDb
??ABCD	C9	bbb??ABCD
??bbbABCDEFG	C4	DEFG
??bbbABCDEFG	C11	?bbbABCDEFG
??bbbABCDEFG	C13	b??bbbABCDEFG

D Format Specification

The double precision format specification Dw.d transfer data to or from internal storage as double precision values.

Input Using Dw.d

On input, the Dw.d format specification assigns the value of the double precision constant written with or without exponential notation in a string of w characters to the corresponding I/O list element.

The Dw.d format specification specifies the number of decimal places in the external string by the decimal place count, d. d must be greater than zero. If no decimal point appears in the input string, a decimal point is implied d places to the left of the D or E denoting the exponent or, if there is no exponent, d places from the right edge of the input field. Thus, the external string 100D0 when read using the format specification D5.2 would be interpreted as the double precision constant 1.D+0. The field width, w, must be greater than or equal to the specified number of decimal places, d. A blank is interpreted as a zero.

The Dw.d format specification functions similarly to the Ew.d format specification with the extended ability to accept double precision constants as well as real constants.

Output Using Dw.d

On output, the Dw.d format specification writes the value of the corresponding item in the I/O list to-e an output character string of length w as a double precision constant expressed in exponential notation. The exponent is adjusted so that the decimal point is positioned as specified by the decimal place count, d. d must be greater than zero.

The value is right justified with blank fill within the field.

The field width, w, must be greater than or equal to the number of specified decimal places, d, plus 10. This provides for a 7-character exponent part, a decimal point, a zero preceding the decimal point, and a sign. If this rule is violated, the field will be filled with asterisks.

The Dw.d format specification is functionally equivalent to the Ew.d specification except for the presence of a D rather than an E in the exponent part of the output string and a five-digit rather than a two-digit exponent.

E Format Specification

The format specification Ew.d transfers data to or from internal storage as real values.

Input Using Ew.d

On input, the Ew.d format specification assigns the value of the numeric constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The EW.d format specification specifies the number of decimal places in the external string by the decimal count d. d must be greater than zero. If no decimal point appears in the input string, a decimal point is implied d places to the left of the E denoting the exponent or, if ther is no exponent, d places from the right edge of the input field. Thus, the external string 100E0 when read using the format specification E5.2 would be interpreted as the real constant 1.E+0. The field width, w, must be greater than or equal to the specified number of decimal places, d.

A blank is interpreted as a zero.

Examples:

External String	Specification	Internal Value
bbbbbb25046	E11.4	+2.5046
bbbbb25.046	E11.4	+25.046
-bb25046E-3	E11.4	-0.0025406
bb250.46E-3	E11.4	+0.25046
bbb250.46-3	E11.4	+0.25046
b-b25.04678	E11.4	-25.04678

Output Using Ew.d

On output, the Ew.d format specification writes the value of the corresponding item in the I/O list to an output character string of length w as a real constant expressed in exponential notation. The exponent is adjusted so that the decimal point is positioned as specified by the decimal place count, d. d must be greater than zero.

The value is right-justified with blank fill within the field. The field width, w, must be greater than or equal to the number of specified decimal places, d, plus 6. This provides for a 4-character exponent part, a decimal point, and a sign. If this rule is violated, the field will be filled with asterisks.

Examples:

Internal Value	Specification	External String
+36.7929	E13.5	bbb.36793E+02
-36.7929	E12.5	-b.36793E+02
-36.7929	E11.5	-.36793E+02
-36.7929	E10.5	**********

F Format Specification

The format specification Fw.d transfers data to or from internal storage as real values.

Input Using Fw.d

On input, the Fw.d format specification assigns the value of the numeric constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The Fw.d format specification specifies the number of decimal places in the external string by the decimal place count, d. If no decimal point appears in the input string, a decimal point is d places to the left of the E denoting the exponent or, if there is no exponent, d places from the right edge of the input field. Thus, the external string 1234 when read using the format specification F4.2 would be interpreted as the real constant 12.34. The field width, w, must be greater than or equal to the specified number of decimal places, d.

A blank is interpreted as a zero.

Examples:

External String	Specification	Internal Value
36725931	F8.4	+3672.5931
3.672593	F8.4	3.672593
-367259.	F8.4	-367259
-3672.E2	F8.4	-367200
367259E2	F8.4	+3672.59
3.672E-1	F8.4	+.3672
b3.672-1	F8.4	+.3672
367259	F6.6	+0.367259
b-b3456	F7.2	-34.56

Output Using Fw.d

On output, the Fw.d format specification writes the value of the corresponding item in the I/O list to an output character string of length w as a real constant expressed without using exponential notation. The decimal point is adjusted such that d digits follow the decimal point.

The value is right-justified with blank fill within the field. The specified width, w, must be greater than or equal to the number of specified decimal places, d, plus 1. This provides for a possible minus sign.

The internal value is rounded to satisfy the decimal point specification, and the field will contain asterisks if the value to be output has an integer part too large for the allotted field.

Examples:

Internal Value	Specification	External String
+36.7929	F7.3	b36.793
+36.7934	F9.3	bbb36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bbb0.00
+579.645	F6.2	579.65
+579.645	F4.2	****
-579.645	F6.2	******

G Format Specification

The format specification Gw.d is a general format specification which may be used with I/O list elements of type integer, real, or double precision. This format specification is interpreted as a F, E, or D format specification depending upon the type and the magnitude of the value of the list element. The external string will have a length of w characters.

On input, the Gw.d specification accepts real or double precision constants.

On output, the Gw.d specification produces an F, E, or D format specification representation of the value of the list element according to the following criteria. If v is the absolute value of the list item, then for k=0,1,2,..., d, if $10^{d-k-1} \leq v < 10^{d-k}$, then format F(w-4).K, 4X is used. For v<.1 or $v \geq 10^{d}$, then Ew.d is used. A scale factor affects the format specification only if Ew.d is used.

Example: The statements

```
  DO 5 K = -1,3
  V = .5*10**(2-K)
5 WRITE(6,6)V
6 FORMAT(2P,X,G10.2)
```

produce the output

```
50.00E+01
  50.
  5.0
 0.50
50.00E-03
```

H Format Specification (Strings)

The string (or Hollerith) format specification wHs, 's', or ''s'' transfer character strings to or from internal storage without employing program storage locations. The item s in this specification is a string of one to 264 characters. To represent an apostrophe within an 's' format specification, two immediately adjacent apostrophes must be used. To represent a quote in an ''s'' format specification, two immediately adjacent quotes must be used.

The field width, w, must reflect the number of characters in the string.

Input Using wHs, 's', or "s"

When used as a format specification in a READ statement, the string format specification replaces the character string s by the character string of the same length appearing on the input record. The format specification is thus modified; use of this specification for a subsequent output action will result in the new string being output.

Example:

The following program excerpt demonstrates the use of the string format specification.

```
FILE  5(TITLE="OXL/T367",KIND=DISK,MAXRECSIZE=10,BLOCKSIZE=3)
 1    FORMAT('INPUT STRING ONE  ',A6,'INPUT STRING TWO  ')
      READ 1,A
      IF(A.IS.'ORDER1') A='MASTER'
      PRINT 1,A
      .
      .
      .
```

Execution of the READ statement will cause the strings 'INPUT STRING ONE ' and 'INPUT STRING TWO ' to be replaced by the first through eighteenth and the 25th through 42nd characters, respectively, of the first record of the file "OXL/T367". These strings would be written by the PRINT statement.

Output Using wHs, 's', or "s"

When the string format specification is used for output, the character string, s, is written in the output record.

I Format Specification

The format specification Iw transfers data to or from internal storage as integer values.

Input Using Iw

On input, the Iw format specification assigns the value of the integer constant written in the input field of w characters to the corresponding I/O list element. Blanks are interpreted as zeros.

Examples:

External String	Specification	Internal Value
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
b-bb234	I7	-234

Output Using Iw

On output, the Iw format specification writes the value of the corresponding I/O list element to an output character string of width w as an integer constant. The constant is right-justified with blank fill within the field, and the plus sign is not printed for nonnegative quantities.

If the value of the I/O list item requires a field larger than w, then w asterisks will be output.

Floating-point values are rounded to an integer value before being printed.

Examples:

Internal Value	Specification	External String
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
-67486	I5	*****
+978	I1	*
0	I3	bb0
+3.6	I2	b4

J Format Specification

The integer format specification Jw transfers data to or from internal storage as integer values.

Input Using Jw

On input, the Jw format specification functions identically to the Iw format specification.

Output Using Jw

On output, the Jw format specification writes the value of the corresponding I/O list element to be printed as an integer constant in the minimum field necessary to contain the value without exceeding the field width w. The plus sign is not printed for non-negative quantities.

If the value to be printed requires more than w characters, w asterisks will be output.

Floating-point values are rounded to an integer value before being output.

Examples:

Internal Value	Specification	External String
+23	J5	23
-23	J5	-23
+233	J3	233
-233	J3	***
0	J3	0
3.6	J3	4

L Format Specification

The logical format specification Lw transfers data to or from internal storage as logical values.

Input Using Lw

On input, the Lw format specification assigns to the corresponding logical element of the I/O list the value .TRUE. or .FALSE., depending upon the contents of the field of width w. If the left-most non-blank character is a "T", the variable will be assigned the value .TRUE.; otherwise, the value FALSE. will be assigned. An all-blank field will yield the value .FALSE..

Example:

External String	Specification	Internal Value
T	L1	.TRUE.
bbF	L3	.FALSE.
bbbTRU	L6	.TRUE.
b	L1	.FALSE.

Output Using Lw

On output, the Lw format specification writes the logical value of the corresponding I/O list item to be printed as an output string of width w as a logical constant. If the logical value of this element is .TRUE. or .FALSE., a T or F, respectively, will be placed right-justified with blank fill in this string.

The I/O list element need not be a logical item. If the numeric value of the item represents the logical value .TRUE. or .FALSE., the logical value of the item is .TRUE. or .FALSE., respectively (see the discussion of logical simple variables in chapter 11).

Examples:

Internal Value	Specification	External String
.FALSE.	L1	F
.FALSE.	L3	bbF
.TRUE.	L2	bT
2	L1	F
3	L3	bbT

O Format Specification

The format specification Ow transfers data to or from internal storage as octal values.

Input Using Ow

On input, the value represented by the octal digits in the input field of width w is assigned to the corresponding I/O list item. Leading, trailing, and embedded blanks are interpreted as zeros. A minus sign (–) causes bit 46 of the storage word allocated to the variable to be complemented (see chapter 11). If the list element is single precision and the input datum is less than 16 octal digits long, it is stored right-justified with zero fill in the storage word. If the list element is double precision, then if the input datum is 16 or fewer digits, it is stored right-justified with zero fill in the first word and zeros are stored in the second word; if it is longer than 16 digits, the first 16 digits are stored in the first word and the remaining digits are stored right-justified with zero fill in the second word.

Examples:

External String	Specification	Internal Value
16	O2	O0000000000000016
1777777777777777	O16	O1777777777777777
-16	O3	O2000000000000016
1234b56	O7	O0000000001234056
77b	O3	O0000000000000770

Output Using Ow

On output, the Ow format specification writes the value of the specified I/O list item as a string of octal digits, right-justified with blank fill in a field of w characters. If w does not exceed 16 (32 for double precision variables), the contents of the right-most w*3 bits of the storage word are printed as a string of w octal digits. A sign is never output.

Examples:

Internal Value	Specification	External String
O000567770123444 4	O5	34444
O0005677701234444	O16	0005677701234444
O0005677701234444	O18	bb0005677701234444
9	O2	11

P Format Specification

The specification fP modifies the scaling of numbers input and output using the D, E, F, and G format specifications. The integer constant f is the scale factor.

The scale factor affects all subsequent D, E, F, and G format specifications in a format specification list until a new scale factor is encountered. A specification list begins with a default scale factor of zero.

A scale factor has the following effect with the F format specification:

(external quantity)=(internal value)*10**f

With E and D format specifications on input, a scale factor has no effect if the external field contains an exponent part, and has the same effect as the F format specification if the external field does not contain an exponent part.

With Ew.d and Dw.d format specifications on output, a scale factor affects the positioning of the decimal point in the field of significant digits, causing the value of the exponent to be adjusted appropriately as follows:

a. If $f<0$, then $|f|$ zeros are placed to the right of the decimal point followed by $d-|f|$ significant digits.
b. If $f\geq0$, then f significant digits are placed to the left of the decimal point and d significant digits will be placed to the right of the decimal point.

A scale factor affects a G format specification only if the value is read or written using an E format.

Examples of input:

External String	Specification	Internal Value
12345	2P, F5.1	12.345
123.4	2P, F5.1	1.234
12345	2P, E5.1	12.345
123.4	2P, E5.1	1.234
1.2E4	2P, E5.1	1.234
123D4	2P, D5.1	1.23D5

Examples of output:

Internal Value	Specification	External String
1.234	2P, F10.2	bbbb123.40
1.234	-2P, F10.2	bbbbbb0.01
1.234	0P, E10.2	bbb.12E+01
1.234	2P, E10.2	b12.34E-01
1.234	-1P, D10.2	bbb.01D+02

T Format Specification

The format specification Tt allows tabbing to a position in the record. The item t is the character number, relative to the beginning of the record, of the character which is to be the start of the next external input or output string.

Example:

```
      X1 = 1.7
      X2 = 2.3
      WRITE(6,6) X1, SIN(X1), X2, SIN(X2)
    6 FORMAT(T20, 'X', T35, 'SIN(X)'/(2F20.2))
      END
```

This program produces the following output:

```
      X          SIN(X)
   1.70            0.99
   2.30            0.75
```

The T format specification may be used to reread a portion of an input record or write over a portion of an output record as illustrated by the following example:

```
      READ(5,5) N1,N2
    5 FORMAT (T6, J3, T7, J2)
      WRITE(6,6) N1, N2
    6 FORMAT(T5, 'N'/('b**********',T5,J3))
      END
```

If a data card contains 123456789 in columns 1-9, then this program produces the following output:

```
    N
***678****
***78*****
```

U Format Specification

The U format specification outputs a data item in a form best suited for the item. Integer, real, and double precision items are output in a format that combines readability with maximum numerical significance. Logical items are output as T or F and occupy one character position in the record. Character strings are treated as real items. If the number of characters required to edit the item is greater than the number of character positions left in the current record, the record is output and the item placed in the next record, with a blank inserted in the first character of printer and remote files.

The form Uw is similar to U, with the added restriction that the edited item may not exceed w characters in length. If the data item cannot be edited into a field of w characters, a field of w asterisks is output.

The form Uw.d is similar to Uw, with the added restriction that the total field width occupied by the edited item may not be less than d characters. The number of non-blank characters representing the data item itself may not exceed w characters. Thus, if d>w, d-w leading blanks will be inserted in the field.

Example: The statements

```
      X=12.3456
      WRITE(6,6).FALSE., 'XXXXXX',X,X,X,X
6     FORMAT(X,U,X,U,X,U,X,U3,X,U5.3,X,U3.5)
```

produce the output

```
Fb-0.0126838235294b12.3456b12.b12.35bbb12.
```

V Format Specification

Refer to the section on run-time format editing, discussed later in this chapter.

X Format Specification

The format specification wX skips w characters on input and inserts a field of w blank characters into the output record. The specification X has the same effect as 1X.

Z Format Specification

The hexadecimal format specification Zw transfers data to or from internal storage as hexadecimal values.

Input Using Zw

On input, the value represented by the hexadecimal digits in the input field of width w is assigned to the corresponding I/O list item. Leading, trailing, and embedded blanks are interpreted as zeros. A minus sign (–) causes bit 46 of the storage word allocated to the variable to be complemented (see chapter 11). If the list element is single precision and the input datum is less than 12 hexadecimal digits long, it is stored right-justified with zero fill in the storage word. If the list element is double precision, then if the input datum is 12 or fewer digits, it is stored right-justified with zero fill in the first word and zeros are stored in the second word; if it is more than 12 digits, the first 12 digits are stored into the first word and the remaining digits are stored right-justified with zero fill in the second word.

Examples:

External String	Specification	Internal Value
6F	Z2	Z00000000006F
1FFFFFFFFFFF	Z12	Z1FFFFFFFFFFF
-16	Z3	Z400000000016
1234b568	Z8	Z000012340568
FFCb	Z4	Z00000000FFC0
C1C2C3C4C5C6	Z12	ZC1C2C3C4C5C6 or 'ABCDEF'

Output Using Zw

On output, the Zw format specification writes the value of the specified I/O list item as a string of hexadecimal digits, right-justified with blank fill in a field of w characters. If w does not exceed 12 (24 for double precision variables), the contents of the right-most w*4 bits of the storage word are printed as a string of w hexadecimal digits. A sign is never output.

Examples:

Internal Value	Specification	External String
Z0000E5551010	Z5	51010
Z0000E5551010	Z12	0000E5551010
Z0000E5551010	Z16	bbbb0000E5551010
32	Z2	20
'123456'	Z12	F1F2F3F4F5F6

FORMAT SPECIFICATION MODIFIERS

Format specification modifiers may be placed immediately to the left of a format specification. If a repeat count is used, it should be to the left of any modifiers used. More than one modifier may be used with a format specification. A modifier may not be used on input.

For example, 2KF10.3 and 8K$G20.6 are valid, but $2J5 is not.

K Format Specification Modifier

The K format specification modifier causes commas to be inserted between digit triples to the left of the decimal point.

$ Format Specification Modifier

The $ format specification modifier causes the character $ to be inserted immediately to the left of an edited item.

Examples:

Internal Value	Specification	External String
17.347	$F10.2	bbbb$17.35
-1234567	K110	-1,234,567
-1234567	K$G15.2	bbbb$-1,234,567
1234567.11111	KF15.5	1,234,567.11111

THE FORMAT STATEMENT

The nonexecutable FORMAT statement specifies what type of data conversion and editing is to be performed during input or output.

The proper format for the FORMAT statement is:

n FORMAT(s)
where n is a statement label and s is a specification list which consists of format specifications modified as desired by parentheses, repeat counts, and specification modifiers, separated by commas and/or one or more slashes.

Correspondence is established between I/O list elements and the format specifications as previously described in this chapter.

If there is an I/O list on the I/O statement referencing the FORMAT statement, the specification list must contain at least one specification besides wHs, 's', "s", Tt, wX, X, P, or slash.

Record Fields

A field begins with the first character position of a record for the first I/O list element, or with the first character after the previous field. The Tt specification may be used to change the character pointer for subsequent editing.

Complex I/O List Elements

A complex I/O list element is considered, for purposes of I/O conversion, as two integer or real list elements. Thus, there must be two format specifications for every complex I/O list element.

Format Specification Separators

Two format specifications in a format specification list are separated by a comma, a slash, or a series of slashes. A slash is used to indicate a skip to the next record. On input, any remaining characters in the current record are ignored when a slash is encountered in the specification list. On output, the construction of the current record is terminated and any subsequent output is placed in the next output record. When performing internal data transfer, a slash in the specification list is ignored. Multiple slashes may be used to skip several records of input or generate several blank records on output. The final right parenthesis of a format also acts to indicate the end of the current record.

Repeat Counts

Format specifications and format specification list portions enclosed in parentheses may optionally be immediately preceded by an unsigned nonzero integer constant. This constant indicates the number of times that portion of the specification list is to be interpreted. If no such repeat count is indicated, a repeat count of 1 is assumed. The repeat count is limited to a maximum of 65,535.

If the outer right parenthesis of the format specification list is encountered before the I/O list is exhausted, control reverts to the repeat count (if present) of the repeat group terminated by the last preceding right parenthesis. If no other right parenthesis exists in the specification list, then control reverts to the first left parenthesis of the specification list.

The following are proper examples of the use of repeat counts. In each case, the repeat count is 3.

```
3F10.4
3(A6/)
3(3A6,3(/I12)/)
```

Run-Time Format Editing

If an asterisk (*) appears in a format specification list in place of a field width (w), a decimal place count (d), scale factor (f), or a tab position (t), then the I/O list will be accessed once and the value of the list item referenced will be used to replace the *.

For example, this program:

```
100     FORMAT(I3,2X,I*)
        PRINT 100,(J,J+1,10**J,J=0,7)
        STOP
        END
```

produces this output:

```
0  1
1  10
2  100
3  1000
4  10000
5  100000
6  1000000
7  10000000
```

In this program, the value of I+1 in the I/O list is used to replace the asterisk in the specification list.

If a V format specification appears in a format specification list, then the I/O list will be accessed and the contents of the rightmost eight bits of the data word referenced will be treated as an EBCDIC character to be used as the actual format specification. If the rightmost eight bits represent the blank character (Z40), then the character represented by the leftmost eight bits will be used. This character may be only an A, C, D, E, F, G, I, J, L, O, or Z; V is not acceptable.

For example, since the rightmost eight bits of the integer constant 193 represent the EBCDIC letter A, this program:

```
      DATA B/'TEXT'/
      PRINT 1,6,'A    ',B,193,B
1     FORMAT(I2,X,V4,/X,V4)
      STOP
      END
```

will produce this output:

```
6 TEXT
TEXT
```

Carriage Control

Any record which is intended to be output to a line printer or remote file must contain a prescribed carriage control character in its first character position. This character will not be output but is interpreted as follows:

Character	Effect
blank	Single space
0	Double space
1	Page eject
+	No line feed (overprint)
n (digit 2 thru 9)	Advance to the channel n

Any other character in the first character position will not be output, and single spacing will be assumed.

All carriage spacing indicated by the carriage control character is performed prior to output of the line.

Format Specifications in Arrays

A formatted READ, WRITE, PRINT, or PUNCH statement may employ an array name as a format designator. This array must contain an EBCDIC string representing a parenthesized format specification list. This specification list is identical to the specification list in a FORMAT statement.

Example:

The following program illustrates the use of format specifications contained in arrays (<I> denotes an invalid character in column 1).

```
<I>BEGIN JOB;
   COMPILE FORMAT/ARRAY WITH FORTRAN; FORTRAN EBCDIC
       DIMENSION FORM(2), DATA(6)
       DATA FORM/'(X,F6.3)'/, DATA/1.2,3.4,5.6,7.8,9.2,4.5/
       DO 1 J=1,3
   1   WRITE(6,FORM) DATA(J)
       READ(5,2) FORM
   2   FORMAT(2A6)
       DO 3 J=4,6
   3   WRITE(6,FORM) DATA(J)
       STOP
       END
<I>EBCDIC FILE5
 (X,F6.1)
<I>END JOB
```

This program produces the following output:

```
1.200
3.400
5.600
  7.8
  9.2
  4.5
```

14. NAMELIST STATEMENTS AND NAMELIST I/O

The nonexecutable NAMELIST statement associates a list of simple variables and arrays with a unique symbolic name called a namelist name. This namelist name may only be used as the format designator in a READ, WRITE, PRINT, or PUNCH statement.

The proper format for the NAMELIST statement is:

NAMELIST/n/p ...
where n is a namelist name, p is a namelist list which is a list of simple variables and array names separated by commas, and the ellipsis denotes as many repetitions of the clause "/n/p" as desired with different namelist names.

A namelist name is constructed in the same manner as a variable name. A namelist name may only be used as a format designator and must be declared as such by a NAMELIST statement appearing in the program unit in which it is used. The I/O statement using the namelist name may not contain an I/O list. The namelist name may not be a dummy argument nor may the elements of a namelist list be dummy arguments. This name may be used for no other purpose.

The NAMELIST statement must follow any declarations referencing the elements of the namelist list.

A particular simple variable or array may be associated with more than one namelist name. For example, if A, B, and C are array names, the NAMELIST statement:

```
NAMELIST/NMLST1/A,G/NMLST2/A,B,A2/LIST/C,XR,S,T
```

associates the array A and the simple variable G with the namelist name NMLST1, associates the arrays A and B and the simple variable A2 with the namelist name NMLST2, and associates the array C and the simple variables XR, S, and T with the namelist name LIST.

INPUT USING NAMELIST

Values may be input to internal storage locations from peripheral storage by formatted READ statements employing namelist names as format designators. When such a READ statement is executed, the referenced file is accessed repeatedly, beginning at its present position, until an appropriate namelist input record (as described below) is found or an end-of-file condition occurs. namelist input records must have the following format:

a. A namelist record consists of one or more physical records.
b. The first character of each physical record is blank, and the second character of the first physical record must be an ampersand (&). The namelist name must immediately follow the & and contain no embedded blanks. This name is followed by at least one blank, which is followed by the list of value assignments.
c. Each namelist record contains a list of value assignments, separated by commas. A value assignment is either a simple variable followed by an equal sign (=) and a constant, or an array name or array element name followed by an equal sign which is followed by one or more constants separated by commas, with or without repeat specifications. No embedded blanks may appear in the names or the constants. Trailing blanks after exponents and integer constants are interpreted as zeros. Each variable and array referenced in this list must have been associated with the namelist name in a NAMELIST statement.
d. The constants in the value assignments are assigned to the indicated variables and array elements when a READ statement referencing that namelist name is executed and encounters the namelist record. The constants may be numeric literals or string literals containing one to the value of CHARS characters (see chapter 4). If assignment is being made to an array name or array element, the constant(s) may be preceded immediately by a repeat specification of the form: n*, where n is an unsigned nonzero integer constant.
e. If assignment is being made to an array name, up to as many values as there are elements in the array may appear in the value list. If assignment is being made to an array element, up to as many values as there are succeeding elements in the array may appear in the value list. Values are assigned to the elements of the array in the order in which they are stored internally, beginning with the indicated element (see chapter 11).
f. The namelist record must be terminated by "&END". The remaining characters of the physical record on which this item appears will be ignored.

After the physical records bearing a namelist record are read, the file is positioned at the next record in the file.

Data items associated with the namelist name which do not have values assigned to them in the namelist record will retain their present values.

Example

The following compile-and-go deck illustrates the use of NAMELIST for input. (<I> denotes an invalid character in card column 1.)

```
<I>BEGIN JOB TEST;
   COMPILE NAMELIST/TEST WITH FORTRAN;FORTRAN EBCDIC
C ** INPUT USING NAMELIST **
      DIMENSION A(2,2),B(3)
      NAMELIST /NTEST/A,B,C,D
      DATA B(2)/6/
      READ 1,C
1     FORMAT(I1)
      FIND(5=C)
      READ 2,C
2     FORMAT(F3.1)
      READ NTEST
      READ 1,B(3)
      PRINT 3,A,B,4,C,D
3     FORMAT(I2/2(X,A6/),F4.1/3(I2/),F*.1/X,A3)
      STOP
      END

<I>EBCDIC FILE5
 3
 3.2
 6.4
 9.6
 2.0
  &NTEST A=1,2*'ABCDEF',1,B=7,
  D=3HIJK7END    (END OF NTEST RECORD)
 5
<I>END JOB
```

When this program is executed with the indicated data deck, the first record of FILE5 is accessed and the value 3 is input to the variable C. The file is then positioned at the record indicated by the value of C (record 3), and the value 6.4 contained on this record is input to C. When the READ NTEST statement is executed, FILE5 is accessed repeatedly until the NTEST namelist record is located, at which time the value assignments indicated in this record are made. Since the names A, B, and D had been associated with the namelist name NTEST by a preceding NAMELIST statement, the value assignments indicated are valid. The values specified are assigned to the elements of the array A as if these assignment statements had been executed in this order:

```
A(1,1)=1
A(2,1)='ABCDEF'
A(1,2)='ABCDEF'
A(2,2)=1
```

The value 7 is assigned to array element B(1), and the value 'IJK ' is assigned to the variable D. The note "(END OF NTEST RECORD)" is ignored. Another read is performed on this file and the value 5 is read from the record immediately following the namelist record into the array element B(3).

Therefore, the printed output is:

```
1
ABCDEF
ABCDEF
1.0
7
6
5
6.4
IJK
```

OUTPUT USING NAMELIST

Values may be output to peripheral storage from internal storage locations by formatted WRITE, PRINT, and PUNCH statements employing namelist names as format designators. When such an output statement is executed, the value of every variable and array element associated with the namelist name is placed in a generated namelist record in the referenced file; the record is suitable for input by a READ statement employing that namelist name.

The generated NAMELIST record is of the form discussed in the preceding section. No repeat specifications will appear in the output, and for arrays a value will be output for every element of the array in the order in which the elements are stored internally (see chapter 11).

Values are output according to the declared or default type of each data item. No facilities are available for the writing of strings using namelist output.

The following program excerpt employs namelist output.

```
DIMENSION A(4,4),M(10),N(20)
NAMELIST /NAMEA/A,D,K,M,N,X /NAMEB/M,N,X
.
.
.
WRITE(6,NAMEA)
.
.
.
PUNCH NAMEB
.
.
.
```

USING NAMELIST WITH SEPARATE COMPILATIONS

A syntax error is given if a global variable (i.e., a variable initialized in a DATA or type statement, a variable in COMMON storage, or a variable affected by the OWN compiler option) appears as an element of a namelist list in a subprogram if the SEPARATE option is set for the compilation.

15. SUBPROGRAMS AND INTRINSICS

Subprograms are program units which may be invoked in the main program, another subprogram, or the same subprogram as a separate executable procedure, or which may be used for data initialization.

SUBROUTINE SUBPROGRAMS

A subroutine is a sequence of statements initiated by a SUBROUTINE statement and terminated by an END statement.

SUBROUTINE Statement

The nonexecutable SUBROUTINE statement indicates the beginning of a subroutine subprogram and may specify the dummy arguments employed in that subprogram.

The proper formats for the SUBROUTINE statement are:

```
1.  SUBROUTINE s
2.  SUBROUTINE s (d)
```

where s is a SUBROUTINE name, and d is a list of dummy arguments, separated by commas.

Subroutine Names

A subroutine name is constructed in the same manner as a variable name. No type is associated with this name, and once a symbolic name is used as a subroutine name it may be used for no other purpose in that program.

Dummy Argument Lists

Each element of a dummy argument list may be a simple variable name, an array name, a dummy subroutine name, or an asterisk (*). A simple varible may optionally be enclosed in slashes (/).

If a dummy argument is a simple variable name, the actual argument in the subroutine call must be a variable or an expression. The actual argument must be of the same type as the dummy argument, with the exception that the actual argument may be real and the dummy integer, and vice versa.

If an array appears as a dummy argument, the corresponding actual argument must be an array or array element. In the latter case, the actual argument may be thought of as an array whose first element is that array element. Array handling between actual and dummy argument lists is discussed in the section on the CALL statement in chapter 8. When a dummy argument is an array, that array may have variable bounds. Care should be taken that the dummy array is not declared larger than the corresponding actual array or an invalid index condition may result when the dummy array is referenced.

An asterisk in a dummy argument list corresponds to a statement label preceded by an ampersand (&) in the actual argument list; this argument is used in connection with nonstandard RETURN statements, described in chapter 8.

The following are valid examples of SUBROUTINE statements:

```
SUBROUTINE SUBA
SUBROUTINE MULTMT(ARRAY1,ARRAY2,*)
SUBROUTINE MAX(A,B,C,/D/,E,*,F)
```

Use of Subroutines

The executable statements in a subroutine body are executed when the subroutine is invoked by a CALL statement. The subroutine may cause program termination by executing a STOP or CALL EXIT statement or may cause control to return to the calling program unit by the RETURN statement. There is always an implied RETURN statement preceding the END statement.

Arrays passed as arguments to a subroutine are always called by name. The subroutine is thus passed the starting address of the array and thereby accesses the actual array. Any array elements which have their values changed in the subroutine will retain these new values when control is returned to the calling program unit.

Simple variables and array elements are also called by name in that if the value of the dummy argument may have changed, the final value of the dummy argument will be assigned to the corresponding actual argument when execution of the subroutine is completed. All other arguments are called by value in that the value of the argument is passed to the subroutine but the subroutine does not affect the actual argument. A simple variable or array element will be passed by value if the actual argument is preceded by a unary +.

If a dummy argument is enclosed in slashes, any change in the value of the dummy argument in the subroutine will be recorded as a change in the value of the actual argument at the same time.

Consider the following program:

```
COMMON X
CALL  S(X)
PRINT/,X
END

SUBROUTINE S(/A/)
COMMON X
A=2
X=3.
RETURN
END
```

The value printed for X is 3.0. However, if the dummy argument A were not enclosed in slashes, the value printed for X would be 2.0, since the value of the dummy argument A is 2.0 upon completion of the subroutine. At that point, it is assigned as the value of the actual argument X, changing its value from 3.0 to 2.0.

FUNCTION SUBPROGRAMS AND STATEMENT FUNCTIONS

FORTRAN functions are procedures which return a single value to a calling program unit at the point at which they are referenced. A function may consist of a program unit called a function subprogram or may be declared as a single expression via a statement function declaration appearing in the program unit in which it is referenced.

Function Subprogram

A function subprogram is a sequence of statements initiated by a FUNCTION statement and terminated by an END statement.

FUNCTION Statement

The nonexecutable FUNCTION statement indicates the beginning of a function subprogram and specifies the dummy arguments employed in that subprogram to obtain the function value.

The proper formats for the FUNCTION statement are:

1. FUNCTION f(d) 2. t FUNCTION f(d)
where f is a function name, d is a list of dummy arguments separated by commas, and t is a type as defined in chapter 7.

Function Names

A function name is constructed in the same manner as a variable name. A type is associated with this symbolic name which indicates the type of the value returned when the function is referenced. Default types are assigned to function names in the same manner as for variable names.

The default type associated with a function name may be altered through the item t in the FUNCTION statement. For example, the function LGL will be of type logical if it is initiated by a FUNCTION statement of the form:

```
LOGICAL FUNCTION LGL (A)
```

When a function is of a non-default type, then the function name must appear in an appropriate type statement or be affected by an IMPLICIT statement in every program unit in which that function is referenced.

Once a symbolic name has been used as a function name, it may be used for no other purpose in that program. Such a name may appear on the left side of an equal sign in an assignment statement only in the body of the function subprogram. The value to be returned must be assigned to the function name in this manner before a return is executed.

The function name of a function of type real may optionally be followed in the FUNCTION statement by the specification: *8. This specification assigns the type double precision to the function.

For example, the FUNCTION statement

```
REAL FUNCTION FNC*8(X)
```

declares FNC to be a function of type double precision.

Dummy Argument Lists

The list of dummy arguments in a FUNCTION statement is defined in exactly the same manner as the dummy argument list of a SUBROUTINE statement. Every FUNCTION statement must contain at least one dummy argument.

Example of Function Subprogram

The following sample function subprogram, RSHIFT, returns a word identical to the word passed to the dummy argument A with each bit in the word shifted over N bit positions. If N is greater than 0, a right circular shift is performed. If N is less than 0, a left circular shift is performed.

```
      FUNCTION RSHIFT(A,N)
C     CIRCULAR RIGHT SHIFT N BITS FOR N>0
C     CIRCULAR LEFT SHIFT -N BITS FOR N<0
      M = MOD(N,48)
      RSHIFT = A
      IF(M.EQ.0) RETURN
      IF(M.LT.0) M = M+48
C     MOVE RIGHT MOST M BITS OF A INTO RSHIFT
      RSHIFT = CONCAT(RSHIFT,A,47,M-1,M)
C     MOVE REMAINING 48-M BITS OF A INTO RSHIFT
      RSHIFT = CONCAT(RSHIFT,A,47-M,47,48-M)
      RETURN
      END
```

The CONCAT and MOD intrinsics are described later in this chapter.

Statement Functions

A statement function is a function which may be defined in one statement.

Statement Function Declaration

The nonexecutable statement function declaration declares a statement function and specifies the dummy arguments used by the statement function.

The proper format for a statement function declaration is:

f(d)=e
where f is a function name, d is a list of dummy arguments separated by single commas, and e is an expression.

A statement function is local to the program unit in which the statement function declaration appears. It is referenced in that program unit exactly like any other function and may not be referenced by any other program unit.

The name of a statement function may not appear in an EXTERNAL statement. A warning message is issued by the compiler if a statement function declaration follows the first executable statement in the program unit.

Function Type

The type of the value returned when a statement function is referenced depends upon the type associated with the function name. A default type is associated with this name in the same manner as for a variable name. This default type may be changed by the appearance of the function name in a type statement or by the implicit typing of the name in an IMPLICIT statement in the same program unit.

For example, the following is a valid combination of statements:

```
IMPLICIT LOGICAL (L)
LFCN(A,B)=A.IS.B
```

When this statement function is referenced, the 48 bits in words A and B will be compared, and if their states are identical, the function will return the value .TRUE.. The expression in a statement function may be a logical expression only if the function name is of type logical.

Dummy Argument Lists

The list of dummy arguments in a statement function may contain only simple variable names (without slashes).

Use of Functions

A function is invoked when a function reference is encountered in an expression. The value obtained from the function invocation is returned to the point of the function reference.

A function reference has the form: f(a), where f is the function name (or the name of an entry in a function), and a is a list of actual arguments.

Execution of a function subprogram begins with the first executable statement following the FUNCTION or ENTRY statement.

The actual argument list of a function reference may contain the same items as an actual argument list of a CALL statement. The actual argument list of a statement function reference may contain only expressions.

If a function reference occurs in an I/O list, the execution of that function may not cause any I/O statement to be executed which references the same file as the I/O statement bearing the I/O list.

An alternate return executed within a function subprogram will result in no value being returned by the function.

Array handling in function subprograms is performed in the same manner as for subroutines and a variable name enclosed in slashes will be called by name in the same manner as for subroutines.

RECURSION

A subroutine or function subprogram may be defined "recursively" by containing a reference to itself or to another subprogram that may directly or indirectly reference it.

Example (Towers of Hanoi): N disks, all of different sizes, are placed on a pole (call it pole 1) in order of their size, with the largest at the bottom. The disks are to be moved to pole 3, using an additional pole 2, subject to the following rules:

a. Only one disk may be moved at a time.
b. No disk may ever be placed on a pole above a smaller disk.

The following program prints the moves to be made for N=4 disks.

```
DATA N/4/
CALL HANOI(N,1,2,3)
END

SUBROUTINE HANOI(N-I,N1,N3,N2)
IF(N.GT.1) CALL HANOI(N-1,N3,N2)
PRINT//,"MOVE DISK",N,"FROM",N1,"TO",N3
IF(N.GT.1) CALL HANOI(N-1,N2,N1,N3)
RETURN
END
```

ENTRY STATEMENT

A subroutine or function subprogram may contain one or more nonexecutable ENTRY statements which allow execution to begin at a point other than the first executable statement in the subprogram when the subprogram is invoked.

The proper formats for the ENTRY statement are:

1. ENTRY p 2. ENTRY p(d)
where p is an entry name and d is a list of dummy arguments separated by commas.

Entry Names

An entry name is constructed in the same manner as a variable name. A type is associated with an entry name only if the ENTRY statement appears in a function subprogram. Once a symbolic name has been used as an entry name it may be used for no other purpose in that program.

Dummy Argument Lists

The dummy argument list of an ENTRY statement in a subroutine subprogram is constructed identically to the dummy argument list of a SUBROUTINE statement. The dummy argument list of an ENTRY statement in a function subprogram is constructed identically to the dummy argument list of a FUNCTION statement.

Entry in Subroutine Subprograms

An ENTRY statement appearing in the body of a subroutine subprogram provides an alternate point at which to begin execution when the subroutine is called. Execution begins with the first executable statement after the ENTRY statement. The associated entry name is considered to be a SUBROUTINE name.

The dummy argument list in an ENTRY statement is independent of any other dummy argument lists in the program unit. If the same names appear in more than one dummy argument list in a subprogram, they need not occupy the same positions in those lists. An ENTRY statement may contain more or fewer dummy arguments than the SUBROUTINE statement. Entry into a subroutine at any point results in initialization of only those dummy arguments included in the dummy argument list of the statement at which entry occurs. Of the remaining dummy arguments, those which are simple variables are reset to zero, and those which are arrays are nonexistent for this call. Local variables are reset to zero or retain the values assigned them by initial value lists in the normal manner (see compiler options OWN and OWNARRAYS, chapter 19).

The following is an example of a subroutine containing an ENTRY statement:

```
SUBROUTINE SUBA(A,B)
A=A/2.0
ENTRY XSUBA(B,A)
A=A**B
B=A
RETURN
END
```

This subroutine may be called by statements of both these forms:

```
CALL SUBA(G,H)
CALL XSUBA(G,H)
```

Entry in Function Subprograms

An ENTRY statement appearing in the body of a function subprogram provides an alternate point at which to begin execution when the function is referenced. Execution begins with the first executable statement after the ENTRY statement. The associated entry name is considered to be a function name.

If the entry name is to have a non-default type, it must be referenced in a type or IMPLICIT statement both in the calling program unit and in the function subprogram body.

All entry names in a function subprogram are equivalenced to the function name; thus, the value to be returned from the function subprogram may be assigned to either an entry name or a function name before a RETURN statement is executed.

The following is an example of a function subprogram containing an ENTRY statement:

```
FUNCTION QSIN(X)
A=2.3
ENTRY RSIN(X)
QSIN=1.1 + A + SIN(X)*2
RETURN
END
```

This function may be referenced, for example, in either of these ways:

```
IF(QSIN(B).LT.1.5) GO TO 7
IF(RSIN(B).LT.1.5) GO TO 7
```

BLOCK DATA SUBPROGRAMS

A block data subprogram is a sequence of statements initiated by a BLOCK DATA statement and terminated by an END statement. The purpose of this subprogram is to allow elements of common storage to be assigned initial values at compile time.

BLOCK DATA Statement

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

The proper format for this statement is:

```
BLOCK DATA
```

Use of Block Data

The block data subprogram may contain no executable statements. Its sole purpose is to specify initial values for common storage by means of DATA statements or initial value lists on type statements.

There may be as many block data subprograms as desired in an executable program. None may be used with separate compilations.

A block data subprogram may contain the following statements in the indicated order:

- BLOCK DATA statement.
- IMPLICIT statement.
- Type statements.
- COMMON statements.
- DATA statements.
- END statement.

The following is a valid example of a block data subprogram:

```
BLOCK DATA
IMPLICIT LOGICAL (L)
COMMON L1,LCNTRL/BLOCKA/A(3)
DATA L1,LCNTRL,A/2*.TRUE,3*1.2/
END
```

FAULT-HANDLING SUBROUTINES

The fault-handling subroutines DVCHK and OVERFL handle divide-by-zero and exponent underflow/overflow faults during execution of a program.

DVCHK Subroutine

A reference to the DVCHK subroutine has the following form:

CALL DVCHK(v)
where v is an integer or real simple variable.

Use of this statement within a program which does not contain a subroutine named DVCHK allows the program to handle divide-by-zero faults. If a program contains at least one CALL DVCHK(v) statement and a divide-by-zero fault occurs during execution of the program, a zero will be returned as the result of the divide and execution of the program will continue from the point where the fault occurred.

When the CALL DVCHK(v) statement is executed, a value will be returned to the variable v, depending on whether a divide-by-zero has occurred since the last call on DVCHK or, in the case of the first call, since the beginning of the program. The possible values and their meanings are as follows:

v=1 means a divide-by-zero has occurred
v=2 means no divide-by-zero has occurred

Example:

```
R=R/S
CALL DVCHK(I)
IF (I.EQ.1) GO TO 20
. . .
```

OVERFL Subroutine

A reference to the OVERFL subroutine has the following form:

CALL OVERFL(v)
where v is an integer or real simple variable.

Use of this statement within a program which does not contain a subroutine named OVERFL allows the program to handle exponent overflow and exponent underflow faults. If a program contains at least one CALL OVERFL(v) statement and an exponent overflow or exponent underflow fault occurs during execution of the program, a result is returned and execution continues from that point. When exponent underflow occurs, zero is returned. When exponent overflow occurs, the largest possible real or double precision value is returned.

When the CALL OVERFL(v) statement is executed, a value will be returned to the variable v, depending on whether an exponent underflow or exponent overflow has occurred since the last call on OVERFL or, if this is the first call, since the beginning of the program. The possible values and their meanings are as follows:

v=1 means exponent overflow has occurred
v=2 means neither exponent overflow nor exponent underflow has occurred
v=3 means exponent underflow has occurred

Example:

```
R=R*R
CALL OVERFL(J)
PRINT/,J,R
. . .
```

DVCHK and OVERFL may not be used when binding is desired, or in a batch job.

INTRINSICS

Intrinsics may be considered as function subprograms which are known to the compiler and need not be supplied in the program. An intrinsic may, however, be redefined by providing a function subprogram with the same name as the intrinsic.

Table 15-1 lists the intrinsic names recognized by the B 7000/B 6000 series FORTRAN compiler. The type of the value returned by the intrinsic is indicated as are the number and types of actual arguments passed to the intrinsic. ("DOUBLE" indicates type double precision.) A brief description of the function performed by each intrinsic is also given. In such descriptions, A1 and A2 represent the first and second arguments passed to the intrinsic, respectively. Multiple arguments must always be separated by single commas; actual arguments must agree in number and type with the specifications in the table, with real and integer types being interchangeable.

The following symbols are used in table 15-1:

* indicates that the intrinsic is an in-line intrinsic; that is, code is placed in the code file at the point of the intrinsic reference to perform the indicated function.

Text Deleted

** indicates that the intrinsic is a special compile-time procedure which produces a value during compilation.

Table 15-1. B 6700/B 7700 Intrinsics

Intrinsic Name	Intrinsic Type	Argument Type(s)	No. of Arguments	Function Performed
ABS*	REAL	REAL	1	Absolute value
AIMAG*	REAL	COMPLEX	1	Obtain imaginary part
AINT*	REAL	REAL	1	Largest integer ≤ |A1|
ALGAMA	REAL	REAL	1	Log gamma function
ALOG	REAL	REAL	1	Natural logarithm
ALOG10	REAL	REAL	1	Common logarithm
AMAX0*	REAL	INTEGER	≥2	Choose maximum value
AMAX1*	REAL	REAL	≥2	Choose maximum value
AMIN0*	REAL	INTEGER	≥2	Choose minimum value
AMIN1*	REAL	REAL	≥2	Choose minimum value
AMOD*	REAL	REAL	2	Remaindering: A1 mod A2
AND*	REAL	REAL	2	48-bit logical product
ARCOS	REAL	REAL	1	Arcosine
ARSIN	REAL	REAL	1	Arcsine
ATAN	REAL	REAL	1	Arctangent
ATANH	REAL	REAL	1	Hyperbolic arctangent
ATAN2	REAL	REAL	2	Arctangent of A1/A2
CABS	REAL	COMPLEX	1	Absolute value
CCOS	COMPLEX	COMPLEX	1	Trigonometric cosine
CDABS	REAL	COMPLEX	1	Absolute value
CDCOS	COMPLEX	COMPLEX	1	Trigonometric cosine
CDEXP	COMPLEX	COMPLEX	1	Exponential: e**A1
CDLOG	COMPLEX	COMPLEX	1	Natural logarithm
CDSIN	COMPLEX	COMPLEX	1	Trigonometric sine
CDSQRT	COMPLEX	COMPLEX	1	Square root
CEXP	COMPLEX	COMPLEX	1	Exponential: e**A1
CLOG	COMPLEX	COMPLEX	1	Natural logarithm
CMPLX*	COMPLEX	REAL	2	Create complex = (A1,A2)
COMPL*	REAL	REAL	1	48-bit logical complement
CONCAT*	REAL	(See text)	5	Partial word manipulation
CONJG*	COMPLEX	COMPLEX	1	Complex conjugate
COS	REAL	REAL	1	Trigonometric cosine
COSH	REAL	REAL	1	Hyperbolic cosine
COTAN	REAL	REAL	1	Trigonometric cotangent
CSIN	COMPLEX	COMPLEX	1	Trigonometric sine
CSQRT	COMPLEX	COMPLEX	1	Square root
CTIME**	(See text)	REAL	1	Special compile-time TIME function
DABS*	DOUBLE	DOUBLE	1	Absolute value
DARCOS	DOUBLE	DOUBLE	1	Arccosine
DARSIN	DOUBLE	DOUBLE	1	Arcsine
DATAN	DOUBLE	DOUBLE	1	Arctangent
DATAN2	DOUBLE	DOUBLE	2	Arctangent of A1/A2
DBLE*	DOUBLE	REAL	1	Type conversion
DCMPLX*	COMPLEX	DOUBLE	2	Create complex = (A1,A2)
DCONJG*	COMPLEX	COMPLEX	1	Complex conjugate
DCOS	DOUBLE	DOUBLE	1	Trigonometric cosine
DCOSH	DOUBLE	DOUBLE	1	Hyperbolic cosine

Table 15-1. B 6700/B 7700 Intrinsics (continued)

Intrinsic Name	Intrinsic Type	Argument Type(s)	No. of Arguments	Function Performed
DERF	DOUBLE	DOUBLE	1	Error function
DERFC	DOUBLE	DOUBLE	1	1-DERF
DEXP	DOUBLE	DOUBLE	1	Exponential: e**A1
DFLOAT*	DOUBLE	INTEGER	1	Type conversion
DGAMMA	DOUBLE	DOUBLE	1	Gamma function
DIM*	REAL	REAL	2	A1-min(A1,A2)
DLGAMA	DOUBLE	DOUBLE	1	Log gamma function
DLOG	DOUBLE	DOUBLE	1	Natural logarithm
DLOG10	DOUBLE	DOUBLE	1	Common logarithm
DMAX1*	DOUBLE	DOUBLE	≥2	Choose maximum value
DMIN1*	DOUBLE	DOUBLE	≥2	Choose minimum value
DMOD*	DOUBLE	DOUBLE	2	Remaindering: A1 (mod A2)
DSIGN*	DOUBLE	DOUBLE	2	(sign of A2)*|A1|
DSIN	DOUBLE	DOUBLE	1	Trigonometric sine
DSINH	DOUBLE	DOUBLE	1	Hyperbolic sine
DSQRT	DOUBLE	DOUBLE	1	Square root
DTAN	DOUBLE	DOUBLE	1	Trigonometric tangent
DTANH	DOUBLE	DOUBLE	1	Hyperbolic tangent
EQUIV*	REAL	REAL	2	48-bit logical equivalence
ERF	REAL	REAL	1	Error function
ERFC	REAL	REAL	1	1-ERF
EXP	REAL	REAL	1	Exponential: e**A1
FLOAT*	REAL	INTEGER	1	Type conversion
GAMMA	REAL	REAL	1	Gamma function
HFIX*	INTEGER	REAL	1	Type conversion
IABS*	INTEGER	INTEGER	1	Absolute value
IDIM*	INTEGER	INTEGER	2	A1-min(A1,A2)
IDINT*	INTEGER	DOUBLE	1	Type conversion
IFIX*	INTEGER	REAL	1	Type conversion
INT*	INTEGER	REAL	1	Largest integer ≤ |A1|
ISIGN*	INTEGER	INTEGER	2	(sign of A2)*|A1|
MAX0*	INTEGER	INTEGER	≥2	Choose maximum value
MAX1*	INTEGER	REAL	≥2	Choose maximum value
MIN0*	INTEGER	INTEGER	≥2	Choose minimum value
MIN1*	INTEGER	REAL	≥2	Choose minimum value
MOD*	INTEGER	INTEGER	2	Remainder: A1 (mod A2)
OR*	REAL	REAL	2	48-bit logical sum
RANDOM	REAL	REAL	1	Pseudo-random number generator
REAL*	REAL	COMPLEX	1	Obtain real part
SIGN*	REAL	REAL	2	(sign of A2)*|A1|
SIN	REAL	REAL	1	Trigonometric sine
SINH	REAL	REAL	1	Hyperbolic sine
SNGL*	REAL	DOUBLE	1	Type conversion
SQRT	REAL	REAL	1	Square root
TAN	REAL	REAL	1	Trigonometric tangent
TANH	REAL	REAL	1	Hyperbolic tangent
TIME	(See text)		1	Time function (returned by MCP)
VALUE**	(See text)		1	Numeric value of attribute mnemonic

The FORTRAN compiler will recognize other intrinsic names besides those just listed if the compiler option INSTALLATION is set for the compilation, the intrinsics file presently in use by the system contains the referenced intrinsic, and the intrinsic may be used by FORTRAN. Non-in-line and non-MCP intrinsics correspond to code in the intrinsics file.

The algorithms which are employed by the system for mathematical intrinsics are discussed in the Mathematical Intrinsics unit of the Systems Operation Guide, Form Number 5001563.

If an intrinsic is to be passed as an actual argument, its name must appear in an EXTERNAL statement. CTIME, TIME, VALUE, and the in-line intrinsics may not be passed as actual parameters.

An intrinsic is referenced in the same manner as a function subprogram.

The intrinsics CDABS, CDCOS, CDEXP, CDLOG, CDSIN, CDSQRT, DCMPLX, and DCONJG are identical to the intrinsics CABS, CCOS, CEXP, CLOG, CSIN, CSQRT, CMPLX, and CONJG, respectively.

Intrinsics whose types do not agree with the types which would be assigned to their names by default need not be named in type statements in the program units in which they are referenced. Intrinsics differ from other functions in this respect.

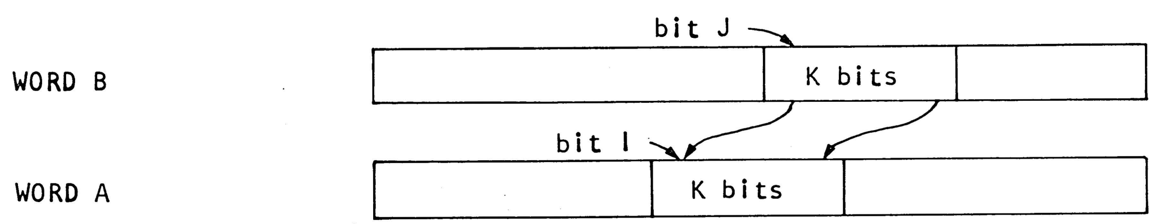
In the intrinsics MOD and AMOD, when the second argument is less than 1, the result is undefined by the current implementation.

CONCAT Intrinsic

The CONCAT intrinsic is provided to allow partial word manipulation.

This intrinsic takes five actual arguments. The first two of these arguments must be integer or real variables. The last three arguments must be non-negative arithmetic expressions. Thus, a typical CONCAT reference has the form: CONCAT(A,B,I,J,K). The truncated values of I and J must be less than 48; the truncated value of K must be less than or equal to 48.

The value returned from the intrinsic reference is equal to the word A with field [I:K] replaced by field [J:K] from word B (see chapter 11 for an explanation of field notation). Neither A nor B are changed by the reference to CONCAT.



The following is an example of a program unit referencing the CONCAT intrinsic:

```
FUNCTION UNITE(P1,P2)
UNITE=CONCAT(P1,P2,23,47,24)
RETURN
END
```

The value returned when UNITE is referenced is a word constructed from P1 and P2 in such a manner that the leftmost 24 bits of P1 and the leftmost 24 bits of P2 are placed into the word in that order.

Greater efficiency may be obtained by using the AND and OR intrinsic, whenever possible, instead of the CONCAT intrinsic.

CTIME Intrinsic

Appearance of the intrinsic CTIME in a statement returns the contents of certain internal timing registers at compile-time. The CTIME intrinsic accepts the same range of argument and returns values of the same type as the TIME intrinsic. The argument must be an integer or real constant--real arguments will be truncated to obtain an integer value.

This intrinsic has effect only at compilation time. The compiler replaces the CTIME reference by the integer value contained from the appropriate timing registers at that point in time. For example, if the assignment statement:

```
X = CTIME(1)
```

appears in the source input, the compiler will substitute for CTIME(1) an integer value which is the current time of day in units of sixtieths of a second. When the statement is executed, the variable X will be assigned this constant integer value.

A reference to CTIME may not appear in an initial value list.

RANDOM Intrinsic

A reference to the RANDOM intrinsic returns a pseudo-random number between 0 and 1. The actual argument passed to this intrinsic must be an integer or real variable. The value of this variable is changed each time the intrinsic is executed; the value returned to the argument is also a pseudo-random number but is an integer in the interval 1 to 2**39-1. Thus, the value of the argument should not be changed in the program after the first reference to RANDOM.

The value returned by the intrinsic is from a uniform distribution.

TIME Intrinsic

A reference to the TIME intrinsic returns a value which represents the present contents of various internal timing registers. The argument must have a value when truncated in the interval 0 to 15. The following argument values return the indicated values:

0 - returns the current date in BCL-coded digits in the format: 0yyddd, where y is a year digit and d is a day digit;

1 - returns an integer value which is the time of day in units of sixtieths of a second;

2 - returns an integer value which is the elapsed processor time of the job in units of sixtieths of a second;

3 - returns an integer value which is the elapsed I/O time of the job in units of sixtieths of a second;

4 - returns an integer value which is the current contents of a 6-bit machine clock which increments every sixtieth of a second;

5 - returns the same as 0, except that the date is expressed as: 00mmddyy, where m is a month digit;

6-9 return an integer zero;

10- returns the same as 0, except that the digits are EBCDIC-coded;

11- returns the same as 1, except that the time unit is 2.4 microseconds;

12- returns the same as 2, except that the time unit is 2.4 microseconds;

13- returns the same as 3, except that the time unit is 2.4 microseconds;

14- returns an integer value which is the current contents of a 36-bit machine clock which increments every 2.4 microseconds;

15- returns the same as 5, except that the digits are EBCDIC-coded.

The following is a sample program unit employing the TIME intrinsic. The subroutine DATIT returns the current date in the form mm/dd/yy left-justified in the array, DATE.

```
SUBROUTINE DATIT(DATE)
REAL DATE(2)
DATE(1)='  /  /'
DATE(2)='      '
B=TIME(15)
DATE(1)=CONCAT(DATE(1),B,47,47,16)
DATE(1)=CONCAT(DATE(1),B,23,31,16)
DATE(2)=CONCAT(DATE(2),B,47,15,16)
RETURN
END
```

VALUE Intrinsic

A reference to the VALUE intrinsic returns a numeric value corresponding to a file attribute mnemonic. The VALUE intrinsic requires one argument, which is a valid file attribute mnemonic. For a list of these mnemonics, refer to Input/Output Subsystem, Form Number 5001779. The compiler replaces the VALUE reference by the numeric value of the mnemonic; this value will be of the same type as the associated attribute.

Examples:

```
INQUIRE(37, KIND-DEVICE,EXTMODE = XMODE)
IF(DEVICE.EQ.VALUE(DISK)) GO TO 35
IF(XMODE.EQ.VALUE (BCL)) XMODE=VALUE(EBCDIC)
```

Installation Intrinsics

A FORTRAN installation intrinsic is a subroutine or function subprogram which is automatically compiled at level 2. An installation intrinsic is bound into an intrinsic file after compilation and is used like a standard intrinsic.

The INTRINSICS compiler option must be set before the first source statement of the compile deck. The SEPARATE option is automatically SET. The LIBRARY option must be explicitly set if desired. The INTRINSICS option cannot be set in batch mode.

The compiler options OWN, OWNARRAYS, and LEVEL cannot be set in an intrinsic. The following items cannot appear in an intrinsic:

a. A STATISTICS or DUMP STATISTICS statement.
b. A STOP or CALL EXIT statement.
c. A DATA statement or initial value list.
d. A COMMON statement.
e. An EXTERNAL statement.
f. An ENTRY statement.
g. Formal subprograms.
h. File declarators.
i. A PRINT or PUNCH statement.
j. A CHANGE, OPEN, INQUIRE, LOCK, PURGE, CLOSE, FIND, BACKSPACE, ENDFILE, or REWIND statement.
k. A READ or WRITE statement not specifying core-to-core I/O.
l. A free-format designator containing an asterisk (*).

16. ADDITIONAL LANGUAGE CAPABILITIES

The B 7000/B 6000 series implementation of FORTRAN contains various language extensions. These include partial word manipulation using the CONCAT intrinsic (chapter 15); allowing arbitrary expressions in the DO statement (chapter 8) and as subscripts (chapter 5); and allowing recursion (chapter 15).

A FORTRAN program may reference subprogram units written in B 7000/B 6000 series ALGOL or COBOL. The subprogram units referenced must be separately-compiled and bound to the FORTRAN program.

An executing FORTRAN program may initiate independent programs through the ZIP WITH statement.

SEPARATELY-COMPILED PROGRAM UNITS

A B 7000/B 6000 series FORTRAN program may make use of the features available to the B 7000/B 6000 series ALGOL and COBOL languages. This process consists of the separate compilation of main program and subprograms in various languages to library, the binding together of these units into a program, and the execution of this program. This discussion will not attempt to explain the details of this process, but it will give an example how the FORTRAN language uses this system feature. For more detail, see the System Binder Reference manual, Form Number 5001456.

The following five program decks exemplify the use of the binder program to extend the FORTRAN language to make use of extended ALGOL constructs:

NOTE
<I> represents an invalid character in card column 1.

Deck 1:

```
<I>BEGIN JOB DECK1;
COMPILE FORT/HOST FORTRAN LIBRARY; FORTRAN DATA
      DIMENSION X(1), Y(1)
      X(1)='3ABCDE'
      Y(1)='MENTAT'
      WRITE(6,10)X,Y
   10 FORMAT(2(X,A6))
      CALL SUB(X,Y)
C***CALL EXCHANGE SUBROUTINE (FORTRAN)
      WRITE(6,10)X,Y
      CALL SUBA(X,Y)
C***CALL PARTIAL EXCHANGE PROCEDURE(ALGOL)
      WRITE(6,10)X,Y
      STOP
      END
<I>END JOB
```

Deck 2:

```
<I>BEGIN JOB DECK2;
COMPILE FTEST/FORT FORTRAN LIBRARY; FORTRAN DATA
$SET SEPARATE
      SUBROUTINE SUB(A,B)
      DIMENSION A(1),B(1)
C***SUBROUTINE EXCHANGING A AND B VALUES
      C=A(1)
      A(1) = B(1)
      B(1)=C
      RETURN
      END
<I>END JOB
```

Deck 3:

```
<I>BEGIN JOB DECK3;
COMPILE FTEST/ALG ALGOL LIBRARY; ALGOL DATA
   PROCEDURE SUBA(A,B); ARRAY A.B[*];
   %  SUBROUTINE REPLACING FIRST CHARS. IN A
   %  BY (1ST CHAR. IN B) CHARACTERS IN B
   BEGIN
      POINTER P,Q;
      LABEL XIT;
      P:=POINTER(A); Q:=POINTER(B);
      IF Q LSS "0" OR Q GRR "5" THEN GO XIT
      ELSE REPLACE P BY Q+1 FOR INTEGER (Q,1);
   XIT: END.
<I>END JOB
```

Deck 4:

```
<I>BEGIN JOB DECK4;
BIND BOUND/PROGM WITH BINDER FOR LIBRARY;
BINDER FILE HOST(TITLE=FORT/HOST);
BINDER EBCDIC
$SET LIST
BIND = FROM FTEST/=;
<I>END JOB
```

Deck 5:

```
<I>BEGIN JOB DECK5;
<I>RUN BOUND/PROGM
<I>END JOB
```

Resultant Output:

```
3ABCDE MENTAT
MENTAT 3ABCDE
ABCTAT 3ABCDE
```

This example exhibits the binding of a FORTRAN subroutine and an ALGOL procedure to a FORTRAN main program. The FORTRAN subroutine performs an argument exchange and the ALGOL procedure performs a partial exchange using pointer expressions. Thus, the main program is able to employ the B 7000/B 6000 series ALGOL pointer facility by merely calling a bound-in ALGOL procedure.

INDEPENDENT PROCESS INITIATION

The executable ZIP WITH statement initiates independent programs at a specific point within a FORTRAN program.

The proper format for the ZIP WITH statement is:

1. ZIP WITH a 2. ZIP WITH e
where e is a file designator, and a is an array name.

The ZIP WITH statement causes the MCP to activate a library program. The information, contained in the array indicated by the array identifier or the file referenced by the file designator, is used as control and program card images. Thus, a program may be initiated completely independent of the initiating program. For information concerning system control cards, see B 6700/B 7700 System Software Handbook, Form Number 5000722 and the Work Flow Language Reference Manual, Form Number 500155.

ZIP WITH Array

A ZIP WITH statement may be executed with an array identifier as its argument. This array must contain a string of EBCDIC characters representing control statement information as it normally appears on card images. The first character of the first word of this array must be a blank or a "?" and initiates the control statement information.

The subsequent control statements must be separated by single semicolons (;). The last control statement must be terminated by a semicolon followed by the word "END" and a period (.), in that order. The array involved is processed as one card image, but it may include more than 72 characters. Each array element contains six characters (or 12 characters in the case of double precision arrays).

When the ZIP WITH statement is executed, the MCP examines the contents of the array for validity and prints a message on the Operator Display Terminal if any errors are detected in this information. If no errors are detected, the control statement information is obeyed. After the execution of the ZIP WITH statement, program control passes to the next statement in the executing program, with any programs initiated by the ZIP WITH statement executing independently.

An example of a FORTRAN program using the ZIP WITH array statement is presented here:

```
      DIMENSION B(3), C(4), D(2,2)
      DATA B/'?RUN ZIP/TEST;END.'/
 1    WRITE(6,2)B
 2    FORMAT(X,3A6)
 3    ZIP WITH B
      C(1)='?RUN M'
      C(2)='AR/VEL'
      C(3)='(6);  '
      C(4)='END.  '
 4    WRITE(6,5)  C
 5    FORMAT(X,4A6)
 8    ZIP WITH C
      D(1,1)=' RUN P'
      D(2,1)='ROGRAM'
      D(1,2)='/ONE; '
      D(2,2)='END.  '
 9    WRITE(6,5)D
10    ZIP WITH D
      STOP
      END
```

This sample program contains three examples of the use of the ZIP WITH array statement.

The control statements: "?RUN ZIP/TEST;END." are placed via a DATA statement into array B. The printed output resulting from the execution of the statement labeled 1 is:

```
?RUN ZIP/TEST;END.
```

When the statement labeled 3 is executed, the library program ZIP/TEST is initiated.

The next statements fill the array C with the control statements: "?RUN MAR/VEL(6); END.". The printed output resulting from the execution of the statement labeled 4 is:

```
?RUN MAR/VEL(6);  END.
```

When statement 8 is executed, the library program MAR/VEL is initiated with the constant 6 passed as a parameter. The "?" in the present example could be replaced with a blank for the successful execution of the ZIP WITH statement, but it could be useful in identifying arrays containing control statement information within printed output. (Only one such question mark may appear in such an array.)

The two-dimensional array D is filled with the control statement: " RUN PROGRAM/ONE; END. ". Since a two-dimensional FORTRAN array such as this is stored internally by columns as a one-dimensional array, the control statement information must be stored sequentially in the array elements in such a way as to position the information correctly in the corresponding one-dimensional internal array. When the statement labeled 9 is executed, the printed output is:

```
RUN PROGRAM/ONE; END.
```

When statement number 10 is executed, the library program PROGRAM/ONE is initiated.

ZIP WITH File

A ZIP WITH statement may be executed with a file designator as its argument. The file designator must be an arithmetic expression corresponding to the internal file name of an existing file. The designated file must contain a string of characters representing control, program, and data information, appearing as it normally appears on card images. The first character in this file must be a "?"; this character initiates the control statement information.

The subsequent control statements may be separated by question marks or semicolons, but the file must adhere to pseudo reader formats (see below). All control card images should be EBCDIC records complying with standard control card syntax. If any other records within the file are BCL, all records following the ?BCL record should be BCL coded up to and including the END card image record or the next control card image for stacked decks. Following this latter card, all subsequent control cards must again be EBCDIC, and so on for stacked decks. Thus, program data may be interspersed among the control statement information as in ordinary program card decks.

File Format

The files used must agree with pseudo readers as to record size and disk area (row) size. Presently, all such file records must be 15 words long and blocked so that there are two records per disk segment. The actual block size is otherwise unimportant except that it must have an area (row) size which is an integer multiple of seven disk segments (i.e., of 14 logical records or 210 words).

When a file is used as the argument of a ZIP WITH statement and that statement is executed, the file is passed to the MCP and the program then continues processing in sequence. If the referenced file is not a disk or pack file or if it is declared to be a disk or pack file but is not currently on disk or pack, then the ZIP WITH statement is effectively ignored. If the file is found, the contents are examined by the MCP for validity and the control statement information contained therein is executed, if no errors are found. The file is then removed from the system.

Example

An example of the creation and use of valid files as ZIP WITH statement arguments is given here.

Constructing the File

The execution of the following program constructs two disk files, ZIPDISK/ONE and ZIPDISK/TWO, which are suitable for use as ZIP WITH statement arguments in later FORTRAN programs:

```
FILE    10(TITLE="ZIPDISK/ONE",KIND=DISK,MAXRECSIZE=15,BLOCKSIZE=2,
     1       AREAS=1, AREASIZE=14, SAVEFACTOR=3)
FILE    11(TITLE="ZIPDISK/TWO",KIND=DISK,MAXRECSIZE=15, BLOCKSIZE=2,
     1       AREAS=1, AREASIZE=14, SAVEFACTOR=3)
        DIMENSION B(6),D(15)
        DATA BLANK/Z404040404040/
        B(1)='?RUN P'
        B(2)='ROGRAM'
        B(3)='/ONE;O'
        B(4)='PTION='
        B(5)='DSED; '
        B(6)='END.  '
  1     WRITE(6,2) B
  2     FORMAT(X,6A6)
  3     WRITE(10,4) B
  4     FORMAT(6A6)
  5     LOCK 10
        D(1)='?RUN J'
        D(2)='OB/SIX'
        D(3)=';DATA '
        D(4)='FILE5 '
        DO 6 I=5,15
  6     D(I)=BLANK
  7     WRITE(6,8)D
  8     FORMAT(X,15A6)
  9     WRITE(11,10) D
 10     FORMAT(15A6)
        C=1.2
 11     WRITE(6,13) C
 12     WRITE(11,13) C
 13     FORMAT(F4.1)
        D(1)='?END.  '
 14     WRITE(6,8) D
 15     WRITE(11,10) D
 16     LOCK 11
        STOP
        END
```

This program provides two examples of constructing disk files suitable for use as ZIP WITH statement arguments. One of the files contains only control statement information, while the other file contains program data in addition to control information.

The program uses two file declaration statements to declare the files to be constructed (see appendix B). The first file is given an internal file name of FILE10 and a TITLE of ZIPDISK/ONE. The second file declared is given an internal name of FILE11 and a TITLE of ZIPDISK/TWO. The KIND, MAXRECSIZE, BLOCKSIZE, AREAS, AREASIZE, and SAVEFACTOR assignments of both file declarations declare both files to be assigned to disk, have records 15 words in size, be blocked two records per disk segment, have an area (row) size of 210 words (14 records or seven disk segments), and be saved on disk for three days from the creation date.

Next, the control statement information: "?RUN PROGRAM/ONE; OPTION=DSED;END." is placed into the first record of file 10. Since this is the only information to be placed into this file, the END statement is followed by a period. This period will terminate the MCP scan of this information when the ZIP WITH statement is executed, so the contents of the other 204 words of this file may be ignored. The statement labeled 3 places the desired information in the file, and the LOCK statement (statement 5) secures the file within the directory to be accessed in later programs.

File 11 is then filled with information. This file contains both control statement information and data of the form:

```
?RUN JOB/SIX;DATA FILE5
1.2
?END.JOB/SIX;DATA FILE5
```

where each line represents one file record (as printed by statements 7, 11, and 14). The MCP scan is terminated by the period following the END statement. The information in the third record following the period is ignored, as is all information in the following 11 records of this file.

Statements 9, 12, and 15 place the information in the file, and the LOCK statement (statement 16) secures the file within the directory to be accessed in later programs.

Use of the File

An example of a program accessing these files is given here:

```
<I>BEGIN JOB ZIPTEST;
COMPILE ZIP/RUN FORTRAN;
FILE FILE4(TITLE=ZIPDISK/ONE,KIND=DISK);
FILE FILE8(TITLE=ZIPDISK/TWO,KIND=DISK);
FORTRAN DATA
1       ZIP WITH 4
2       ZIP WITH 8
        STOP
        END
<I>END JOB
```

This deck provides an example of a FORTRAN job using the ZIP WITH file statement. Two file control cards are employed to equate the internal file names FILE4 and FILE8 to the TITLEs ZIPDISK/ONE and ZIPDISK/TWO, respectively. These two files are then used as arguments in ZIP WITH statements.

When the two ZIP WITH statements are executed, the contents of the two files are passed to the MCP for processing and the files are removed from the system. The execution of statement 1 results in the execution of the library program entitled PROGRAM/ONE with the option DSED set. The execution of statement 2 results in the execution of the library program entitled JOB/SIX with the data record "1.2" prepared to be read from a file with a TITLE of FILE5. These two initiated jobs are free to run independently of the present program. The initiating program is free to reach EOJ without having an effect on the execution of PROGRAM/ONE or JOB/SIX.

17. DEBUGGING AND TIMING AIDS

DEBUG statements assist in isolating programming errors and in identifying heavily executed program sections so that more efficient programming techniques may be applied to these sections. All such statements contain the characters DEBUG in columns 1 through 5 and conform to the usual FORTRAN statement formulation rules. The output from all DEBUG statements may be deleted by compiling the program with the compiler option OMITDEBUG set. Thus, DEBUG statements can become a permanent part of a symbolic program, included while the program is being checked out and excluded while the program is in production form.

DEBUG MONITOR STATEMENT

The DEBUG MONITOR statement provides the ability to monitor the changes in value of simple variables and array elements within a program unit. The proper syntax for the DEBUG MONITOR statement is:

DEBUG MONITOR (n) m
where DEBUG appears in columns 1 through 5; MONITOR begins in or after column 7; n is a constant file designator; and m is a monitor list.

The file designator must be an integer constant from 1 to 99 that specifies the file to which the monitoring output is to be directed.

The monitor list is a series of variables, array names, array elements, and subscripts separated by commas. If the list contains an array name without subscripts, then every element of that array is monitored. If the array name is followed by a series of one or more subscripts enclosed in parentheses and separated by commas, then only those elements of the array are monitored. An individual subscript is either a constant or a simple variable. It may not be any other arithmetic expression. Only the first appearance of a simple variable or an array name in a monitor list is processed. All successive appearances are ignored, and a warning message is issued.

At execution time, whenever the value of a monitored variable is changed, diagnostic information, including the old and new values of the variable, is written to the designated file. Monitoring action takes place for a simple variable whose value is changed through an assignment statement, a READ statement, or through being passed as an actual argument to a subprogram. Monitoring action occurs for an array element only when its value is changed through an assignment statement.

There can be more than one DEBUG MONITOR statement per program unit, and DEBUG MONITOR statements can be continued through continuation cards. Arrays must be dimensioned prior to appearing in a DEBUG MONITOR statement.

Example

```
      DIMENSION A(5), B(5), C(5)
DEBUG MONITOR (6) X, A(1), (I), (5), B, Y, C, Z
      I=3
      A(3)=1
      B(4)=2
      X=3
      A(4)=4
      READ (5,/) Y, A(1)
      CALL SUB(Z,C)
      STOP
      END
      SUBROUTINE SUB(T,V)
      DIMENSION V(5)
      V(1)=4
      T=3
      RETURN
      END
```

With input data "5,5", the monitoring output not including location information for this example will be:

```
A(3)=0:=1
B(4)=0:=2
X=0:=3
Y=0:=5
Z=0:=3
```

DEBUG DUMP STATEMENT

The DEBUG DUMP statement provides the ability to dump the values of a list of simple variables and arrays within a program unit when program control passes to a label. The proper form for the DEBUG DUMP statement is:

1. DEBUG DUMP(n) s,f(j=k,m)/d 2. DEBUG DUMP(n) s,f/d 3. DEBUG DUMP(n) s(j=k,m)/d 4. DEBUG DUMP(n) s/d
where DEBUG appears in columns 1 through 5; DUMP begins in or after column 7; n is a file designator; s is a statement label; f is a positive integer denoting frequency; (j=k,m) is a control condition with j the control variable and k and m positive integers denoting lower and upper bounds, respectively; and d is a list of simple variables and array names separated by commas.

The file designator, n, is an integer constant from 1 to 99 that specifies the file to which the diagnostic information is to be directed.

When program control passes to the specified label, dump action depends, in part, on the frequency f; the diagnostic output will occur every fth time program control passes to the label. If frequency f is not specified, it will have a default value of 1.

The control condition also sets restrictions on when dump action will occur by designating a program control variable j with lower and upper bounds k and m. As long as j is bounded by k and m, then diagnostic output will be produced upon transfer to label s, subject also to the frequency f. If the lower bound k is not specified, it will have a default value of 1. If upper bound m is not specified, there will be no upper bound on the control variable j. If a control condition is not specified, dump action will be dependent only on the frequency f and the number of times execution control passes to label s.

There may be more than one DEBUG DUMP statement per program unit, and DEBUG DUMP statements may be continued through continuation cards. Each such statement must follow the dimensioning of all arrays and precede the occurrence of all labels referenced by it.

Use of the DEBUG DUMP statement together with the options OWN and SEPARATE set in a subroutine is not allowed.

Use of a formal parameter in the list of simple variables and array names is illegal.

Example

```
DEBUG DUMP (6) 10,2 (K=1,15)/M
      DO 10 M=1,6
      K=M+10
   10 CONTINUE
      STOP
      END
```

The following is the diagnostic output for this example:

```
10 FROM 0002:000C:1
M=2
10 FROM 0002:000C:1
M=4,
```

DEBUG PROGRAMDUMP STATEMENT

The DEBUG PROGRAMDUMP statement provides a "snapshot" of the program stack by invoking the MCP procedure PROGRAMDUMP at the point in the program where the statement is executed.

The proper format for the DEBUG PROGRAMDUMP statement is:

DEBUG PROGRAMDUMP
where DEBUG appears in columns 1 through 5; and PROGRAMDUMP begins in or after column 7.

The DEBUG PROGRAMDUMP statement causes program execution to be temporarily suspended while the contents of the program's stack are dumped. The program will then be resumed without affecting the program's execution. The options available to the MCP PROGRAMDUMP procedure (i.e., ARRAYS, CODE, FILES, etc.) can be invoked by using an OPTION control card specifying the desired options.

DEBUG STATISTICS STATEMENT

The DEBUG STATISTICS statement produces a frequency/timing study of a program's execution. Heavily executed areas of a program can thus be isolated and perhaps improved through more efficient programming.

The proper format for the DEBUG STATISTICS statement is:

```
1.    DEBUG STATISTICS(m)
2.    DEBUG STATISTICS(m) BLOCKS
3.    DEBUG STATISTICS DIAGNOSTICS
4.    DEBUG STATISTICS DIAGNOSTICS BLOCKS
5.    DEBUG STATISTICS NO
```

where DEBUG appears in columns 1 through 5; STATISTICS begins in or after column 7; and m is a constant file designator.

The file designator m, must be an integer constant from 1 to 99 that specifies the file to which the statistics output is to be directed. DIAGNOSTICS may be substituted for the file designator; the system diagnostic file will then be used for the STATISTICS output.

Two options are available: subprogram times and counts only, or times and counts for each logical segment or block in each subprogram. The first of these options is specified by leaving the columns following the file designator blank, while the second is specified by following the file designator with BLOCKS. The statement DEBUG STATISTICS NO resets STATISTICS.

Any time the program executes a STOP or DEBUG DUMP STATISTICS statement or the program's exceptionevent (<mixid>HI) is caused, the accumulated statistics are output to the indicated file. The subprogram and block (if applicable) being executed at the time the statistics output occurs are flagged with an asterisk to indicate that the elapsed time for this program and block includes the time necessary to output the statistics, and thus may be slightly inaccurate.

When the BLOCKS option is specified, program execution time increases substantially (30 to 50 percent). The DEBUG STATISTICS statement stays in effect until reset by the statement DEBUG STATISTICS NO.

It is not possible to bind programs with DEBUG STATISTICS. If a DEBUG STATISTICS statement is encountered, no bind information will be generated. If SEPARATE, LIBRARY, or AUTOBIND is set, an error will be flagged; otherwise, if NOBINDINFO is not already set, a warning will state that no bind information will be generated.

DEBUG DUMP STATISTICS STATEMENT

The DEBUG DUMP STATISTICS statement causes the accumulated timing/frequency statistics to be output (see DEBUG STATISTICS statement in this chapter).

The proper format for the DEBUG DUMP STATISTICS statement is:

DEBUG DUMP STATISTICS
where DEBUG must appear in columns 1 through 5; and DUMP must begin in or after column 7.

DEBUG TRACE STATEMENT

The DEBUG TRACE statement traces a program's flow by printing out all sequence numbers and labels of specified FORTRAN statements and subprogram calls in execution order. Comparator action may also be specified to detect deviations in predetermined program flow and to have program dumps taken if a mismatch should occur.

The proper format for the DEBUG TRACE statement is:

1. DEBUG TRACE (f,c)t 2. DEBUG TRACE (f)t Text Deleted
where DEBUG appears in columns 1 through 5; TRACE begins in or after column 7; f and c are file designators; and t is a list of TRACE options separated by commas.

The file designators, f and c, must be integer constants from 1 to 99. f is the output file and c is the comparator file. The value of c may specify either a disk, pack, or tape file. If c is not specified, then the trace output is written on the file f. If c is specified, then f is not used for the trace output. Instead, a record is read from file c and compared to the trace record that would have been output to f, if c had not been specified. If the records compare as identical, then no action is taken. If they do not compare, then a message is output to file f; and if DUMP (or PROGRAMDUMP) is specified in the trace list, then a program dump is taken.

Trace List

Specifications in the trace list may be DUMP, NODUMP, PROGRAMDUMP, NOPROGRAMDUMP, CALLS, NONE, s, and #s, where s is defined to be any of the following: ALL, IO, GO, GO TO, READ, WRITE, BACKSPACE, REWIND, PRINT, PUNCH, =, CALL, and CONTINUE. DUMP or PROGRAMDUMP specifies that a program dump is to occur if the record comparison with file c fails. NODUMP or NOPROGRAMDUMP resets this option. CALL traces all subroutine and function entries and exits; however, intrinsics calls are not traced. NONE resets all requests preceding it. Otherwise, each trace specification is processed accumulatively. None of these options has any meaning if file c has not been specified.

The specification s is used to indicate that tracing is desired on an executable statement or on a class of executable statements. If preceded by a pound sign (#) then only labelled statements of the specified class are traced. An explanation of the various statement classes follows:

ALL	indicates all executable statements including subprogram entries and exits.
IO	indicates all input/output statements.
GO, GO TO	indicates all GO TO statements.
PRINT, PUNCH, READ, WRITE, CALL, CONTINUE, BACKSPACE, REWIND	indicate the specified statements.
=	indicates all assignment and DO statements.

18. FORTRAN COMPILER

INTRODUCTION

The B 7000/B 6000 series FORTRAN compiler is written in B 7000/B 6000 Extended ALGOL. This program compiles source programs producing object programs which may be executed on the B 7000/B 6000 system. The FORTRAN compiler and the object programs generated by this compiler are designed to operate under control of the B 7000/B 6000 Master Control Program (MCP) and with an intrinsic file loaded on the system.

COMPILER FILES

The compiler files are illustrated in figure 18-1.

Input Files

The primary compiler input file is a file with the internal name CARD; the secondary input file is a file with the internal name TAPE. The presence of the primary file CARD is required for each compilation; the presence of the secondary file TAPE is optional. The file TAPE is used as input only when the MERGE compiler option is set in the primary files. Both the CARD file and the TAPE file may be label-equated via FILE system control card to change their file attributes.

Additional files may be input to this compiler through use of the INCLUDE compiler control statement (see chapter 19).

Output Files

Output files produced by this compiler include the object code file, an updated symbolic file, a listing, and an error message file.

The object code file has the internal name CODE and is saved after the compilation unless the COMPILE system control card specifies compilation for syntax only or syntax errors are detected in the source language input (see INPUT DECKS, this chapter). If compile-and-go is specified by the COMPILE card, then the object code file is discarded after the code is executed.

The TITLE of the saved code file is identical to the program name appearing on the COMPILE card except in the case of separately-compiled subprograms. When a subprogram is compiled separately, the TITLE of the resultant object code file consists of the program name appearing on the COMPILE card with the rightmost identifier in the program name replaced by the subprogram name. If there is only one identifier in the program name, the subprogram name is assigned as the code file TITLE.

The updated symbolic file is generated only if the compiler option NEW is set. This file contains the compilation source input or a selected portion of this input as specified by the states of the NEW and INCLNEW compiler options and may be used for a succeeding compilation. This output file has the internal file name NEWTAPE and the TITLE FORSYM.

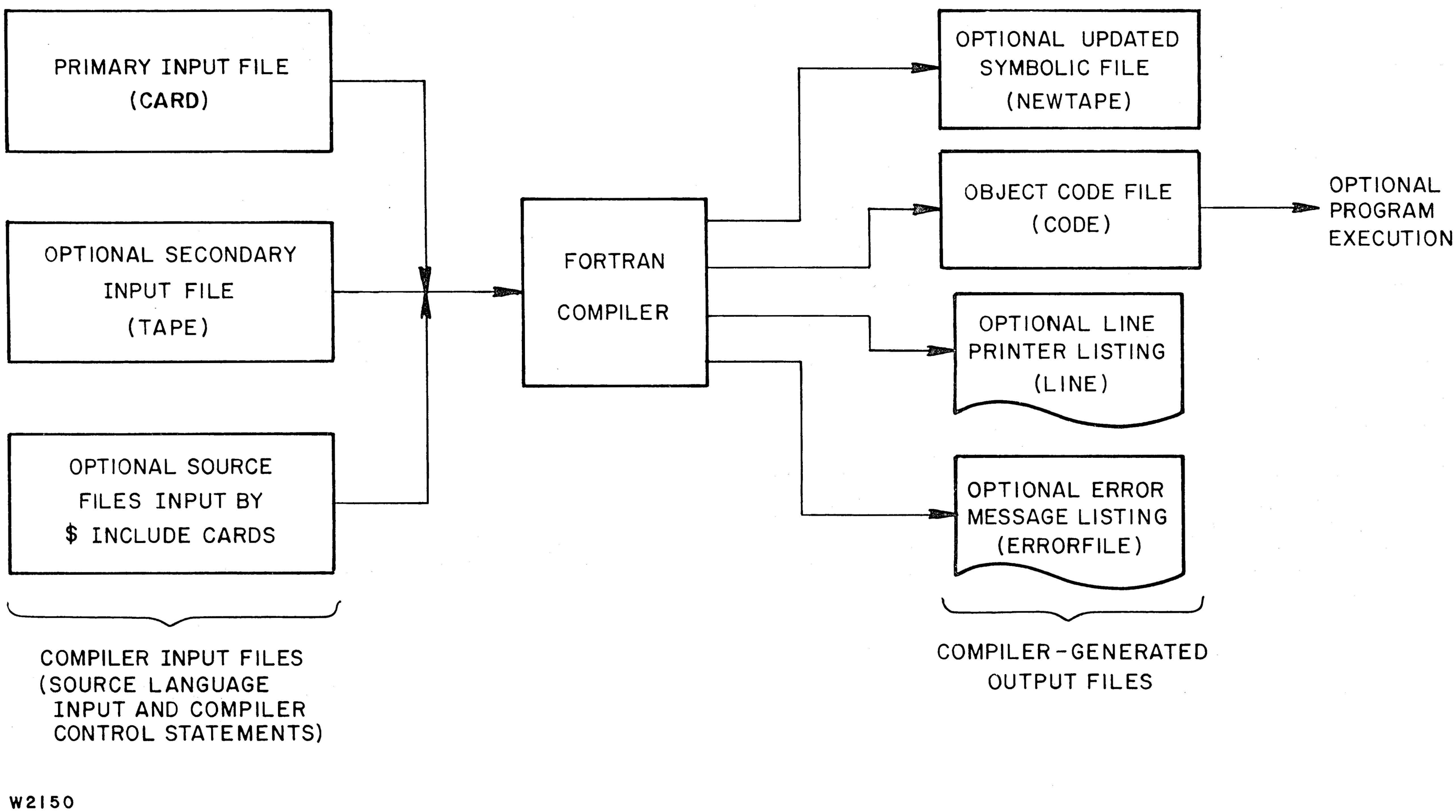


Figure 18-1. FORTRAN Compilation System

The listing is a file that is created when the compiler option LIST or TIME is set. The LIST option is set by default unless the compilation is initiated through CANDE. The file has the internal name LINE, and bears the following information:

Compiler version.
Date and time of compilation.
Code file title.
Source and compiler control statements input to the compiler.
Code segmentation information.
Error messages and error count.
Warning messages.
A listing of the code generated.
A list of stack addresses of data items used in the program.
Number of input card images scanned.
Elapsed compilation time.
Processing time required for the compilation.
Estimated size of the program's working stack when the program is executed.
Estimated size of the program files and related storage.
Estimated core storage requirement.
Total number of words of object code generated.
Number of words of array storage required for the execution of the program.
Number of segments in the generated program.
Number of disk segments required for the code file.
The date the compiler was compiled.

Depending upon the setting of various compiler options, this listing may contain some or all of the items listed above. Card images from file CARD are denoted on the listing by a "C" after the card image. Card images from file TAPE are denoted by a "T". Card images from an INCLUDEd file are denoted by an integer from 1 to 5 indicating the level of INCLUDE nesting. A "P" denotes a CARD image that has superseded ("patched") a TAPE card image.

The output error message file with the internal file name and TITLE of ERRORFILE is created when the ERRLIST compiler option is set. This file is normally used for compilations initiated through CANDE, in which case ERRLIST is set by default and the ERRORFILE compiler file is assigned to the remote device involved. When a syntax error is detected, the offending card image is written on this file, and an error message is written on the following line of text.

Compiler File Table

The FORTRAN compiler input and output files are listed in table 18-1, which provides information concerning the configuration of each file. The attributes of any of these files may be changed through use of FILE system control card images.

NOTE

If either of the attributes UNITS and MAXRECSIZE is file-equated in the compiler files, the compiler input/output is undefined.

Table 18-1. FORTRAN Compiler Files

Internal Name	Purpose	Kind	Code	Record Size	Block Size	Comments
CARD	Primary Input File	READER	EBCDIC	14 Words	May be Blocked or Unblocked	Required for each compilation. CANDE file is equated to this file automatically for compilations initiated through CANDE.
			BCL or BCD	10 Words		
TAPE	Secondary Input File	DISK	EBCDIC	14 Words	May be Blocked or Unblocked	Optional file. Selected as input by setting MERGE compiler option.
			BCL	10 Words		
May be Specified On INCLUDE Compiler Control Card	Library Include File	DISK	EBCDIC	14 Words	May be Blocked or Unblocked	Optional input file opened by $INCLUDE card bearing appropriate file TITLE or internal file name. Five levels (maximum) of nesting permitted (see discussion of INCLUDE option).
			BCL	10 Words		
CODE	Object Code File	DISK	Hexadecimal	30 Words	270 Words	Generated object code file. Saved or discarded and assigned a TITLE as indicated in text.
NEWTAPE	Updated Symbolic Output File	DISK	EBCDIC	14 Words	420 Words	Optional output file produced when NEW compiler option is set. This file contains portions of the source input, is assigned a default TITLE of FORSYM, and is suitable for use in a later compilation.
LINE	Listing	PRINTER	EBCDIC	20 Words	20 Words	Optional file produced when either the compiler option LIST or TIME is set. Label-equatable to REMOTE.
ERRORFILE	Error Listing Output File	PRINTER	EBCDIC	14 Words	14 Words	Error listing file produced when ERRLIST compiler option is set. Contains card images and error messages.

INPUT DECKS

The following discussion applies to FORTRAN compilations that do not utilize the batch facility discussed later in this chapter and are not initiated through CANDE.

For compilation, the only required card images are:

a. A BEGIN JOB system control card image.
b. A COMPILE system control card image to initiate the compilation and specify the type of compilation to be performed.
c. An END JOB system control card image.

System control card images may be placed after the COMPILE card image to specify compilation parameters and execution parameters, and to alter the attributes of files (see the Work Flow Language Reference Manual, form no. 5001555).

If the deck is to contain source input, the system control card images are followed by a, FORTRAN (or COMPILER) EBCDIC (or DATA or BCL) system control card image, depending on the form of the source input (EBCDIC and DATA are synonymous), and this card image is followed by the source input and compiler control cards. If the source input is in BCD, the source input is preceded by a BCL system control card image and a BCD compiler control card image precedes the first BCD-coded source input card image. If the deck is to contain input data, then the source card images are followed by data decks consisting of an EBCDIC, DATA, BCL, or BINARY system control card images, depending on the form of the data card images, bearing the TITLE of the data file, followed by the card images bearing the data. If this data is coded in BINARY format, then the data is terminated by a binary end (BEND) card.

Four kinds of FORTRAN compilation are available to the B 7000/B 6000 user. They are:

Compile for Syntax
Compile for Library
Compile and Go
Compile for Library and Go

Compile for Syntax

Compile for syntax allows a program to be compiled for syntax-checking purposes only. The object code file is not saved after the compilation.

The form of the COMPILE system control card image for a compile for syntax is:

```
COMPILE <program name> FORTRAN SYNTAX;
```

Compile for Library

Compile for library indicates that the object code file resulting from a successful compilation is to be saved. Execution of this code file is not initiated.

The form of the COMPILE system control card image for a compile for library is:

```
COMPILE <program name> FORTRAN LIBRARY;
```

The code file resulting from a compile for library may be executed via a RUN or EXECUTE system control card. For execution, the only required card images are:

a. A BEGIN JOB system control card image.
b. A RUN or EXECUTE system control card image bearing the TITLE of the code file to be executed.
c. An END JOB system control card image.

System control card images may be placed after the RUN or EXECUTE card image to specify execution parameters and to alter file attributes. If the deck is to contain input data, the system control card images are followed by data decks consisting of an EBCDIC, DATA, BCL, or BINARY system control card image bearing the TITLE of the data file, followed by the data card images. If this data is coded in BINARY, the data is terminated by a binary end (BEND) card.

Compile and Go

Compile and go indicates that the object code file resulting from a successful compilation is to be immediately executed and then discarded.

The two forms of the COMPILE system control card image for a compile and go are:

```
COMPILE <program name> FORTRAN;
COMPILE <program name> FORTRAN GO;
```

Compile for Library and Go

Compile for library and go indicates that the object code file resulting from a successful compilation is to be immediately executed and saved.

The form of the COMPILE system control card image for a compile for library and go is

```
COMPILE <program name> FORTRAN LIBRARY GO;
```

BATCH FACILITY

In certain situations, such as an educational environment, it may be possible to submit as a group a number of programs having the following characteristics:

a. The programs are to be compiled for syntax or compiled and executed.
b. If a program has no syntax errors and execution is requested, the execution time is relatively small.
c. Each program requires no more than one printer file and no more than one card reader file.
d. No program requires operator intervention.

The batch facility is available so that groups of such programs may share the cost of required system overhead functions normally associated with one job.

Deck Set-Up

Each individual job is set up in one of the following two ways.

Compile and Execute

```
$ JOB <optional job title>
    <source input>
$ ENTRY
    <data, if any>
```

The beginning of the job is indicated by the $JOB card image. If the optional job title appears, it is used to identify the printed output produced by the compiler for this job. Following the $JOB card image is the source input, which may include compiler control cards.

The $ENTRY card image indicates both the end of the source input and that the program is to be executed following successful compilation. If the program requires a data deck, this data deck follows the $ENTRY card image.

Compile for Syntax

```
$ JOB <optional job title>
    <source input>
```

The absence of a $ENTRY card image indicates that the program is not to be executed.

In addition, there are three compiler options: PROCESSTIME, IOTIME, and PRINTLIMIT, which may be used in batch mode (see chapter 19). If PROCESSTIME, IOTIME, or PRINTLIMIT is set prior to the first $JOB card image, the values apply to all of the programs. If PROCESSTIME, IOTIME, or PRINTLIMIT is set following the $JOB card image, the values apply only to that program. PROCESSTIME has a default setting of 5.

Using the Batch Facility

The job decks are placed together to form a single source input deck. This deck is preceded by system control card images invoking the FORTRAN compiler and followed by the END system control card. This deck is then processed by the system. The output listing contains the compilation listing for each program followed by its corresponding execution output.

Example of a batch of three FORTRAN jobs:

```
<I>BEGIN JOB BATCH/TEST;
COMPILE BATCH/JOB FORTRAN;FORTRAN DATA
$PROCESSTIME = 5.5 IOTIME = 1

$JOB                JOB # 1
$SET FREE RESET SINGLE
REAL A(10)
READ 1,A
PRINT 1,A
1     FORMAT(10A6)
END
$ENTRY
THIS IS A DATA CARD FOR THE ABOVE JOB

$JOB                JOB # 2
      WRITE(6,1) (SIN(3.14*1),  I=1, 20)
1     FORMAT(1H,  F13.4)
      I=X + + X       % SYNTAX ERROR
      END
$ENTRY



$JOB                JOB # 3
C***          COMPILE FOR SYNTAX
      SUBROUTINE XXXXXX
      DO 10 J=1,10
10    CONTINUE      % A SILLY LOOP
      RETURN
      END
      X=Y
$SET  CODE
      CALL XXXXXX
      STOP
      END     % THERE IS NO $ENTRY CARD -- JUST A COMPILE FOR SYNTAX

<1>END JOB
```

No job in this three-job batch will be allowed more than 5.5 seconds of processor time, nor more than 1 second of I/O time. Job #1 will compile correctly and run, using the one-card data deck immediately following its $ENTRY card image. Job #2 has a syntax error, but would have run had it been error-free. Job #3, despite being syntactically correct, will not run.

Restrictions

Each of these restrictions must be met in order for a job to be a candidate for successful processing by the batch FORTRAN compiler.

a. The PRINTER file may not be explicitly opened, closed, or have its attributes changed. Attempts to do this will terminate the job.
b. Binding is not allowed. No compiler option pertinent to binding is valid.
c. Missing subroutines or functions are fatal errors.
d. The PAUSE statement is allowed but has no effect.
e. Any program action requiring operator intervention is a fatal execution error.
f. Only one printer file is allowed. If two or more printer files are declared, their output will be joined into a single file. Similarly, at most one reader file is allowed.
g. The following compiler options are invalid or ignored when using the batch facilities: BCD, END, ERRLIST, INSTALLATION, LEVEL, LIBRARY, LINEINFO, NEW, NEWSEGMENT, SEPARATE, TIME, XREF, XREFFILES.
h. All job decks must be punched with the same card codes (EBCDIC, BCD, or BCL). The system control card preceding the job decks (DATA, EBCDIC, or BCL) must indicate the proper card code.

COMPILATIONS THROUGH CANDE

A FORTRAN compilation initiated using the Command and Edit (CANDE) language differs from other FORTRAN compilations in the following ways:

a. Compiler option LIST is reset by default.
b. Compiler options FREE, ERRLIST, and LINEINFO are set by default.
c. Compiler option LIMIT has a default value of 10.
d. The ERRORFILE into which syntax errors are written is equated to the remote terminal initiating the compilation.
e. Unless otherwise specified.
   1. The CARD file will be equated to the work file or specified disk file.
   2. No TAPE file is used.
   3. The object code file created by compiling the file <*filename*> is <*usercode*> OBJECT/<*filename*>.
f. All compilations are compile for library. Execution of the object code may be initiated by either of two CANDE commands: RUN or EXECUTE.

LANGUAGE COMPATIBILITY

FORTRAN IV, Level H

Source programs written in FORTRAN IV, Level H, are acceptable as input to this compiler, with the exceptions stated below. These exceptions refer to features of the B 7000/B 6000 series FORTRAN compiler.

a. The number of continuation card images is not limited.
b. The characteristics of allowable values are as follows:
   1) Single Precision: an integer or the mantissa of a real value may be from one to 11 decimal digits. The approximate total range of a real value is 0 and from 8.758E-47 to 4.314E68.
   2) Double Precision: the mantissa of a double precision value may be from one to 23 decimal digits. The approximate total range of a double precision value is 0 and 1.939D-29581 to 1.949D29603.
   3) Complex: The characteristics of each component of a complex value are the same as those of a single precision number. A component of a complex number that is outside the range of a single precision number will be rounded or disallowed, as necessary. Each double precision complex intrinsic identifier is treated as a synonym for the corresponding single precision complex intrinsic identifier.
c. Variables local to a subroutine or function subprogram will not retain their values between calls on that subprogram unless the compiler option OWN is set for the subprogram (see also point m).
d. The indexing parameters of a DO statement may be changed within the range of the DO loop.
e. Transfer out of and into a DO loop is unrestricted.
f. A "STOP n" statement is allowed, but n is ignored.
g. The DEFINE FILE statement is not recognized. The functions of this statement are included in the file declaration and FILE system control card.
h. An end-of-file action specifier (,END = *<label>)* is allowed in a random I/O (direct access) statement.
i. The optional asterisk specification in a type statement is allowed, but it will have no effect except that REAL *8 will designate double precision type.
j. Array elements may appear in the expression in an arithmetic statement function.
k. DATA and EQUIVALENCE statements may contain array names as elements of the list of data items affected by the statement. Such elements are not restricted to array elements and simple variables.
l. Recursion is allowed; that is, a subprogram may reference itself either directly, indirectly, or through an entry point.

m. An entry into a subprogram at any point results in initialization of only those formal arguments included in the formal argument list of the FUNCTION, SUBROUTINE, or ENTRY statement through which the subprogram was entered. Of the remaining formal arguments, those which are simple variables are reset to zero, and those which are arrays are nonexistent. All local variables not appearing in a DATA statement or initialized in a type statement are reset to zero.
n. Any in-line intrinsic specified in an EXTERNAL statement will not be expanded in-line. Such a function is assumed to be user-supplied and will be treated as a function subprogram.
o. Items in blank common may be initialized by a block data subprogram.
p. The COMMON statement in a block data subprogram need contain only those elements of the block up to and including the last element to be initialized.
q. Elements of common blocks may be initialized with any of the following: (1) initial value lists, (2) DATA statements in the main program, or (3) a block data subprogram.
r. Strings can be stored only as integer or real variables (CHARS characters per variable) or as double precision variables (2*CHARS characters per variable).

B 3500 FORTRAN

Source programs written in B 3500 FORTRAN are acceptable as input to this compiler, with the exceptions stated below. These exceptions refer to features of the B 7000/B 6000 series FORTRAN compiler.

a. Integer and real values are limited to 11 significant digits. Double precision values are limited to 23 significant digits.
b. Adjacent exponentiations are carried out in right-to-left order.
c. A "STOP n" statement is allowed, but n is ignored.
d. Simple variables in a formal parameter list are call by value unless enclosed in slashes.
e. Execution of a "PAUSE n" statement results in suspension of execution of the program and display of n. Execution may be continued with an "OK" message from the Operator Display Terminal.
f. The compiler options are different.
g. B 3500 extensions to ANSI FORTRAN such as the ALPHA type are not accepted.

ANSI FORTRAN

Source programs written in ANSI FORTRAN are acceptable as input to this FORTRAN compiler without exception.

Other FORTRAN Languages

The B 7000/B 6000 series FORTRAN compiler was designed to take advantage of the similarities among FORTRAN languages. However, since some language features are machine-dependent and since variations and extensions in implementation exist, no guarantees of compatibility can be made.

The language portion of this document should be used as a tool in rendering source programs written in other FORTRAN languages acceptable as input to the B 7000/B 6000 series FORTRAN compiler.

19. COMPILER OPTIONS

COMPILER CONTROL STATEMENTS

Compiler control statements control how FORTRAN processes source input and handles input errors. These control statements may also affect the compiler generated output. Compiler control statements are input on compiler control card images.

Compiler control information appears in columns 1 through 72 of the control card image. An eight digit sequence number may optionally appear in columns 73 through 80. Control card images may not be continued.

Each control card image is identified by a dollar sign ($) in column 1 of the card image or a blank in column 1 and a dollar sign in column 2. In addition to the dollar sign, compiler control information may consist of compiler option actions, compiler option names, and associated option arguments. Blanks separate compiler options and option actions. A blank may optionally appear between the dollar sign and the first compiler option or option action.

A compiler option may be either a parameter option or a standard option.

Parameter options are initiated by the appearance of the option name and any associated option arguments on a compiler control card image.

Standard options may have a state of SET or RESET. Each standard option has a default state and the state may be specified on a compiler control card image. A standard option which has a default state of RESET is initially assigned a 24 bit stack filled with zeros; a standard option which has a default state of SET is initially assigned a 24-bit stack with a 1 on top and zeros in the remaining positions. The top stack position denotes the state of the option at any time.

Compiler control statements assign a desired state to a compiler option or a group of compiler options. A compiler control statement, which applies to standard options, begins with an explicit or implicit option action, contains a list of standard options to which the option action is to apply, and ends when the next explicit option action is encountered on the same compile control card image or when a percent sign or column 72 of the card image is reached.

Option Actions

Explicit Option Action

An explicit option is one of the mnemonics:

```
SET
RESET
POP
```

If a compiler control statement begins with the option action SET, the standard options following the option action are assigned the state SET; the states of all other standard options will be unchanged. If the control statement begins with the option action RESET, the standard options following the option action are assigned the state RESET; the states of all other standard options will be unchanged. If the specified option action is POP, then the standard options indicated will revert to their immediately previous states (their states will become RESET if these options have not been changed from their default states). The states of all other standard options will be unchanged.

Any parameter option, except INCLUDE, may appear on a compiler control statement beginning with an explicit option action and the parameter option action will be initiated. The INCLUDE parameter option must be the only option on the compiler control statement.

Each SET option action causes the stacks allocated to the designated standard options to be pushed down one bit with a 1 placed on top. Each RESET option action causes the appropriate stacks to be pushed down one bit with a 0 placed on top. POP causes the stacks corresponding to the designated options to be popped up one bit, causing the associated options to revert to their immediately previous states. A maximum history of 24 states may be recorded.

An option affected by a particular compiler control statement will retain the indicated state for all input card images with sequence numbers greater than the sequence number on the compiler control card image, or for all physically succeeding input card images when all sequence number fields are blank, until another compiler control statement is encountered which affects the option. Within a card image, options are assigned states from left to right.

It is possible to assign the state of one compiler option as the state of another compiler option; for example:

```
$ SET SINGLE = LIST
```

The following are examples of compiler control statements employing the SET, RESET, and POP explicit option actions:

```
SET LIST SINGLE NEW INCLNEW
RESET VOID
POP NEW NEWSEQERR
```

The following compiler control card image:

```
$SET SINGLE LIST SEQ 10+5 RESET OWN BCD
```

Assigns the state SET to the options SINGLE, LIST, and SEQ (with the sequencing arguments of 10 and +5) and assigns the state RESET to the options OWN and BCD.

Implicit Option Action

An implicit option action is indicated when a compiler control statement begins with the name of a standard option. In this case, all standard options named in the control statement are assigned the state SET and all other standard options except ASCII, BCL, B5500, B5700, B7700, OWNARRAYS, SEGS, and SINGLE are assigned the state RESET.

When a compiler control card bears an implicit option action, then all 24 bits of each standard option stack except those for the options ASCII, BCL, BT5500, B5700, B7700, OWNARRAYS, SEGS, and SINGLE are reset and an explicit SET is performed on each indicated standard option. For example, after the appearance of this card:

```
$ CODE
```

the history stack for the CODE option contains a 1 in the top stack position and all zeros in the following positions. The history stacks for each of the other standard options, with the exceptions noted above, contain all zeros.

The appearance of a parameter option on a compiler control card with an implicit option action does not result in the resetting of any options.

The B 7000/B 6000 series FORTRAN compiler recognizes the following identifiers as valid compiler option names. A (p) denotes a parameter option, and a (u) denotes a standard option which is not reset when implicit option action is taken.

```
ASCII          (u)
AUTOBIND
BCD
BCL            (u)
BINDER         (p)
B5500          (u)
B5700          (u)
B7700          (u)
CHARS          (p)
CHECK
CODE
DBLTOSNGL
DEBUG
END            (p)
ENTRY          (p)
ERRLIST
FREE
```

FREENEW	
FREETAPE	
GRAPH	
INCLNEW	
INCLUDE	(p)
INSTALLATION	
IOTIME	(p)
JOB	(p)
Text Deleted	
LIBRARY	
LIMIT	(p)
LINEINFO	
LIST	
LONG	
MERGE	
MONITOR	
NEW	
NEWSEGMENT	(p)
NEWSEQERR	
NOBINDINFO	
NOWARN	
OMIT	
OMITDEBUG	
OPT	(p)
OWN	
OWNARRAY	(u)
PAGE	(p)
PRINTER	(p)
PRINTLIMIT	(p)
PROCESSTIME	(p)
READER	(p)
SEGMENTATION	(p)
SEGS	(u)
SEPARATE	
SEQ	
SEQERR	
SINGLE	(u)
STACK	
SUPRS	
TIME	
TRACE	
VECTORMODE	
VOID	
VOIDT	
XREF	
XREFFILES	
$	

The FORTRAN compiler options are discussed alphabetically in the following paragraphs. The default of the LIST option is SET and the default state of the balance of the standard options is RESET unless the compiler is called from CANDE, in which case the default state of the ERRLIST, FREE, and LINEINFO options is SET and the default state of the balance of the standard options is RESET.

ASCII (Reset)

The ASCII compiler option causes strings to be stored internally in ASCII code.

AUTOBIND (Reset)

The AUTOBIND compiler option combines the processes of compiling and program binding into one job. During compilation, the compiler produces a set of instructions to be passed to the binder. In many cases, these binder instructions are sufficient; however, the user may insert his own binder control card through the BINDER compiler option. The AUTOBIND option should be set or reset only once, at the beginning of a compilation.

During binding, the main program will serve as the host. At most one host may be compiled in a job along with any number of separate subprograms. If no eligible host is being compiled, a host must be specified. However, specifying a host will not override the use of the main program as the host.

Two types of control information may be passed to the binder when using the autobinding feature. Binder option cards (dollar cards) are input by a "$ BINDER" followed by the list of options and option actions. Except for the word "BINDER" following the dollar sign, the card is written exactly as if it were a dollar card input directly to the binder.

Binder input statements are passed to the binder by placing a dollar sign before them. Except for the dollar sign, the input statement is written exactly as if it were input directly to the binder, including a semicolon at the termination of the statement.

Example:

The following input deck is used to compile a main program and a subroutine named SUBR, and replace an existing routine named OLDSUBR with the newly compiled SUBR. The program is then executed.

```
<I>BEGIN JOB;
<I>COMPILE MAIN/PROG FORTRAN LIBRARY GO
<I>FORTRAN DATA
 $ SET SEPARATE
 $ SET AUTOBIND
 $ USE OLDSUBR FOR SUBR;
 C** MAIN PROGRAM
           .
           .
           .
     SUBROUTINE SUBR
           .
           .
           .
<I>END JOB
```

BCD (Reset)

The BCD compiler option allows the compiler to accept source programs in BCD code. The compiler input file must be labeled using a BCL system control card and the BCD option must be set for the region of the file which is in BCD. The BCD option may be set and reset as necessary within the input file.

NOTE

This option will be deimplemented on the 3.3 release. A warning message is issued to this effect.

BCL (Reset)

The BCL compiler option causes strings to be stored internally in BCL code. Setting compiler options B5500 or B5700 causes this option to be set.

NOTE

A warning message is issued for INTMODE=BCL in file declarations, OPEN statements, and CHANGE statements.

BINDER (No associated states)

See the AUTOBIND option.

B5500 (Reset)

The B5500 and B5700 compiler options allow B 5500/B 5700 source programs to be compiled, with the exceptions listed below. These options cause the options BCL and CHARS = 6 to be set.

a. Some compiler options are different.
b. With respect to call by name and call by value, all formal arguments are handled in the manner described in chapter 15 of this manual.
c. No alternate for .EQ., .NE., .GE., or .LE. is recognized.
d. Adjacent exponentations are evaluated in right-to-left order.
e. Common blocks may not be extended backward by EQUIVALENCE statements.
f. If no file declaration card appears with READER instead of a file number, the forms "READ (5, n)" and "READ n" are equivalent.
g. If no file declaration card appears with PRINTER instead of a file number, the forms "WRITE (6, n)" and "PRINT n" are equivalent.
h. If no file declaration card appears with PUNCH instead of a file number, the forms "WRITE (7, n)" and "PUNCH n" are equivalent.
i. There is no LOCK option on the file declaration card.
j. The first character of each NAMELIST input record is ignored. The first record must begin with one blank followed by $ and then followed by the NAMELIST name, followed by at least one blank. The last record must be terminated with &END.
k. Executable and nonexecutable statements may not be mixed on the same source card.

NOTE

This option will be deimplemented on the 3.3 release. A warning message is issued to this effect.

B5700 (Reset)

The B5700 option is the same as the B5500 option.

NOTE

This option will be deimplemented on the 3.3 release. A warning message is issued to this effect.

B7700 (Reset)

When the B7700 option is set and OPT=1, the compiler will generate code better suited for the B 7700 (refer to chapter 20, OPTIMIZATION AND VECTORMODE).

CHARS (No associated states)

The compiler option CHARS=n causes all strings to be stored in the rightmost n *8 (or n *6 for BCL and BCD) bits of each word. The default value of CHARS is 6 for EBCDIC characters. Setting compiler options B5500 or B5700 causes CHARS to be set to 6.

CHECK (Reset)

The CHECK compiler option causes the compiler source input files and updated source file NEWTAPE to be checked for sequence number errors. A sequence error condition exists when a sequence number on a source card image is less than or equal to the sequence number on the preceding card image. A warning message is placed on the output listing when such an error is found. When the CHECK, SEQERR, and NEWSEQERR options are reset, the compiler performs no sequence checking.

CODE (Reset)

The CODE compiler option causes the output listing to contain the object code generated by the compiler. LIST must be set to produce this listing.

DBLTOSNGL (Reset)

When the DBLTOSNGL compiler option is set, all double precision quantities are treated as single precision, and all variables declared to be type DOUBLE PRECISION or REAL *8 will be treated as type REAL.

DEBUG (Reset)

The DEBUG compiler option causes extensive compilation information to be included in the printer listing. When set, the DEBUG option causes lists of stack allocations, scanner information, and indications of some compiler procedure entries and exits to be included in the output listing. The function of this option may change without notice. Setting the TRACE option causes DEBUG to be set.

END (No associated states)

The END compiler option signals an end of file on the medium where it is encountered.

ENTRY (No associated states)

The ENTRY option is used only with the batch facility and is described in chapter 18.

ERRLIST (Reset for non-CANDE compilations, set for CANDE compilations)

The ERRLIST compiler option lists compilation errors separate from the listing of the source input listing. When set, the ERRLIST option outputs the error listing to the compiler output file ERRORFILE. When a compilation error is detected in the source input, the offending line of text and an error message are written to ERRORFILE. This option is provided primarily for use when the FORTRAN compiler is called from a remote terminal by CANDE, but it may be used regardless of the manner in which the compiler is called. The compiler ERRORFILE should be equated to PRINTER or REMOTE only. Otherwise, it may contain unexpected characters. When the compiler is called from CANDE, ERRORFILE is automatically equated to the remote device involved.

FREE (Reset for non-CANDE compilations, set for CANDE compilations)

The FREE compiler option allows the compilation of free-form source programs from the source file CARD. This option allows simple input from remote devices. The following restrictions must be observed:

a. Continuation card images must contain a minus sign (–) or asterisk (*) as the first non-blank character in the first six columns.
b. Comments must have a "C" in column one, and a minus sign or asterisk in column 2.
c. For statements with labels, the label must appear in the first five columns.
d. Each free-form statement may contain 66 characters at most, including blanks and not including sequence numbers. Any characters beyond the 66th character but before column 73 will be flagged with a warning and ignored.
e. Continuation card images are shifted to the right until the continuation character " *" or "–" is in column 6. If this shift causes a card image to extend past column 72 and if there are blanks between the continuation character and the source, the source will be shifted to the left until either the card image no longer extends past column 72 or until the source begins in column 7.
f. Columns 73 through 80 of the source card image are reserved for the sequence numbers, as in the case of formatted source input.
g. Compiler option statements must have a dollar sign ($) in column one or a blank in column 1 with a dollar sign in column 2.
h. FILE and DEBUG statements must have the strings FILEbb and DEBUGb, respectively, in columns 1-6, where b represents a blank.

FREENEW (Reset)

When FREENEW and NEW compiler options are set, the updated symbolic output file NEWTAPE will be created in a form that can be compiled with FREE set.

FREETAPE (Reset)

The FREETAPE option permits the source file TAPE to be in free format.

GRAPH (Reset)

When the GRAPH option is set and OPT=1, a flow chart of the source program will be created (see chapter 20). LIST must be set to produce the graph.

INCLNEW (Reset)

The INCLNEW compiler option controls the contents of the updated symbolic output file NEWTAPE. When the INCLNEW and NEW options are set,the source text, which is input to the compiler via the INCLUDE compiler option, is output to file NEWTAPE. The $ INCLUDE card images will not appear in NEWTAPE. If INCLNEW is reset, then the INCLUDED text will not be placed on the NEWTAPE file; however, $INCLUDE cards will be output to NEWTAPE when the $ is in column 2. If the state of the NEW option is RESET, the INCLNEW option is ignored.

INCLUDE (No associated states)

The special action compiler option INCLUDE allows the user to input text from specified files in addition to the CARD and TAPE files. When the compiler is accepting source input from the CARD or TAPE file and a $INCLUDE card image is encountered, the indicated portion of the file denoted on this card image is inserted into the input text in place of the $INCLUDE card image. The included text may also contain $INCLUDE card images, but only five text nesting levels are permissible.

The following are proper forms of the INCLUDE compiler control statement:

```
INCLUDE
INCLUDE m
INCLUDE m - n
INCLUDE - n
INCLUDE f
INCLUDE f m
INCLUDE f m - n
INCLUDE f - n
```

In these examples, m is either an unsigned integer constant from 1 to 8 digits or an asterisk (*). This item indicates the sequence number of the first card image to be included from the file referenced by this INCLUDE statement. If m is an * rather than a sequence number, then inclusion will begin with either the first card image of the file or the card image following the last card image included from this file by a previous INCLUDE statement at the same level as the present INCLUDE statement. If item m is missing, inclusion will begin at the same point indicated by the use of an *. The item n denotes a one-to-eight digit unsigned integer constant which is the sequence number of the last card image to be included from the file referenced by the INCLUDE statement. If this item is missing, card images are included from the file until the end of the included file is encountered.

The item f denotes either an internal file name (1 to 17 alphanumeric characters long) or a file TITLE enclosed in apostrophes (') or quotes ('') which indicates the file from which the included card images are to be drawn. If this item is missing from the INCLUDE statement, then the file to be employed is the file designated by the last INCLUDE statement at that level. An error condition exists if an INCLUDE statement appears which does not specify a file and no previous INCLUDE statement at the same level has specified a file. Thus, the first INCLUDE compiler control card at any of the five possible levels of text nesting must contain either a file TITLE or an internal file name.

A compiler control card bearing an INCLUDE compiler control statement may contain no other compiler control statements.

The following are examples of INCLUDE compiler control cards:

```
$INCLUDE FILE8 00001000 - 09000000               00569010
$INCLUDE                                         00570000
$INCLUDE'SOURCE/XYZ' - 900                       00000100
$INCLUDE INCLFILE
```

The first example instructs the compiler to accept as input all card images from the library file with internal file name FILE8 which have sequence numbers between and including 00001000 and 09000000. The TITLE and other attributes of the FILE8 file may be specified on a FILE system control card.

The second example instructs the compiler to accept as input a portion of the file accessed by the last INCLUDE statement in this deck at this level. The card images to be included consist of all card images following the last card image accessed by the last INCLUDE statement at this level. If, for example, this example is in the same deck as the first example and no other INCLUDE statements intervene, the file to be accessed is the file with internal name FILE8 and the first card image to be included is the card image with the next sequence number after 09000000.

The third example instructs the compiler to accept as input a portion of the file with the TITLE of SOURCE/XYZ. The card images to be included consist of all card images between the last card image included from that file at this level and the card image with the sequence number immediately higher than 00000900.

The fourth example instructs the compiler to accept as input a portion of the file with the internal name INCLFILE. The card images to be included consist of all card images in the file between the last card image included from that file at this level and the end of the file.

INSTALLATION (Reset)

The INSTALLATION compiler option allows a program to employ installation-provided (nonstandard) intrinsics. When the INSTALLATION option is set, the user may reference intrinsics which the FORTRAN compiler would not otherwise accept. The state of the option, when the first source statement is seen by the compiler, is the state which the option will retain for the compilation. When the option is reset, only standard intrinsics may be employed by the program being compiled.

INTRINSICS (Reset)

This option is discussed in the Installation Intrinsics section of chapter 15.

IOTIME (No associated states)

The IOTIME compiler option specifies a maximum I/O time for the execution of programs submitted under the batch facility, which is discussed in chapter 18. The proper format for the IOTIME control statement is:

```
IOTIME = n
```

where n is an unsigned integer or real constant specifying the time limit in seconds.

JOB (No associated states)

The JOB compiler option is used with the batch facility discussed in chapter 18.

Text Deleted

Text Deleted

LIBRARY (Reset)

The LIBRARY compiler option is similar to the SEPARATE option except that all of the subprograms are compiled into one code file. The title of the resulting code file will be the title given the code file on the compile card. To modify one of the subprograms, all of them must be recompiled.

If the LIBRARY option is set, the SEPARATE option will be set by the compiler. If the compilation is initiated through CANDE, the LIBRARY option is reset, and the SEPARATE option is set; the LIBRARY option will be set by the compiler.

LIMIT (No associated states)

The parameter compiler option LIMIT allows the user to indicate that compilation is to be terminated when a certain maximum number of syntax errors are detected in the source input.

The proper format of the LIMIT compiler control statement is:

```
LIMIT n or LIMIT=n
```

where n is an unsigned integer constant specifying the maximum number of syntax errors. A limit of 0 indicates there is no limit on the number of errors allowed. The LIMIT control statement may appear only on a compiler control card which precedes the first source input statement. The default error limit is 200 unless the compilation is initiated through CANDE, in which case the default error limit is 10.

LINEINFO (Reset for non-CANDE compilations, set for CANDE compilations)

The LINEINFO compiler option causes sequence number information and its relationship to code to be saved. If a program terminates abnormally, the sequence number of the termination point is provided.

LIST (Set for non-CANDE compilations, reset for CANDE compilations)

The LIST compiler option controls the generation of the LINE compiler file. An output listing is generated on the file LINE only if the LIST option is set for some portion of the source input or if the option TIME is set. The source statements for which the LIST option is set will be placed in this listing. The LIST option may be set and reset as desired within the source input. Setting the TRACE compiler option automatically sets the LIST option.

LONG (Reset)

The LONG compiler option alters array storage handling. Normally, arrays requiring more than 4096 words of storage are segmented into 256-word segments for overlay purposes. Arrays, which are declared in a portion of the source input for which the LONG option is set, will not be segmented regardless of their storage requirements.

MERGE (Reset)

The MERGE compiler option allows the compiler to accept source input from more than one file. When the MERGE option is set, the compiler accepts source input from the input file TAPE in addition to source input from the file CARD. The primary input (from file CARD) is merged with the secondary input (from file TAPE) on the basis of sequence numbers to produce the total source input to the compiler. If matching sequence numbers occur on card images from these two files, the primary input card image overrides the secondary input card image. When a primary control card image containing only a dollar sign is encountered, the secondary card image with the same sequence number is ignored. When MERGE is reset, only primary input is accepted by the compiler.

A TITLE for the tape input file may be specified by following the MERGE option with the TITLE enclosed in parentheses.

MONITOR (Reset)

The MONITOR compiler option uses monitoring facilities.

NEW (Reset)

The NEW compiler option causes the updated symbolic output file NEWTAPE to be generated. When the NEW option is set, the source input from the CARD and, if MERGE is set, TAPE files will be placed in the file NEWTAPE. Text inserted into the source input by the INCLUDE option will also be placed in the NEWTAPE file only if the INCLNEW compiler option is also set. All compiler control cards in the source input whose dollar sign appears in column 2 will be placed on the NEWTAPE file when NEW is set.

The NEW option may be set and reset as desired throughout the source input.

The NEWTAPE file will be created despite the occurrence of syntax errors in the source input.

If the NEW compiler option and the FREE or FREETAPE compiler options are set concurrently, the compiler will perform a minor amount of editing on the output to the NEWTAPE file to ensure that the updated symbolic is acceptable as either free or fixed form input.

A TITLE may be specified for the NEWTAPE file by following the NEW option with a TITLE enclosed in parentheses.

NEWSEGMENT (No associated states)

The parameter compiler option NEWSEGMENT controls code file segmentation. Automatic segmentation of the output code file occurs when the size of a program unit exceeds a specified limit. The code generated for the source statements following the NEWSEGMENT compiler option, will be placed in a new code file segment. If this additional segment is not resident during execution, it must be fetched from overlay memory. This fetch causes time consuming branching instructions. Therefore, the use of this control statement should be avoided at points in the code frequently passed during execution such as within loops. When separate compilation is involved, this option may only be employed within main programs.

NEWSEQERR (Reset)

The NEWSEQERR compiler option allows sequence number error checking to be performed on the updated symbolic output file as the file is generated. When the NEW and NEWSEQERR options are set, the updated symbolic file NEWTAPE is generated and the sequence numbers on the card images in this file are checked for errors. A sequence error condition exists when a sequence number on a card image is less than or equal to the sequence number on the preceding card image. When such a sequence error is detected, a warning error message is placed on the printer listing and the NEWTAPE file is not saved.

NOBINDINFO (Reset)

When the NOBINDINFO compiler option is set, the compiler will not generate binding information in the code file; hence, the resulting code file cannot be processed by the binder.

NOWARN (Reset)

The NOWARN compiler option allows the user to alter the compiler's error-handling facility. When the NOWARN option is set, errors which otherwise would have generated warning messages are considered syntax errors.

OMIT (Reset)

When the OMIT compiler option is set, card images from both CARD and TAPE files are ignored by the compiler. They are, however, listed and included in an updated symbolic file. On a listing, the omitted statements are flagged by the word OMIT. See also the VOID and VOIDT options.

OMITDEBUG (Reset)

When the OMITDEBUG compiler option is set, all debugging statements are omitted.

OPT (No associated states)

The parameter compiler option OPT allows the user to alter the form of the object code generated by the compiler by altering the manner in which the compiler handles various source input statements.

The proper format of the OPT compiler control statement is:

```
OPT n or OPT=n
```

where the item n is one of the integer constants 0, 1, or –1. The default value assigned to OPT for a compilation is 0.

A detailed description of optimization appears in chapter 20.

OWN (Reset)

The OWN compiler option affects subroutine variables and arrays which are not formal arguments and are declared or first appear when the OWN option is set. These variables and arrays will retain their values between subroutine calls.

Example:

```
          SUBROUTINE AUGMNT (X, Y, Z)
          DIMENSION Z(5), HOLD(5)
    $SET OWN
          INTEGER  A
          DIMENSION B(5)
          I=I+1
    $RESET OWN
          X=X+I
          HOLD(1)=Y
          Y=Y+A
          A=HOLD(1)
          DO 1 J=1,5
          HOLD(J)=Z(J)
          Z(J)=Z(J)+B(J)
        1 B(J)=HOLD(J)
          RETURN
          END
```

In this example, the OWN option is used to indicate that the variables A and I and the elements of the array B are to retain their values between subroutine calls. When the subroutine is first called, these items are initialized to zero as is normally the case with local variables. Thus, the function of this subroutine is to increment the value of X by 1 during the first call, by 2 during the second call, by 3 during the third call, etc. During each call, the variable Y and the elements of the Z array are increased by the values passed to these formal parameters when the AUGMNT subroutine was last called. During the first call on AUGMNT, these six data items are not increased.

OWNARRAYS (See text for default state)

When the OWNARRAYS option is set, all arrays (and only arrays, unless the OWN option is also set) are treated as if they had been declared with OWN set. If array reinitialization is not required, faster execution will be realized if a program is compiled with OWNARRAYS set.

The default state of OWNARRAYS is the same as that of the compiler generation option SETOWNARRAYS discussed later in this chapter.

PAGE (No associated states)

The parameter compiler option PAGE allows the user to partially control the format of the output listing. The appearance of the PAGE control statement within the source input causes a skip to the top of the next physical page. The output listing then continues, beginning with the PAGE control statement. No argument is associated with the PAGE option.

PRINTER (No associated states)

Option PRINTER=n causes each PRINT statement to employ file n.

PRINTLIMIT (No associated states)

The PRINTLIMIT option specifies a maximum number of print lines that may be generated by a program submitted under the batch facility, which is described in chapter 18. The proper format for the PRINTLIMIT control statement is:

```
PRINTLIMIT = n
```

where n is an unsigned integer constant specifying the maximum number of lines.

PROCESSTIME (No associated states)

The PROCESSTIME option specifies a maximum processor time for the execution of programs submitted under the batch facility which is described in chapter 18.

The proper format for the PROCESSTIME statement is:

```
PROCESSTIME = n
```

where n is an unsigned integer or real constant specifying the time limit in seconds.

READER (No associated states)

Option READER=n causes each READ statement, which does not contain an explicit file designator, to employ file n.

SEGMENTATION (No associated states)

The option SEGMENTATION=n modifies the size of a code segment. A segment size of n words will be used. In place of a number n, the characters MAX may be used. This will be interpreted as 65,535 words. The value of n may not exceed this limit.

SEGS (See text for default state)

The SEGS compiler option causes segmentation information to be included in the output listing. If reset, segmentation information will be deleted with the exception of information appearing alongside card images.

The default state of SEGS depends on the state of compiler generation option NOSEGS discussed later in this chapter.

SEPARATE (Reset)

The SEPARATE compiler option produces separate code files for the following program units: main programs, and function and subroutine subprograms. These seperate code files may be bound to other program unit files via the program binder. When the SEPARATE option is set during a compile for library, a separate code file is generated for each program unit contained in the source input. The code file generated for each of these program units will have a TITLE constructed from the program name contained on the COMPILE system control card initiating the compilation. The TITLE of the main program code file will be identical to the program name. The TITLE of the subprogram code file will be identical to the program name with the lowest-level (rightmost) identifier in the program name replaced by the subprogram name. If the program name on the COMPILE system control card consists of only one identifier, then the TITLE of subprogram code file will simply be the subprogram name.

The SEPARATE option state is not changed after the first source input statement is processed. Hence, one may only specify that all program units in the source input are to be separately-compiled or that all these program units are not to be separately-compiled. The appearance of a LEVEL or LIBRARY compiler control statement at the beginning of a source deck sets SEPARATE for that compilation.

SEQ (Reset)

When the SEQ compiler option is set, new sequence numbers are assigned to the card images placed in the listing and the NEWTAPE file. This option has effect only when LIST or NEW are set.

The default base sequence number is 00001000 and the default increment is 1000. These default sequencing parameters may be overridden in the following manner. A one-to eight-digit unsigned integer appearing on a compiler control card not immediately preceded by INCLUDE, LEVEL, LIMIT, or OPT is assumed to be a special action control statement associated with the SEQ option. This integer will be used as the sequencing base when the SEQ option is set. If the integer is preceded by a plus sign (+), it will be used as the increment size when the SEQ option is set. These sequencing arguments may appear on the same compiler control card image as the SEQ option, on a preceding control card image, or on a later control card image. The following are examples of sequencing arguments appearing on the same control card image as the SEQ option:

```
$SET SEQ 100
$SEQ 20+1
$RESET CODE SET SEQ LIST +200
```

In the first example, sequencing of the LINE and NEWTAPE files will begin at the sequence number 00000100 and continue in default increments of 1000 if no other sequencing increment was specified on a previous compiler control card image. In the second example, sequencing will begin at the sequence number 00000020 and proceed in increments of 1. In the third example, sequencing will begin at the default sequence number 00001000 if this is the first time SEQ is set and no other base sequence number was specified on a previous compiler control card image, and proceed in increments of 200. Otherwise, sequencing will begin at a sequence number 200 greater than the last sequence number assigned, or at the base sequence number assigned by a preceding control card.

The following is an example of a compiler control card specifying sequencing arguments but not setting the SEQ option:

```
$ 100 + 100
```

This compiler control card only specifies that, when SEQ is set, sequencing is to begin with the sequence number 00000100 and proceed in increments of 100. If the SEQ option is set when this control card appears, these two sequencing arguments take effect immediately.

SEQERR (Reset)

The SEQERR compiler option, when set, performs sequence number error checking on the source input files. A sequence error condition exists when a sequence number is less than or equal to the sequence number on the preceding card image. When such a sequence error is found, an error message is placed on the listing and the code file generated by the compilation will not be saved.

SINGLE (See text for default state)

The SINGLE compiler option alters the format of the compilation listing. When SINGLE is set, the listing will be single-spaced; when SINGLE is reset, the listing will be double-spaced. The default value of SINGLE depends upon the state of the compiler generation option SINGLEBYDEFAULT discussed later in this chapter.

STACK (Reset)

The STACK compiler option increases the amount of compilation information contained on the listing. When STACK is set, the listing will contain the address couples of items named in the program being compiled. LIST must be set to produce the listing. When the TRACE compiler option is set, the STACK option is set.

SUPRS (Reset)

The SUPRS compiler option, when set, causes suppression of warning messages in the listing.

TIME (Reset)

The TIME compiler option generates minimal compilation listing. When the TIME option is set and the LIST option is reset, the listing will contain only compilation "trailer" information such as the number of cards scanned, number of errors, compilation times, and so on. This trailer information is also generated when LIST is set.

TRACE (Reset)

The TRACE option sets the options LIST, CODE, DEBUG, and STACK. Once set, the TRACE option cannot be reset; however, any or all of the four options TRACE sets may be reset individually.

VECTORMODE (Reset)

This option is discussed in chapter 20.

VOID (Reset)

The VOID compiler option limits source input. When VOID is set, all input from files CARD and TAPE, except compiler control cards in the CARD file, will be ignored by the compiler. The ignored input will be neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW. The VOID option may be reset only by a compiler control card in the CARD file.

VOIDT (Reset)

The VOIDT compiler option limits source input. When VOIDT is set, only input from file CARD will be compiled; input from the secondary file TAPE, including compiler control cards, will be ignored until VOIDT is reset. The ignored input will be neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW.

XREF (Reset)

The XREF compiler option outputs a cross reference of all identifiers used in a compiled program. The LIST compiler option need not be set to generate a cross reference listing. The identifiers are arranged according to the EBCDIC collating sequence with lists of sequence numbers of card images on which the identifier appears.

The state of the XREF option is not changed after the first source statement is seen by the compiler.

XREFFILES (Reset)

The XREFFILES compiler option causes files to be saved for INTERACTIVEXREF. These files have the following titles:

```
XREFREFS/<code file name>
XREFDECS/<code file name>
```

<code file name> is the name of the code file the compiler is generating.

Running SYSTEM/XREFANALYZER with a negative task value creates the same files as does the XREFFILES option.

$ (Reset)

The $ compiler option causes all compiler control statements to be included in the listing if the LIST option is set.

FORTRAN COMPILER GENERATION OPTIONS

Several different versions of the standard FORTRAN compiler may be generated by setting or resetting the following compiler generation options. Each option name is followed by its default setting in parentheses.

BIGOPTJOB (Reset)

The BIGOPTJOB option facilitates the optimization of FORTRAN programs that are larger than the standard compiler can handle. If set, BIGOPTJOB increases the memory requirements of the FORTRAN compiler. This option may be set only if the OPTLSS1 compiler generation option is reset.

FASTGRAPH (Reset, but must be set if OPTLSS1 is set)

When the FASTGRAPH option is set, graphs will be printed faster by ignoring end-of-page sensing on the line printer. When the option is reset, end-of-page sensing occurs and the graphs will not be drawn across a fanfold in the paper.

GETFIRSTDOLLARCARDFROMSPO (Reset)

When the GETFIRSTDOLLARCARDFROMSPO option is set, the compiler will require that the first compiler control card (dollar card) be input through the operator's supervisory console.

MODELI (Reset)

If the MODELI option is set, the compiler will not emit code to perform indexable value calls on arrays. This code is emitted when the option is reset. The option should not be reset unless the hardware feature is available.

MULTI (Set)

The MULTI option allows the compiler to optionally operate in batch mode. Batch facilities are not available if this option is reset (see the discussion of batch facilities in chapter 18).

NOSEGS (Reset)

When the NOSEGS option is set, the compiler resets the compiler option SEGS by default.

OPTLSS1 (Reset)

When the OPTLSS1 option is set, the FORTRAN compiler control option OPT may only be set to values less than 1. All OPT values are valid if OPTLSS1 is reset. A smaller compiler will result if OPTLSS1 is set.

SETOWNARRAYS (Reset)

When the SETOWNARRAYS option is set, the compiler sets the compiler control option OWNARRAYS by default. The OWNARRAYS compiler option will default to reset if SETOWNARRAYS is reset.

SINGLEBYDEFAULT (Set)

When the SINGLEBYDEFAULT option is set, compiler control option SINGLE will be set by default, causing single-spaced FORTRAN listings. When SINGLEBYDEFAULT is reset, the listings will be double-spaced by default.

Paragraph Deleted

20. OPTIMIZATION AND VECTOR MODE

Optimization and vector mode are two important facilities that in many cases will allow the compiler to generate faster-running object programs.

OPTIMIZATION

The parameter compiler option OPT may be used to alter the object code generated for a program by altering the manner in which the compiler handles individual or groups of source input statements.

The proper format of the OPT compiler control statement is:

```
OPT n or OPT = n
```

where n is one of the integer constants –1, 0, or 1. The default value is 0. The value of OPT may be changed at any time. However, a change to or from the value 1 takes effect only at the beginning of a program unit. A change from 0 or –1 to the other takes effect with the next executable statement.

OPT = 0

When the value 0 is associated with OPT, the FORTRAN compiler performs local optimization within statements to take advantage of special hardware features. In addition, floating point constants with zero fractional parts are not normalized; that is, they are stored as integers. When this OPT value is used, the compiler enforces the rule that operands used in arithmetic expressions may be evaluated in any order.

A statement function now passes parameters by name when OPT=0. Previously, these parameters were passed by value.

OPT = –1

When the value –1 is associated with OPT, compiler action is similar to that which occurs when the OPT=0, except that floating point integer constants are normalized and operands are evaluated from left to right.

In order to illustrate the difference between OPT = 0 and OPT = –1, the following examples are provided. Assume that F is a one-argument function subprogram which alters the value of its argument when the function is invoked and that A is a one-dimensional array.

The first example consists of the input source statement:

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

When OPT=–1, the value of I changes, from the value it holds when it is first used as a subscript, after the function F is invoked. The subscript A(I) may be changed. With a value of 0 assigned to OPT, the subscript is evaluated only once (it is the value held by I at the time the execution of the statement is initiated). The invocation of F, however, will alter the value of I used in the last term of the expression.

A second example of the difference between OPT = 0 and OPT = –1 concerns a two-dimensional array B. The example statement is:

```
X=B(I,F(I))
```

The value associated with the OPT option at the time this statement is compiled will determine which element of the B array has its value assigned to the variable X. When OPT=0, the I subscript will be referenced after the invocation of F and thus hold the value assigned to it by the execution of F. When OPT=–1, the two subscripts will be evaluated from left to right; thus, I will hold the value it contains before F is referenced.

Normally, the difference between OPT = 0 and OPT = –1 will have no effect on program results; however, the use of the OPT value of 0 may result in better code generation.

OPT = 1

When the value 1 is assigned to OPT at the beginning of a program unit, much more powerful optimization is invoked. This value is only valid if the FORTRAN compiler was compiled with the compiler generation option OPTLSS1 reset.

A function subprogram which has a label as a formal argument, or any program unit referencing such a function, is not eligible for optimization. The compiler will generate a syntax error for any ineligible program unit if OPT = 1.

Since OPT may be changed on a program unit basis, the presence and use of ineligible function subprograms does not prevent other program units from benefitting, perhaps significantly, from optimization.

Additional Compiler Control Card Considerations

The OPT option affects the way in which OWN variables are handled. If OPT = 1, then OWN array handling is improved, but simple variables declared with OWN set do not result in the best code (see chapter 19 for the discussion of compiler options OWN and OWNARRAYS).

The compiler option GRAPH has been added to the FORTRAN compiler for use with optimization. If OPT = 1, GRAPH is set, and LIST is set when the END statement is encountered in a program unit, the compiler will produce a rough flow chart of the (pre-optimized) program unit; this can assist programmers in making fuller use of the optimization features by showing complex flows of control that reduce the effectiveness of optimization.

If OPT < 1, vector mode code will not be emitted by the FORTRAN compiler. See the discussion of vectormode elsewhere in this chapter.

FORTRAN statements MONITOR and DUMP cannot be used with OPT = 1; however, TRACE, STATISTICS, PROGRAMDUMP, and DUMP STATISTICS may be used.

Free-format designators other than / are not available with OPT = 1.

Although the B 7000 series and B 6000 series are code compatible, some constructs optimal on one computer are not optimal on the other. If the compiler option B7700 is set, code favoring the B 7700 is emitted. This option is reset by default, in which case the code best suited for the B 6000 series is emitted. This option only applies when OPT = 1.

If a program unit becomes too large, the compiler will print the following message:

```
PVALUENO TOO SMALL, INCREASE IT
```

This indicates that certain compiler limits have been exceeded. To correct this situation, the user should either break up the program unit into two or more subunits, or he should recompile on a version of the FORTRAN compiler that has BIGOPTJOB set. If the same message appears when BIGOPTJOB is set and the program unit cannot be broken up, optimization is not available for that program unit.

VECTOR MODE

Vector mode is a special hardware feature of some B 6700 systems which can provide significantly increased efficiency in array handling and loop control. This hardware may be used only under certain conditions described below, but serves as a powerful tool when the requirements are met.

The FORTRAN compiler will generate vector mode code if all of the following conditions are met:

a. The FORTRAN compiler has been compiled with the option OPTLSS1 reset.
b. The compiler option OPT has been set to 1 for the affected program unit and the unit is eligible for that level of optimization (see the description of optimization).
c. The compiler option VECTORMODE has been set for the portion of the program to be optimized.
d. The program unit contains a loop that has all of the following characteristics:

1. It is a DO loop or a loop constructed with IF statements that functions equivalently to a DO loop.
2. The loop is entered only from the statement immediately preceding the loop; that is, the loop is always fallen into. No statement branches into the loop and no statement branches to the first statement of the loop except the actual looping and testing code.
3. The increment expression is a constant less than or equal to 3.
4. The initial and final values are integers and are constant with respect to the loop; that is, if the initial and final values are not constants, then they must not be altered within the loop.
5. The value of the control variable is not changed inside the loop, except when it is implicitly incremented at the end of the loop.
6. The control variable is not in COMMON storage.
7. No CALL, I/O, PAUSE, ZIP, or other file-handling statements appear in the loop.
8. No function references appear in the loop.
9. There are no intrinsic calls, including exponentiation (**), within the loop unless the result of the call is invariant with respect to the loop.
10. If arrays or COMMON elements are altered within the loop, or if arrays are referenced within the loop using a subscript that changes with each iteration of the loop, then there are no more than three such usages. That does not mean a limit of three arrays:

```
      DO 10 I = 1, N
10    A(I) = A(I + 1) + A(I - 1) + B(N)
```

The above loop would generate vector mode code if the other conditions are met, because there are only three arrays being altered or used with varying subscripts. If B (N) had been replaced by B (I), then no vector mode code would be generated. If the statement had been

```
10 B(N) = A(I) +A(I + 1) + A(I + 3)
```

then no vector mode code would be generated, because while only three array subscripts were varying, a fourth array was being altered in value.

11. An array or COMMON element which is altered within the loop or an array referenced in the loop whose subscript changes with each iteration of the loop is termed a "vector" array.

A vector array is not referenced in a purely conditional manner. If it is, no vector mode code is generated.

```
      DO 10 I = 1, 10000
10    IF (I.LT.50) A(I) = A(I) + 1
```

The above example would not generate vector mode code.

```
      DO 10 I = 1, 10000
      A(I) = A(I) + 1
10    IF (I.LT.50) A(I) = A(I) + 1
```

The above example would generate vector mode code if the other conditions are met.

12. The partial derivative of each of the subscript expressions with respect to the control variable is an integer which is constant with respect to the loop.

This essentially means that if the subscript is too complicated, vector mode code will not be emitted.

```
      DO 10 K = 1, N
10    A(I, J) = B(I, K)* C(K,J) + A(I,J)
```

The above will give vector mode code, if the other conditions are met.

```
      DO 10 K = 1, N
10    A(K*K) = B(K + K)
```

The above will not cause vector mode code to be generated because of the subscript of A. The subscript of B is acceptable.

13. There are no segmented arrays.
14. There are no formal arrays unless LONG is set for the entire program.
15. The loop is an innermost loop, that is, it does not contain any other loops.

Use of the intrinsics REAL and AIMAG within an assignment statement in the loop, where the left side of the assignment depends on the loop control variable, results in program termination if VECTORMODE is set and OPT=1.

An attempt to execute vector mode code on a processor not equipped with the proper hardware will result in program termination.

NOTE

The VECTORMODE option has been deimplemented on B 7000 series systems. An attempt to use this option results in a warning message.

21. LIBRARIES

A library is a program which provides a set of procedural "entry points" which can be called on by other programs called "user programs". Libraries may be created with, or called from, FORTRAN programs. (A detailed discussion of the usage of libraries is found in Chapter 20 of the B 5000/B 6000/B 7000 System Software Operational Guide, Volume II; familiarity with this material is assumed in the discussion that follows.)

BLOCK GLOBALS SUBPROGRAM

In FORTRAN, a non-executable program unit called the BLOCK GLOBALS subprogram in used to declare libraries, export lists, and files.

BLOCK GLOBALS STATEMENT

The non-excutable BLOCK GLOBALS statement indicates the beginning of a BLOCK GLOBALS subprogram.

The proper format of this statement is:

```
BLOCK GLOBALS
```

USE OF BLOCK GLOBALS

Only one BLOCK GLOBALS subprogram is allowed in a program. This subprogram must occur after any old-style FILE declarations (see Appendix B) and must precede all other source statements in the program.

A BLOCK GLOBALS subprogram must end with an END statement and may contain any of the following statements:

LIBRARY statement
EXPORT statement
FILE statement

FILE STATEMENT

Files can be declared in the BLOCK GLOBALS subprogram by use of the non-excutable FILE statement, which has the same format as the first form of the FILE declaration given in Appendix B. The only difference between these two forms of file declaration is that the file declaration statement outside the BLOCK GLOBALS subprogram must appear in the format specified in Appendix B; no such requirement is made of the file declaration statement in the BLOCK GLOBALS subprogram.

CREATING LIBRARIES

Two constructs in FORTRAN create libraries: the EXPORT statement and the CALL FREEZE statement.

EXPORT STATEMENT

The non-executable EXPORT statement is used to specify which subroutines and functions are to be provided as entry points to the library on execution of a CALL FREEZE statement.

The proper format for this statement is:

EXPORT s
where s is a list, separated by commas, of subprogram names or subprogram assignment clauses used as entry point designators.

Subprogram assignment clauses are of the form s=n, where n is the name by which the subprogram is known to users of the library.

CALL FREEZE STATEMENT

The executable CALL FREEZE statement changes the running program into a library stack.

The proper formats for this statement are:

```
1.  CALL FREEZE (TEMPORARY)
2.  CALL FREEZE (PERMANENT)
```

If TEMPORARY is specified, the library stack exists only as long as at least one user is referencing it. If PERMANENT is specified, the library stack continues to exist even if no users are referencing it.

Example

The following is an example of a FORTRAN program that creates a library.

```
$ SHARING - PRIVATE
      BLOCK GLOBALS
          FILE 6(KIND=PRINTER)
          EXPORT SINE="SIN",COSINE
      END
      REAL FUNCTION SINE(X)
C*           PERFORM SINE CALCULATION...
           SINE=X
      END
      REAL FUNCTION COSINE(X)
C*           PERFORM COSINE CALCULATION...
           COSINE=X
      END
C*    MAIN PROGRAM
           WRITE(6,/)SINE(X),COSINE(X)
           CALL FREEZE(TEMPORARY)
      END
```

REFERENCING LIBRARIES

Two constructs in FORTRAN reference libraries: the LIBRARY statement and the IN LIBRARY statement.

LIBRARY STATEMENT

All libraries used by a program must be declared in a LIBRARY statement.

The proper formats of this statement are:

```
1.  LIBRARY n
2.  LIBRARY n (t)
```

where n is the name of the library as declared in the BLOCK GLOBALS subprogram, and t is a list of file attribute assignment clauses separated by commas.

File attribute assignment clauses are of the form fa=s, where fa is one of the library file attributes TITLE, INTNAME, and LIBPARAMETER, and s is an EBCDIC string. One such clause may appear in the LIBRARY statement for each file attribute.

Specification of the attributes is optional; by default, the declared library name is used for TITLE and INTNAME. If the INTNAME is assigned but the TITLE is not, the name assigned to the INTNAME is used for the TITLE, as well. (See LIBRARY ATTRIBUTES in this section.)

IN LIBRARY STATEMENT

The IN LIBRARY statement is used to indicate that a subprogram is to be found in a library.

The proper formats of this statement are:

```
1.  IN LIBRARY n
2.  IN LIBRARY n (ACTUALNAME=s)
```

where n is the name of the library as declared in the BLOCK GLOBALS subprogram, and s is the actual name of the subprogram in the library.

The only statements allowed in a subprogram that contains an IN LIBRARY statement are type statements and DIMENSION statements. These statements must specify the types and dimensionalities of the parameters of the subprogram.

Example

The following program uses the two functions created in the example under CREATING LIBRARIES and a subroutine that is in another library.

```
          BLOCK GLOBALS
               FILE 5(KIND=REMOTE)
               FILE 6(KIND=PRINTER)
               LIBRARY LIB1(TITLE="MATHINTRINSICS",
         *                    INTNAME="MATHINTRINSICS")
               LIBRARY LIB2
          END
          REAL FUNCTION SIN(X)
               REAL X
               IN LIBRARY LIB1
          END
          REAL FUNCTION COS(X)
               REAL X
               IN LIBRARY LIB1(ACTUALNAME="COSINE")
          END
          SUBROUTINE SUB
               IN LIBRARY LIB2
          END
        C* MAIN PROGRAM
               READ(5,/)X
               WRITE(6,/)SIN(X),COS(X)
               CALL SUB
          END
```

ENTRY POINTS AND PARAMETERS

A library program provides entry points for use by user programs. When a library entry point is first called, the user program is suspended, and the library program, if not already initiated, is initiated by the MCP. The library runs as a normal program until it executes a CALL FREEZE statement; this statement makes EXPORTed entry points available. All entry points that the user program declared for the library are linked to the entry points provided by the library, and the user program resumes execution.

A library entry point may be any of the following:

Subroutine
Untyped function
Real function
Integer function
Double-precision function
Logical function
Complex function

Parameters associated with library entry points affect the type matching that is performed on entry points during library linkage. Parameters are passed according to the following rules:

1. Variable parameters are passed by name.
2. Array parameters are passed by reference.
3. A formal parameter array is passed as two parameters, the array itself and the lower bound of the array as an integer, by value.

Parameters to a library entry point may be of any of the following types:

Integer
Real
Logical
Double-precision
Complex
Integer array
Real array
Logical array
Double-precision array
Complex array

LIBRARY ATTRIBUTES

Libraries, like files, have attributes that can be set and tested programmatically. The EBCDIC string-type attributes INTNAME, TITLE, and LIBPARAMETER are associated with libraries. INTNAME is the internal identifier for the library. TITLE is the name of the library code file. LIBPARAMETER is an EBCDIC string used to transmit information from the user program to the selection procedures of library programs that provide dynamic entry point linkage. These attributes may be assigned names in the LIBRARY statement used to declare the library.

CHANGING LIBRARY ATTRIBUTES

Library attributes may be changed dynamically by the user program before the first attempt to link to the library.

In addition, the executable CHANGE statement may be used to change library attributes before the library is accessed. (See CHANGE Statement in Section 12.) The format for the CHANGE statement as applied to libraries is:

CHANGE (n,t)
where n is the library name, and t is a list of the file attribute assignment clauses separated by commas.

The file attribute assignment clause, t, is of the form fa=i, where fa is one of the library attributes INTNAME, TITLE, and LIBPARAMETTER, and i is the new attribute name to be assigned. One such clause may appear for each library attribute.

Example

The following example uses the CHANGE statement to change the TITLE attribute of the library LIB1.

```
REAL A(4)/"LIBRARY/ONE."/
CHANGE(LIB1,TITLE=A)
```

APPENDIX A. SAMPLE PROGRAM

An example of a deck containing a typical, valid B 7000/B 6000 series FORTRAN program is presented here and discussed in the following paragraphs. The symbol <I> denotes an invalid character (for example, a 1-2-3 punch) in card column 1. The card images in the deck appear in the order listed here.

```
<I>  BEGIN JOB SAMPLE/PROGRAM;
     COMPILE FORTRAN/EXAMPLE FORTRAN;FORTRAN DATA
FILE  6(TITLE="PLOT")
$SET LIST
      SUBROUTINE PLOT(YMAX,YMIN)
C   SUBROUTINE SUBPROGRAM GENERATING A PRINTER PLOT OF THE 101 ELEMENTS
C   OF THE ARRAY Y VS. A SCALE RANGING FROM 0 TO 100.
      COMMON/A. Y(101)
      REAL LINE(101),MINUS
      DATA BLANK,PLUS,MINUS,BAR,PLT/" ", "+","-",":","*"
      WRITE(6,1)
1     FORMAT('1     PLOT'//)
      X=1
      DO 2 I=1,101
      LINE(I)=MINUS
2     IF(MOD(I-1,10).EQ.0) LINE(I)=PLUS
      WRITE(6,3) YMIN,YMAX
3     FORMAT(15X,E15.5,80X,E15.5)
      DY=(YMAX-YMIN)/100.
100   YY=Y(X)
      IY=IFIX((YY-YMIN)/DY+0.5) + 1
      IF(IY.LT.1.OR.IY.GT.101) GO TO 4
      LINE(IY)=PLT
4     WRITE(6,5)LINE
5     FORMAT(20X,101A1)
      IF(MOD(LINES,10).EQ.0)WRITE(6,8)X-1
8     FORMAT('+',3X,E15.5)
      LINES=LINES+1
      IF(MOD(LINES,10).EQ.0) GO TO 9
      DO 6 I=1, 101
      LINE(I)=BLANK
6     IF(MOD(I-1,10).EQ.0) LINE(I)=BAR
7     X=X+1
      IF(X.GT.101) RETURN
      GO TO 100
```

```
9       DO 310 I=1, 101
        LINE(I)=MINUS
310     IF(MOD(I-1,10).EQ.0) LINE(I)=PLUS
        GO TO 7
        END
C START OF MAIN PROGRAM
$SET SEQ
        DIMENSION X(101)
        COMMON/A/X
        DO 1 I=0,100
        X(I+1)=SIN(I/6.0)
1       WRITE(6,2) I,X(I+1)
        CALL PLOT (1,-1)
C THE MAXIMUM RANGE OF THE FUNCTION TO BE PLOTTED IS 1 to -1
2       FORMAT (I4,3X,F7.2)
        STOP
        END
<I>END JOB
```

This is an example of a compile-and-go FORTRAN job. The end result of the execution of this program is a line printer plot and listing of values of the function:

$$y = \sin\left(\frac{x}{6}\right)$$

for x= 0,1,2,3,...,100. The plot produced contains grid lines and axis values, and each function value is plotted on one line of print beginning with a line for x=0 and terminating with a line for x=100. The program which produces this output is constructed from two program units: a main program and a subroutine named PLOT. The deck containing these program units is described here.

The first card image of the sample deck contains a COMPILE system control statement and a DATA system control statement. This card image causes a compile-and-go FORTRAN compilation to be initiated with the following EBCDIC-coded cards being used as source input for the compilation. The TITLE of the code file which is to be generated by the compiler and executed is FORTRAN/ EXAMPLE. This code file is not retained after execution.

The first card image in the compiler source file is a file declaration card image which specifies that the program file referenced by the file designator 6 (and identified by the internal file name FILE6) be assigned the TITLE PLOT (see appendix B). The other attributes of this file are allowed to retain their default values.

The following card image is a compiler control card image which sets the compiler option LIST, causing the compiler to produce a listing of the source program.

The following source input card images contain the subprogram PLOT. This subroutine generates the plot and has two formal arguments, YMAX and YMIN which specify the range of the values to be plotted. The values to be plotted are transmitted to the subroutine via the COMMON block named A to the array Y. The values contained in this 101-element array are plotted on the plot grid using the character "*". The plot grid is constructed line by line via the LINE array. The PLOT program unit illustrates the use of comment card images, the DATA statement, strings in FORMAT specifications, the MOD intrinsic, statement labels, and contains three DO loops. The subroutine is terminated by the END statement which appears two statements after the IF statement with the label 310.

The subroutine is immediately followed by a comment card image and a compiler control card image which assigns the state SET to the SEQ compiler option. This control card image causes the following source input card images to bear sequence numbers when placed in the printer listing, beginning with the sequence number 00001000 and continuing in increments of 1000. The source card images following the subroutine constitute the main program.

This main program places into the 101-element array X the 101 values of the sine function. These values are transmitted to the PLOT array Y via the common block with the name A and are listed by the main program after they are calculated. The main program calls the PLOT subroutine with the actual parameters 1 and –1, which correspond to the formal parameters YMAX and YMIN, respectively.

The END JOB system control card image which terminates the compilation deck immediately follows the END statement which terminates the main program.

APPENDIX B. FILE DECLARATIONS

Default file attributes are associated with each program file. A special source statement type is provided which assigns non-default file attributes to program files. (For a discussion of file attributes, see I/O Subsystem Reference Manual, form number 5001779.) These source statements are not part of the FORTRAN language; when used, these file declaration statements must precede all FORTRAN source statements.

File declarations may also inform the compiler of file designators which do not explicitly appear in the program but which may be referenced during program execution by an arithmetic expression.

A file declaration is denoted by the appearance of the word FILE in columns 1 through 4 of the card image. Columns 5 and 6 must be blank. File information is contained in columns 7 through 72, and sequence numbers may appear in columns 73 through 80. Also, as many continuation card images as desired may be used to extend the file declaration.

Two forms of the file declaration are available.

The first form resembles FILE system control card images. The file designator, an unsigned integer from 1 to 99, is followed by a list of attribute assignment clauses separated by commas and enclosed in parentheses. Each clause is of the form fa=v, ,where fa is a valid file attribute identifier as listed in I/O Subsystem Reference Manual, form number 5001779, and v is an expression or mnemonic whose type and value are valid for that attribute identifier. This form of the file declaration is not valid if compiler option B5500 or B5700 is set.

Any pointer valued attribute, such as TITLE, is set using an EBCDIC string; that is, the compiler options ASCII, BCD and BCL must be reset when the file declaration is encountered by the compiler. The compiler will append a period to the end of the string.

NOTES

A warning message is issued for INTMODE=BCL in file declarations.

The BCD option will be deimplemented on the 3.3 release. A warning message is issued to this effect.

Example:

```
FILE 16 (TITLE="(USERME)MY/FYLE.",MAXRECSIZE=10,KIND=DISK)
```

Any attribute whose type is boolean may be assigned the value TRUE by use of the attribute alone.

Example:

```
FILE 27 (TITLE="MINE",KIND=DISKPACK,CYLINDERMODE)
```

CYLINDERMODE, a boolean attribute, will be set to TRUE.

The following restriction now exists for the mnemonic values which can be assigned to the attribute KIND. Plurals (SPOS, READERS, DISKPACKS, PACKS, PUNCHES, PRINTERS, TAPES, TAPE9S, TAPE7S, PETAPES) must be specified in the singular (SPO, READER, DISKPACK, PACK, PUNCH, PRINTER, TAPE, TAPE9, TAPE7, PETAPE).

File attributes not explicitly assigned in the file declaration retain their default values.

The second form of the file declaration consists of 1 to 13 file information clauses. Each clause assigns a value to a file attribute.

The first file information clause is required and must be of the following form:

```
n = t
```

where n is either an unsigned integer file designator from 1 to 99, the literal READER, the literal PRINTER or PRINT, or the literal PUNCH, and t is the string of characters which are to be assigned as the TITLE of the designated program file.

The following replacements for the item n produce the indicated effects:

a. When n is an unsigned integer constant, the file declaration affects all I/O statements employing a file designator with a value of n. Other I/O statements may be affected when n is 5, 6, or 7. If n is 5, the I/O statements READ f and READ f,m are affected by the file declaration unless n is READER in another file declaration. If n is 6, then the I/O statements PRINT f and PRINT f,m are affected by the file declaration unless n is PRINT or PRINTER in another file declaration. If n is 7, then the I/O statements PUNCH f and PUNCH f,m are affected by the file declaration unless n is PUNCH in another file declaration.
b. When n is READER, the file declaration affects all READ f and READ f,m statements. Such a file declaration causes a program file with the internal name READER to exist. Hence, input statements of the form READ(5,f) or READ(5,f)m will reference the program file with the internal name FILE5, while READ statements without file designators will reference the file with the internal name READER.
c. When n is PRINTER or PRINT, the file declaration affects all PRINT f and PRINT f,m statements. Such a file declaration causes a program file with the internal name PRINTER to exist. Hence, output statements of the form WRITE(6,f) or WRITE(6,f)m will reference the program file with the internal name FILE6, while the PRINT statement will reference the file with the internal name PRINTER.
d. When n is PUNCH, the file declaration affects all PUNCH f and PUNCH f,m statements. Such a file declaration causes a program file with the internal name PUNCH to exist. Hence, output statements of the form WRITE(7,f) or WRITE(7,f)m will reference the program file with the internal name FILE7, while PUNCH statements will reference the file with the internal name PUNCH.

NOTE

The files declared by file declarations bearing the designators READER, PRINTER (or PRINT), or PUNCH may be referenced only by the I/O statements READ without file designator, PRINT, and PUNCH, respectively.

The item t may assume one of the following forms:

a. This item may be a series of 1 to 14 identifiers separated by slashes and optionally delimited by blanks. This series of identifiers will be assigned to the TITLE attribute when the file is created or is the directory title of an existing file. Each identifier in the file TITLE consists of 1 to 17 EBCDIC characters. An identifier may have one of two forms: it may begin with a letter character which is followed by 0 to 16 letter or digit characters (including the character \$), or it may contain any valid EBCDIC characters including special characters enclosed in apostrophes (') or quotes(“). Such delimiters are not considered as part of the file TITLE.

   Valid examples of this form of t are (for EBCDIC-coded input):

```
 A9
A/BCDEFGHISTRQZXCVY
 'A@BCD&F'
 G678/'R&S'/ DATAFILE /'END-OF-TABULATION'
```

b. Alternately, t may reflect the implementation of file security on the system. A file TITLE, which is constructed as described in (a), may be preceded by the character * or by the following item:

```
(<usercode>)
```

where *<usercode>* which is constructed in the same manner as each identifier in the TITLE. This identifier may not be USERCODE. The * and *(<usercode>)* forms of t are for use with jobs run under a usercode. Under such conditions, every TITLE specified as in (a) will have the usercode automatically added to it as the leftmost identifier. An * preceding the TITLE indicates that no such addition is to occur, and a *(<usercode>)* item preceding the TITLE allows the user to specify the usercode to be added.

Valid examples of this form of t are (for EBCDIC-coded input):

```
*A9
(USERX) A/BCDEFGHISTROZXCVY
('USER-THREE') 'A@BCD&F'
* 'A/B'/CDE99
```

The n=t file information clause is the only required clause of the second form of a file declaration. Hence, the following are valid examples of file declarations for EBCDIC-coded input:

```
FILE  6=OUTPUT/'INFO-FILE'/ONE
FILE  PUNCH = FILESEVEN
```

Up to twelve unique file information clauses (separated by commas) may follow the initial clause just described. These clauses are listed here and discussed in the following paragraphs. These clauses may appear in a file declaration in any order. Those attributes not specified in a file declaration retain their default values.

```
UNIT = h
AREA = a or AREA = j * k
BLOCKING = b
BUFFER = c
CYLINDERMODE
INTERCHANGE
LINKWORD
RANDOM
RECORD = r
SAVE = s
SERIAL
SINGLEPACK
UNLABELED
```

FILE INFORMATION CLAUSES

UNIT = h

The UNIT file information clause specifies the physical device used by the file. The KIND file attribute is affected by this clause. The item h may be one of the following hardware unit names:

PRINT	(007)	printer
PRINTER	(007)	printer
PRINT,BD	(071)	printer backup disk
PRINTER,BD	(071)	printer backup disk
PRINT,BT	(135)	printer backup tape
PRINTER,BT	(135)	printer backup tape
PRINT,BDBT	(199)	printer backup tape or disk
PRINTER,BDBT	(199)	printer backup tape or disk
READER	(009)	card reader
PUNCH	(011)	card punch
PUNCH,BD	(075)	punch backup disk
PUNCH,BT	(139)	punch backup tape
PUNCH,BDBT	(203)	punch backup tape or disk
DISK	(001)	disk
TAPE	(045)	any magnetic tape
TAPE7	(013)	7-track magnetic tape
TAPE9	(014)	9-track magnetic tape
PETAPE	(015)	phase-encoded magnetic tape
REMOTE	(003)	remote
DISKPACK	(017)	diskpack

In this list, the integer value in parentheses following the unit name is the value of the KIND attribute associated with that unit. PRINT, BDBT and PRINTER, BDBT indicate that the file is a printer backup tape file unless no tape is available, in which case the file is a printer backup disk file. PUNCH,BDBT indicates that the file is a punch backup tape file unless no tape is available, in which case the file is a punch backup disk file.

An example of this clause is:

```
UNIT = PRINT,BD
```

AREA = a or AREA = j * k

The AREA file information clause controls the number of records an output disk or pack file may contain. This clause must be used for output files and must not be used for input files.

The AREA clause affects the AREAS and AREASIZE file attributes. The AREAS file attribute specifies the maximum number of areas allocated for a disk file. The AREASIZE file attribute specifies the number of logical records in each area.

Two forms of the AREA clause are available: AREA=j*k and AREA=a.

AREA=j*k

The value j is the number of logical records in each area (AREASIZE=j). When the BLOCKING clause is used, j must be an integral factor of the blocking factor b; otherwise, unpredictable end-of-file errors may occur.

The value k is the maximum number of disk areas allocated for the file (AREAS=k). The value k must not exceed 1000.

AREA=a

The value a is the total number of logical records which may be stored in the file. The file is assigned 20 areas (AREAS=20) and the logical records are evenly divided among them (AREASIZE=a/20). The value a should be an integral multiple of 20; otherwise, it will be increased to the next higher multiple.

When the BLOCKING clause is used, AREASIZE must be an integral multiple of the blocking factor b; otherwise, unpredictable end-of-file errors may occur.

The following are valid examples of the AREA clause:

```
AREA=100

AREA=1000*10
```

BLOCKING = b

The BLOCKING file attribute causes b logical records per block. The blocking factor b must be an unsigned integer constant. This clause determines the BLOCKSIZE file attribute (BLOCKSIZE =b*logical record size). The file is unblocked by default (BLOCKING=1).

BUFFER = c

The BUFFER file information clause specifies the number of buffers to be used with the file. This clause affects the BUFFERS file attribute. The value c is an unsigned nonzero integer constant which denotes the number of buffers to be employed. The default number of buffers is 2. The size of each buffer is identical to the size of a physical file record, which may be found by taking the product of the blocking factor and the logical record size. No buffer may be longer than 65,535. In general, job times will be diminished by assigning more than two buffers to jobs performing a great deal of I/O operations.

The following is an example of the BUFFER clause:

```
BUFFER=6
```

CYLINDERMODE

The CYLINDERMODE file information clause assigns the value TRUE to the boolean file attribute CYLINDERMODE; the default value of this attribute is FALSE. This attribute is meaningful for diskpack files (UNIT=DISKPACK) only. When TRUE, this attribute assigns file space by diskpack cylinders with each file area (row) within the bounds of a single cylinder. When the value of this attribute is FALSE, file space assignment is made without regard for cylinder bounds and on a space available basis. The size of each file area must not be greater than the cylinder size when the CYLINDERMODE clause appears or an error condition will exist.

INTERCHANGE

The INTERCHANGE file information clause assigns the value TRUE to the boolean file attribute INTERCHANGE; the default value of this attribute is FALSE. This attribute is meaningful for diskpack files (UNIT=DISKPACK) only. When TRUE, this attribute indicates that the file is stored or is to be stored on a Burroughs Interchange pack.

LINKWORD

The LINKWORD file information clause is intended for use with disk, diskpack, and tape files which are accessed via unformatted-I/O statements. This clause indicates that the length of a logical file record may be greater than the length of a physical record. This clause assigns the value 6 to the file attribute FILETYPE.

Because the logical records of a LINKWORD file are of variable length, direct access of a LINKWORD file will not work in all cases.

RANDOM

The RANDOM file information clause is used solely as a documentation aid for disk files. Since there is no distinction between "random" and "serial" disk files on this system, the presence of the RANDOM clause is flagged with a warning and then ignored.

RECORD = r

The RECORD file information clause specifies the size of each logical record in the file. The value r is an unsigned integer constant in the range 1 to 65,535 and it specifies the number of data words contained in each logical record. The default record size is 14 words for reader and punch files, and 22 words for all other files. Care should be taken that buffer size, which is the product of the blocking factor and logical record size, does not exceed 65,535.

The RECORD clause affects the MAXRECSIZE file attribute. This example record clause:

```
RECORD = 30
```

assigns the value 30 to the MAXRECSIZE attribute of the file.

SAVE = s

The SAVE file information clause specifies the length of time an output disk file is to remain in the file directory. Item s is an unsigned integer constant, the value of which may not exceed 999, which specifies the number of days the file is to be retained.

The SAVE clause affects the value of the SAVEFACTOR file attribute. This example SAVE clause:

```
SAVE = 30
```

assigns the value 30 to the SAVEFACTOR attribute of the file.

SERIAL

The SERIAL file information clause is used solely as a documentation aid for disk files. Since there is no distinction between "random" and "serial" disk files on this system, the presence of the SERIAL clause is flagged with a warning and then ignored.

SINGLEPACK

The SINGLEPACK file information clause assigns the value TRUE to the Boolean file attribute SINGLEPACK; the default value of this attribute is FALSE. This attribute is meaningful for diskpack files (UNIT = DISKPACK) only. When TRUE, this attribute specifies that all areas of an output file are to be assigned to a single pack. When the value of this attribute is FALSE, file areas will be uniformly distributed among all packs of the present pack set when the file is open. The left-most identifier of the file TITLE must match the name of the pack set. Thus, the file PACKFILE/ONE requires a pack or packset PACKFILE.

UNLABELED

The UNLABELED file information clause specifies that an output file is to have an omitted label. The presence of this clause assigns the value OMITTED to the file attribute LABELTYPE. The default specification is a LABELTYPE value of STANDARD, except for REMOTE files which are unlabeled by default.

Examples:

The following are examples of valid file declarations.

a. The first example consists of this source card image:

```
FILE  PRINTER=ANALYSIS/L198,  UNIT=PRINT,BD,   BUFFER=6
```

This file declaration assigns the TITLE ANALYSIS/L198 to the program file with the internal name PRINTER, designates this file as a printer backup disk file, and assigns it six buffers.

b. The second example consists of these source card images:

```
FILE  4=PACKSET2/RESULTS, UNIT=DISKPACK, SAVE=20
     1,CYLINDERMODE
     2,INTERCHANGE
```

This file declaration (with continuation lines) assigns the TITLE PACKSET2/RESULTS to the program file denoted by the file designator 4, designates the file as a diskpack file to be retained for 20 days, and indicates that file areas are to be assigned by cylinders to Burroughs Interchange packs.

c. The third example consists of these source cards:

```
FILE  5=
     1* CARDDATA/'HALF-SET', UNIT=DISK, RECORD=15
```

This file declaration (with continuation line) equates the program file denoted by the file designator 5 to the disk file with the TITLE CARDDATA/HALF-set. The * indicates that the file TITLE does not have the usercode under which the job is being run. The file has 15-word logical records. As an addition to this example, assume this file declaration is preceded by this file system control card:

```
<I>FILE FILE5(TITLE=*CARDDATA/"HALF-SET", MAXRECSIZE=13)
```

In the event of conflicts between this system control card and the file declaration, the control card overrides the file declaration. Both the system control card and the file declaration in this example apply to the program file with the internal name FILE5. The control card indicates that the actual size of file records is 13 words rather than 15.

SEPARATE COMPILATION CONSIDERATIONS

Each program unit in a compilation deck may be immediately preceded by file declarations if the compiler option SEPARATE is set for the compilation. At compilation time, file declarations apply to all subsequent program units until a later file declaration redefines the file.

At execution time, the first file declaration that is bound to the host program for a given file designator will have precedence. File declarations for main program hosts will apply to all subprograms that are bound to that main program. Such file declarations may be specified by file declarations preceding the main program at compile time, or by file designators in the main program.

The following sample compilation deck illustrates the use of file declarations with separate compilations.

```
<I>BEGIN JOB;
COMPILE SEQ/PROC FORTRAN LIBRARY; FORTRAN DATA
$SET SEPARATE
FILE  10(TITLE="ABC",KIND=TAPE)
      COMMON A(10)
      DATA A/10*3/
      WRITE(10) A
      CALL SUB
      Y=F(1)
      STOP
      END
FILE  10(TITLE="ARP",KIND=DISK,AREAS=20,AREASIZE=5)
FILE  7(TITLE="ERB",KIND=DISK,AREAS=20,AREASIZE=5)
FILE  5(TITLE="EHU",KIND=TAPE)
      FUNCTION F(X)
      COMMON A(10)
      WRITE(7) A
      WRITE(10)A
      WRITE(5) A
      F=X
      RETURN
      END
FILE  10(TITLE="ETX/FILE10",KIND=PACK,SINGLEPACK)
FILE  5(TITLE="FILE5/AMK",KIND=DISK,AREAS=20,AREASIZE=50)
      SUBROUTINE SUB
      COMMON A(10)
      WRITE(5) A
      WRITE(10)A
      RETURN
      END
<I>END JOB
```

When this main program and two subprograms are bound and executed, file assignments will be made as follows.

a. All writes on FILE10 (the file referenced by the designator 10) will be on the file declared before the main program.
b. All writes on FILE5 (referenced by designator 5) will be on the file declared preceding SUB, since SUB is referenced before F in the main program and hence is bound in first.
c. All writes on FILE7 (referenced by designator 7) will be on the file declared preceding F since this file is not referenced in the main program or SUB.

DEFAULT FILE SPECIFICATIONS

Each file designator is associated by default with a certain kind of file. The standard FORTRAN associations are made between the designators 5, 6, and 7 and card reader, line printer, and card punch files, respectively. Table B-1 lists default file specifications by internal file name.

Table B-1. Default Program File Specifications

Internal File Name	Default File Attribute Values				
	Title	Maxrecsize	Blocksize	Buffers	Kind
FILE1 thru FILE4	FILE1 thru FILE4	22	22 (unblocked)	2	Tape
FILE5	FILE5	14	14 (unblocked)	2	Card Reader
READER	specified in FILE declaration	14	14 (unblocked)	2	Card Reader
FILE6	FILE6	22	22 (unblocked)	2	Line Printer
PRINTER	specified in FILE declaration	22	22 (unblocked)	2	Line Printer
FILE7	FILE7	14	14 (unblocked)	2	Card Punch
PUNCH	specified in FILE declaration	14	14 (unblocked)	2	Card Punch
FILE8 thru FILE99	FILE8 thru FILE99	22	22 (unblocked)	2	Tape

APPENDIX C. FORTRAN FORMAT ERROR MESSAGES

The following is an explanation of format error numbers.

The meanings of the various format error numbers pertaining to freefield input are as follows:

No.	Error Message
15	Data magnitude too large (>=8**64) for phrase or list item.
30	String too long.
32	Array source all blanks (internal data transfer).
64	An error on input occurred when the intrinsic did a logical I/O.
84	An expression as a list element which receives a value on input is not allowed.

The meanings of the various format error numbers pertaining to output are as follows:

No.	Error Message
100	An error on output occurred when the intrinsic did a logical I/O.
102	Format was V specifier, and the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, T, U, X, or Z. [Note: If the list element is single precision, the rightmost character is used. If the list element is double precision, the rightmost character of the first (most significant) word is used.]
103	Format was V specifier of the form rV, and the resultant specifier needed a field width. For example, 3V appears in the format specification list and I is the actual specifier, then the resultant specifier of 3I needs a field width w.
104	Format was V specifier of the form rV, and the resultant specifier needed a field width and decimal places. For example, a 3V specifier resulted in a 3E specifier.
105	Format was V specifier of the form rVw, and the resultant specifier needed decimal places: For example, a 2V* specifier resulted in a 2F6 specifier.
106	Format specifier evaluated to Fw.d form, and d<0.
107	Format specifier evaluated to Ew.d or Dw.d, and d<0.
109	Format specifier evaluated to Gw, and corresponding list element was neither of type integer nor type logical; therefore, the decimal places are considered missing.
110	The list contains an element whose type is inappropriate for its associated format phrase. [Note that a long (>48 bits) string cannot be used with a numeric editing phrase.]
111	Format specifier evaluated to Gw.d, and Gw.d logic chose to edit the expression under Ew.d, but d<1.
112	Format statement had no format specifiers requiring list elements, and format was used with a list.
113	Format specifier evaluated to Ew.d or Dw.d and w<=d.
114	Dynamic w or d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.

No.	Error Message
116	Attempted recursive I/O: evaluation of a list element caused a READ/WRITE/CLOSE on the current file.
117	Record overflow: an attempt was made to output more characters than the record contains.
120	Format specification requires output of more than one record when performing internal data transfer.
131	Dynamic r part of format specifier evaluated to a value greater than the maximum real allowed, 4.31359146673*10**68.
132	Dynamic w part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
133	Dynamic d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
163	MAXRECSIZE not large enough to allow FREEFIELD write.

The meanings of the various format error numbers pertaining to formatted input are as follows:

No.	Error Message
200	An error on input occurred when the intrinsic did a logical I/O.
202	Format was V specifier, and the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, T, X, or Z. [Note: If the list element is single precision, the rightmost character is used. If the list element is double precision, the rightmost character of the first (most significant) word is used.]
203	Format was V specifier of the form rV, and the resultant specifier needed a field width. For example, a 2V specifier resulted in a 2I specifier.
204	Format was V specifier of the form rV, and the resultant specifier needed a field width and decimal places. For example, a 3V specifier resulted in a 3E specifier.
205	Format was V specifier of the form rVw, and the resultant specifier needed decimal places. For example, a 2V* specifier resulted in a 2F6 specifier.
206	Format specifier evaluated to Fw.d form, and d<0.
207	Format specifier evaluated to Ew.d or Dw.d, and d<0.
209	Format specifier evaluated to Gw, and corresponding list element was neither of type integer nor type logical; therefore, the decimal places are considered missing.
210	The list contains an element whose type is inappropriate for its associated format phrase. [Note that a long (>48 bits) string cannot be used with a numeric editing phrase.]
213	Format specifier evaluated to Ew.d or Dw.d and w<=d.
214	Dynamic w or d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
216	Attempted recursive I/O: evaluation of a list element caused a READ/WRITE/CLOSE on the current file.
217	Record overflow: an attempt was made to input more characters than the record contains.

No.	Error Message
218	Invalid data for O or Z format phrase.
220	Format specification requires input of more than one record when performing internal data transfer.
231	Dynamic r part of format specifier evaluated to a value greater than the maximum real allowed, 4.31359146673*10**68.
232	Dynamic w part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
233	Dynamic d part of format specifier evaluated to a value greater than the maximum integer allowed, 549755813887.
250	The U format phrase has yet to be implemented for input.
271	The $ and K format modifiers are not allowed on input.
281	Invalid data for I format phrase.
284	An expression as a list element which receives a value on input is not allowed.
285	The list element was type real, but the input value exceeded the maximum real allowed, 4.31359146673*10*68.
286	The list element was type integer or logical, but the input value exceeded the maximum integer allowed, 549755813887.
291	While inputting a constant using a numeric editing phrase, a non-digit was detected in the exponent part following at least one legitimate digit.
292	While inputting a constant using a numeric editing phrase, two or more exponent signs were detected.
293	While inputting a constant using a numeric editing phrase, an illegal character was detected after the exponent sign and before the exponent value.
294	While inputting a constant using a numeric editing phrase, an illegal character was detected past the decimal point.
295	While inputting a constant using a numeric editing phrase, two or more mantissa signs were detected.

The meanings of the various format error numbers pertaining to free-field input are as follows:

No.	Error Message
400	An error on input occurred when the intrinsic did a logical I/O.
416	Attempted recursive I/O-evaluation of a list element caused a READ/WRITE/CLOSE on the current file.
420	Input from the <core-to-core file part> requires more records than allowed by the <core-to-core records per file part>. [Note: The default is one record per file part.]
442	Hex string for single-precision list element contains more than 12 significant digits.
443	Unmatched quote for hex string, or non-hex character encountered on hex string.
444	Hex string for double-precision list element contains more than 24 significant digits.
467	Input value exceeded the maximum value that the list element is capable of representing.

No.	Error Message
473	Unmatched quote.
484	An expression may not be used as a list element which receives a value on input.

The meanings of the various format error numbers pertaining to format specifications in arrays are as follows:

No.	Error Message
500	Left parenthesis expected.
501	Illegal format character.
502	Integer too large: truncated to 65535.
503	Integer constant expected.
504	Unspecified field width.
505	Decimal field not compatible with field width.
506	Missing decimal point in F, E, D, or G format.
507	Empty editing specifications.
508	Illegal repeat count.
509	Error in Hollerith character count.
510	Illegal increment for T format.
511	Unspecified decimal field.
512	Unspecified scale factor.
513	Unmatched quote or apostrophe.
514	Too many right parentheses.
515	Comma expected.

INDEX

INDEX (Cont)