

---

**FORTRAN EXTENDED  
VERSION 4  
REFERENCE MANUAL**

---

**CONTROL DATA®  
CYBER 170 SERIES  
CYBER 70 SERIES  
6000 SERIES  
7000 SERIES  
COMPUTER SYSTEMS**

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

<b>REVISION RECORD</b>	
<b>REVISION</b>	<b>DESCRIPTION</b>
A (10-22-71)	Original Printing
B (10-06-72)	This revision uses shading to denote non-ANSI features and footnotes to indicate information that applies only to the Model 76 and 7600 computers or only to the Models 72, 73, 74, and 6000 computers. The sections on the Reference Map and COMPASS coded subprograms are new with more details and examples. This manual supersedes (but does not invalidate) the previous edition.
C (5-25-73)	This revision corrects typographic errors and expands the description of some features. This revision reflects Version 4.0 of FORTRAN Extended available with SCOPE 3.4 and KRONOS 2.1 operating systems. Pages affected are: iii, iv, vii thru xi, xvi, xviii, I-1-1, I-1-2, I-1-4, I-2-5 thru I-2-12, I-3-5, I-3-6, I-3-8, I-5-8, I-5-15, I-5-16, I-6-1, I-6-6, I-6-9, I-6-11, I-6-21 thru I-6-26, I-7-1, I-7-2, I-7-20, I-7-21, I-8-1 thru I-8-4, I-8-6, I-8-8 thru I-8-11, I-8-13, I-9-1 thru I-9-4, I-9-8, I-9-15, I-9-16, I-9-19, I-9-20, I-10-2, I-10-13, I-10-14, I-10-16 thru I-10-18, I-10-21, I-10-23, I-10-24, I-10-31, I-10-32, I-11-1, I-11-3, I-11-4, I-11-6, I-12-5 thru I-12-9, I-13-1, I-13-20 thru I-13-22, II-1-1, II-1-2, II-1-15, II-1-17, II-1-37 thru II-1-39, III-2-1 thru III-2-13, III-2-19, III-2-20, III-4-8, III-4-10, III-5-10, III-5-17, III-6-1 thru III-6-9, II-7-1, III-7-6, III-10-2, III-10-5, III-11-1, III-12-1, III-12-2, III-13-1, III-13-9, A-1, A-2, Index-1, Index 12, and Comment Sheet
D (11-30-73)	This revision includes the new features of Version 4.1, as well as minor corrections. Major changes occur in sections I-9 and I-10 for the I/O enhancements. FTN control card options are now arranged alphabetically. Pages affected: iii thru xxi; Part I: 1-2, 1-3, 1-4; 2-1, 2-2, 2-9, 2-10, 2-17; 3-8; 5-15, 5-16; 6-6, 6-8, 6-9, 6-11, 6-12, 6-13, 6-21, 6-25; 7-1, 7-2, 7-3, 7-20, 7-21; 8-12 thru 8-15; 9-1 thru 9-26; 10-1, 10-2, 10-6 thru 10-14, 10-18 thru 10-35; 11-1 thru 11-9; 12-9; 13-30; Part II: 1-37 thru 1-41; Part III: 1-1, 1-2, 1-9, 1-12; 2-3 thru 2-14, 2-19, 2-20; 3-3 thru 3-11; 4-11; 5-1, 5-3, 5-7, 5-11, 5-14; 8-1; 12-1, 12-2; 13-1, 13-2; A-1; Index 1 thru 18; Comment Sheet.
Publication No. 60305600	

Additional copies of this manual may be obtained from the nearest Control Data Corporation sales office.

FORTRAN Extended Version 4  
Reference Manual

© 1971, 1972, 1973, 1974, 1975  
Control Data Corporation  
Printed in the United States of America

Address comments concerning this manual to:

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

or use Comment Sheet in the back of this manual

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

## REVISION RECORD (Cont'd)

REVISION	DESCRIPTION
E (5-10-74)	This revision includes new features of Version 4.2 for use under SCOPE 3.4, KRONOS 2.1, and SCOPE 2.1; and it incorporates clarifications and technical and typographical corrections. Pages affected: iii thru x; Part I: 4-3; 5-8, 5-9, 5-10; 6-12, 6-17, 6-18, 6-21, 6-25; 7-2, 7-8, 7-9; 8-1, 8-2, 8-5 thru 8-8, 8-13, 8-15; 9-7, 9-9, 9-11, 9-14; 10-1, 10-6, 10-34; 11-2 thru 11-9; Part III: 2-15 thru 2-19; 5-12, 5-19; 7-4; 11-1 thru 11-3; Index-2, 9, 10, 13; Comment Sheet.
F (10-5-74)	This revision documents Version 4.3 of FORTRAN Extended for use under NOS 1.0, SCOPE 3.4, KRONOS 2.1, and SCOPE 2.1. Changes include the following features: Time-sharing FORTRAN Option, FTN Optimization (Phase III), FORTRAN Extended - SORT Interface, Multiple Arguments for Logical Functions, NAMELIST Rewrite, Multiple-index Capability for Record Manager Advanced Access Files, and Record Manager Word Addressable File Enhancements. Also incorporated are clarifications and technical corrections. Pages affected: Cover, iii thru x.1; Part I: 2-2, 6, 7, 17; 3-2 thru 5, 11, 12; 4-5; 5-5, 8 thru 10; 6-4, 12, 13, 17, 23, 24, 25; 7-2, 7, 8, 18 thru 21; 8-1 thru 21; 9-13 thru 22, 25; 10-4, 7, 8, 13, 16, 24, 25, 31 thru 38; 11-1 thru 10; 13-5, 13, 14, 15; Part II: 1-5, 9, 10, 15, 16, 18, 23, 38 thru 41; Part III: 1-1 thru 24; 2-1 thru 33; 3-1 thru 10; 4-1 thru 5, 9, 11; 5-1 thru 12; 6-1, 10, 11; 7-2, 11; 8-1; 10-1, 2, 5, 6, 7; 11-1 thru 3; 13-1 thru 3; 14-1 thru 4; 15-1, 2; 16-1 thru 5; Index 1 thru 19; Comment Sheet. Specific pages affected by the features are as follows: Time-sharing FORTRAN option: I-11-7, 8, 10; III-1-17 thru 24; 2-1, 14 thru 25; 13-2; 14-1; 15-1, 2. FTN Optimization (Phase III): III-14-1 thru 3. FORTRAN Extended - SORT Interface: III-16-1 thru 5. Multiple Arguments for Logical Functions: I-7, 8; 8-1, 3; 11-8, 9; III-8-1; 10-1, 5. NAMELIST Rewrite: I-6-24; 9-16 thru 19; II-1-10, 18, 39 thru 41; III-2-31, 32. Multiple Index Capability for Record Manager Advanced Access Files: III-6-10, 11. Record Manager Word Addressable File Enhancements: III-5-5, 6; 7-2.
G (3-28-75)	This revision documents Version 4.4 of FORTRAN Extended. Changes include the following features: Math Library Upgrade Phase II, CP079, I/O APLIST Modification, CP123, Dynamic Listing Control and UO option, CP121.
Publication No. 60305600	



## LIST OF EFFECTIVE PAGES

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

Feature	Page	Revision
	Front Cover	—
	Title Page	—
	ii thru xxviii	G
	I-1-1 thru 1-3	G
	1-4	D
	I-2-1 thru 2-17	G
	I-3-1	B
	3-2	G
	3-3, 3-4	F
	3-5	G
	3-6	C
	3-7	B
	3-8	G
	3-9	B
	3-10	A
	3-11, 3-12	G
	3-13 thru 3-15	B
	3-16	A
	I-4-1, 4-2	B
	4-3	E
	4-4	B
	4-5	F
	4-6	B
	I-5-1 thru 5-16	G
	I-6-1 thru 6-24	G
	I-7-1 thru 7-19	G
	I-8-1 thru 8-3	G
	8-4, 8-5	F
CP079	8-6, 8-7	G
	8-8 thru 8-14	G
	8-15	F
	8-16	G
	8-17	F
	8-18	G

Feature	Page	Revision
	I-8-19, 8-20	F
	8-21	G
	I-9-1, 9-2	G
	9-3, 9-4	D
	9-5, 9-6	G
	9-7	E
	9-8	D
	9-9	G
	9-10	D
	9-11 thru 9-13	G
	9-14	F
	9-15 thru 9-27	G
	I-10-1	E
	10-2	G
	10-3	B
	10-4	F
	10-5	B
	10-6	E
	10-7	G
	10-8	F
	10-9 thru 10-11	D
	10-12	A
	10-13	F
	10-14	D
	10-15	A
	10-16	F
	10-17	C
	10-18, 10-19	D
	10-20	A
	10-21, 10-22	G
	10-23	D
	10-24	G
	10-25	F
	10-26 thru 10-28	D
	10-29 thru 10-39	G
	I-11-1 thru 11-7	G
CP121	11-8	G
	11-9	G
CP121	11-10	G
	I-12-1 thru 12-6	G
	12-7, 12-8	C
	12-9	G

Feature	Page	Revision
	I-13-1	C
	13-2	G
	13-3, 13-4	B
	13-5	G
	13-6	B
	13-7	G
	13-8	G
	13-9 thru 13-11	B
	13-12 thru 13-15	G
	13-16, 13-17	B
	13-18	G
	13-19	B
	13-20 thru 13-22	C
	13-23 thru 13-25	B
	13-26 thru 13-28	G
	13-29	B
	13-30	G
	13-31	B
	II-1-1, 1-2	C
	1-3	A
	1-4	B
	1-5	G
	1-6 thru 1-9	B
	1-10	F
	1-11, 1-12	B
	1-13	A
	1-14	B
	1-15	G
	1-16	F
	1-17	C
	1-18	F
	1-19	G
	1-20 thru 1-22	B
	1-23	F
	1-24, 1-25	B
	1-26	A
	1-27	B
	1-28 thru 1-36	A
	1-37	D
	1-38	G
	1-39 thru 1-41	F
	III-1-1 thru 1-4	G
	1-5	B
	1-6	F
	1-7	G
	1-8, 1-9	F
	1-10, 1-11	B
	1-12	F

Feature	Page	Revision
	III-1-13	G
	1-14	F
	1-15	B
	1-16 thru 1-24	F
	III-2-1	G
	2-2	A
	2-3 thru 2-13	F
	2-14 thru 2-31	G
CP079	2-32 thru 2-38	G
CP079	III-3-1 thru 3-5	G
	3-6 thru 3-12	G
	III-4-1	A
	4-2	F
	4-3 thru 4-5	G
	4-6, 4-7	A
	4-8	C
	4-9	G
	4-10	C
	4-11	G
	4-12	B
	III-5-1	F
	5-2 thru 5-5	G
	5-6	F
	5-7 thru 5-12	G
	III-6-1 thru 6-11	G
	III-7-1 thru 7-13	G
	III-8-1	F
	III-9-1	B
	9-2	A
	9-3	G
	III-10-1 thru 10-3	G
	10-4	B
	10-5	G
	10-6, 10-7	F
	III-11-1	G
	11-2	F
	11-3	G
	III-12-1	G
CP121	12-2	G
	12-3 thru 12-7	G
	III-13-1	D
	13-2	G
	13-3 thru 13-11	G
CP121	III-14-1	G
	14-2	G
CP121	14-3	G
	14-4, 14-5	G
	III-15-1, 15-2	G

Feature	Page	Revision
	III-16-1 thru 16-6	G
	A-1	G
	A-2	C
	A-3	B
	Index-1 thru 7	G
	Comment Sheet	G
	Return Env	-
	Back Cover	-

Feature	Page	Revision





# PREFACE

---

This manual describes the FORTRAN Extended 4.4 language. FORTRAN Extended is designed to comply with American National Standards Institute FORTRAN language, as described in X3.9-1966. It is assumed the reader has knowledge of an existing FORTRAN language and is familiar with the computer system on which the language is used.

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

NOS 1.0 operating system for the CONTROL DATA® CYBER 170, CYBER 70/Models 72, 73, 74, and 6000 Series Computer Systems

SCOPE 3.4 and KRONOS 2.1 operating systems for the CDC CYBER 70/Models 72, 73, 74, and 6000 Series Computer Systems

SCOPE 2.1 operating system for the CDC CYBER 70/Model 76 and 7600 Computer Systems

Version 4.4 of FORTRAN Extended provides dynamic compilation listing control, Phase II of the Math Library Upgrade (addition of SINH and COSH, plus further math library improvements), and optional indexed array element prefetching and B-register preservation across FORTRAN Math Library basic external function references.

This manual is in three parts. The reference section, Part I, contains a full description of the FORTRAN Extended language.

Part II consists of a set of sample programs with input cards and output. Each program is preceded by a short introduction which explains some of the more difficult aspects of the language for the less experienced FORTRAN programmer.

Part III contains mainly information related to debugging, various interfaces, and additional details related to the operation of FORTRAN Extended.

## Other Documents of Interest

## Publication Number

COMPASS 3 Reference Manual	60360900
FORTRAN Common Library Mathematical Routines	60387900
FORTRAN Extended DEBUG User's Guide	60329400
INTERCOM 4 Reference Manual	60307100
INTERCOM Interactive Guide for Users of FORTRAN Extended	60359700

KRONOS 2.1 Reference Manual	60407000
KRONOS 2.1 Time-Sharing User's Reference Manual	60407600
LOADER Reference Manual	60344200
NOS 1.0 Reference Manual	60435400
NOS 1.0 Time-Sharing User's Reference Manual	60435500
Record Manager Reference Manual	60307300
Record Manager Guide for Users of FORTRAN Extended	60385200
CYBER Record Manager User's Guide	60359600
SCOPE 3.4 Reference Manual	60307200
SCOPE 2 Reference Manual	60342600
SIFT Programming System Bulletin	60358400
Sort/Merge Reference Manual	60343900
UPDATE Reference Manual	60342500

Throughout the manual, Control Data extensions to the FORTRAN language are indicated by shading. Otherwise, FORTRAN Extended conforms to ANSI standards.

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

# CONTENTS

---

PREFACE	iii		
STATEMENT FORMS	xi		
PART I			
1	CODING FORTRAN STATEMENTS	I-1-1	4
	FORTRAN Character Set	I-1-1	
	FORTRAN Statements	I-1-2	
	Continuation Lines	I-1-2	
	Statement Separator	I-1-2	
	Statement Labels	I-1-3	
	Comments	I-1-3	
	Columns 73-80	I-1-3	
	Blank Lines	I-1-3	
	Data	I-1-3	
2	LANGUAGE ELEMENTS	I-2-1	5
	Constants	I-2-1	
	Integer Constant	I-2-1	
	Real Constant	I-2-2	
	Double Precision Constant	I-2-3	
	Complex Constant	I-2-4	
	Octal Constant	I-2-5	
	Hollerith Constant	I-2-6	
	Logical Constant	I-2-8	
	Variables	I-2-9	
	Integer Variables	I-2-10	
	Real Variables	I-2-10	
	Double Precision Variables	I-2-11	
	Complex Variables	I-2-11	
	Logical Variables	I-2-11	
	Arrays	I-2-12	
	Subscripts	I-2-14	
	Array Structure	I-2-15	
3	EXPRESSIONS	I-3-1	
	Arithmetic Expressions	I-3-2	
	Evaluation of Expressions	I-3-3	
	Type of Arithmetic Expressions	I-3-5	
	Exponentiation	I-3-6	
	Relational Expressions	I-3-7	
	Logical Expressions	I-3-9	
	Masking Expressions	I-3-13	
	ASSIGNMENT STATEMENTS	I-4-1	
	Arithmetic Assignment Statements	I-4-1	
	Conversion to Integer	I-4-2	
	Conversion to Real	I-4-3	
	Conversion to Double Precision	I-4-3	
	Conversion to Complex	I-4-4	
	Logical Assignment	I-4-5	
	Masking Assignment	I-4-5	
	Multiple Assignment	I-4-6	
	CONTROL STATEMENTS	I-5-1	
	GO TO Statement	I-5-1	
	Unconditional GO TO Statement	I-5-1	
	Computed GO TO Statement	I-5-2	
	ASSIGN Statement	I-5-3	
	Assigned GO TO Statement	I-5-4	
	Arithmetic IF Statement	I-5-5	
	Three-Branch Arithmetic IF Statement	I-5-5	
	Two-Branch Arithmetic IF Statement	I-5-5	
	Logical IF Statement	I-5-6	
	Standard-Form Logical IF Statement	I-5-6	
	Two-Branch Logical IF Statement	I-5-7	
	DO Statement	I-5-7	
	DO Loops	I-5-8	
	Nested DO Loops	I-5-9	
	CONTINUE Statement	I-5-13	
	PAUSE Statement	I-5-14	
	STOP Statement	I-5-14	
	END Statement	I-5-15	
	RETURN Statement	I-5-15	
6	SPECIFICATION STATEMENTS	I-6-1	
	Type Statements	I-6-1	
	Explicit Type Statements	I-6-2	

	IMPLICIT Type Statement	I-6-3	PRINT	I-9-3
	DIMENSION Statement	I-6-5	PUNCH	I-9-4
	COMMON Statement	I-6-6	Formatted WRITE	I-9-5
	EQUIVALENCE Statement	I-6-10	Unformatted WRITE	I-9-6
	EQUIVALENCE and COMMON	I-6-13	List Directed WRITE	I-9-7
	LEVEL Statement	I-6-15	INPUT Statements	I-9-7
	EXTERNAL Statement	I-6-16	Formatted READ	I-9-7
	DATA Statement	I-6-19	Unformatted READ	I-9-8
			List Directed READ	I-9-9
7	PROGRAMS, SUBPROGRAMS, AND PROCEDURES	I-7-1	List Directed Input Data Forms	I-9-10
	Main Program	I-7-2	List Directed Output Data Forms	I-9-11
	PROGRAM Statement Format	I-7-2	File Manipulation Statements	I-9-12
	PROGRAM Statement Usage	I-7-3	REWIND	I-9-12
	Block Data Subprogram	I-7-5	BACKSPACE	I-9-12
	Procedures	I-7-6	ENDFILE	I-9-13
	Subroutine Subprogram	I-7-6	BUFFER Statements	I-9-13
	Function Subprogram	I-7-8	NAMELIST	I-9-15
	Basic External Function	I-7-9	Input Data	I-9-17
	Intrinsic Function	I-7-10	Output	I-9-18
	Statement Function	I-7-10	Arrays in NAMELIST	I-9-19
	Procedure Communication	I-7-12	ENCODE and DECODE	I-9-21
	Passing Values to a Procedure	I-7-12	ENCODE	I-9-21
	Using Arguments	I-7-12	DECODE	I-9-24
	Using Common	I-7-14		
	Using Arrays	I-7-14	10 INPUT/OUTPUT LISTS AND FORMAT STATEMENTS	I-10-1
	Referencing a Function	I-7-15	Input/Output Lists	I-10-1
	Calling a Subroutine Subprogram	I-7-16	IMPLIED DO in I/O List	I-10-2
	Using the ENTRY Statement	I-7-18	FORMAT Statement	I-10-5
			Data Conversion	I-10-6
8	FORTRAN EXTENDED SUPPLIED PROCEDURES	I-8-1	Field Separators	I-10-7
	Intrinsic Functions	I-8-1	Conversion Specification	I-10-7
	Basic External Functions	I-8-6	Scale Factors	I-10-22
	Additional Utility Subprograms	I-8-9	X Specification	I-10-24
	Operating System Interface		nH Output	I-10-25
	Routines	I-8-9	nH Input	I-10-26
	Debugging Aids	I-8-15	* . . . * ≠ . . . ≠	I-10-27
	Random Number Generator	I-8-16	FORTRAN Record Slash	I-10-29
	Mass Storage Input/Output	I-8-16	Repeated Format Specification	I-10-31
	Input/Output Status Checking	I-8-18	Printer Control Character	I-10-32
	Other Input/Output Subprograms	I-8-19	Tn Specification	I-10-34
	ECS/LCM Subprograms	I-8-20	V Specification	I-10-35
	Terminal Interface	I-8-21	Equals Sign	I-10-36
	CYBER Record Manager Interface	I-8-21	Execution Time Format	
	Sort/Merge Interface	I-8-21	Statements	I-10-38
9	INPUT/OUTPUT	I-9-1		
	FORTRAN Record Length	I-9-2	11 FORTRAN CONTROL CARD	I-11-1
	Carriage Control	I-9-2	Parameters	I-11-1
	Output Statements	I-9-3	A Exit Parameter	I-11-2
			B Binary Object File	I-11-2
			BL Burstable Listing	I-11-2

C	COMPASS Assembly	I-11-2	FTN	Control Card Samples	I-11-9
D	Debugging Mode Parameter	I-11-3			
E	Editing Parameter	I-11-3	12	OVERLAYS	I-12-1
EL	Error Level	I-11-3		Overlay Communication	I-12-3
G	Get System Text File	I-11-4		Creating an Overlay	I-12-3
GO	Automatic Execution	I-11-4		Calling an Overlay	I-12-5
I	Source Input File	I-11-4			
L	List Output File	I-11-4	13	DEBUGGING FACILITY	I-13-1
LCM	Level 2 and Level 3			Debugging Statements	I-13-3
	Storage Access	I-11-5		Continuation Card	I-13-4
ML	Modlevel	I-11-5		ARRAYS Statement	I-13-4
OL	Object List	I-11-5		CALLS Statement	I-13-6
OPT	Optimization Parameter	I-11-5		FUNCS Statement	I-13-8
P	Pagination	I-11-6		STORES Statement	I-13-11
PL	Print Limit	I-11-6		Variable Names	I-13-12
Q	Program Verification	I-11-6		Relational Operators	I-13-13
R	Symbolic Reference Map	I-11-6		Checking Operators	I-13-14
ROUND	Rounded Arithmetic			Hollerith Data	I-13-14
	Computations	I-11-7		GOTOS Statement	I-13-15
S	System Text (Library) File	I-11-7		TRACE Statement	I-13-16
SEQ	Sequential Input	I-11-7		NOGO Statement	I-13-18
SL	Source List	I-11-7		Debug Deck Structure	I-13-19
SYSEEDIT	System Editing	I-11-8		DEBUG Statement	I-13-24
T	Error Traceback	I-11-8		AREA Statement	I-13-26
TS	Timesharing Mode	I-11-8		OFF Statement	I-13-28
UO	Unsafe Optimization	I-11-8		Printing Debug Output	I-13-30
X	External Text Name	I-11-9		STRACE Entry Point	I-13-30
Z	Zero Parameter	I-11-9			

## PART II

1	SAMPLE PROGRAMS	II-1-1	PROGRAM X	II-1-24
	PROGRAM OUT	II-1-1	PROGRAM VARDIM	II-1-26
	PROGRAM B	II-1-4	PROGRAM VARDIM2	II-1-28
	PROGRAM MASK	II-1-6	SUBROUTINE IOTA	II-1-28
	PROGRAM EQUIV	II-1-9	SUBROUTINE SET	II-1-28
	PROGRAM COME	II-1-11	FUNCTION AVG	II-1-29
	PROGRAM LIBS	II-1-14	FUNCTION PVAL	II-1-30
	PROGRAM PIE	II-1-17	FUNCTION MULT	II-1-30
	PROGRAM ADD	II-1-19	Main Program – VARDIM2	II-1-31
	DECODE (READ)	II-1-19	PROGRAM CIRCLE	II-1-35
	ENCODE (WRITE)	II-1-19	PROGRAM OCON	II-1-37
	PROGRAM PASCAL	II-1-22	List Directed Input/Output	II-1-40

## PART III

1	CROSS REFERENCE MAP	III-1-1	Entry Points	III-1-6
	Optimizing Compilation Mode	III-1-1	Variables	III-1-7
	Source Program	III-1-2	File Names	III-1-9

	External References	III-1-10		FILE Control Card	III-5-6
	Inline Functions	III-1-11		Sequential File Backspace/Rewind	III-5-8
	NAMELISTS	III-1-11		BUFFER Input/Output	III-5-10
	Statement Labels	III-1-12		BUFFER IN	III-5-10
	DO Loops	III-1-13		BUFFER OUT	III-5-11
	Common Blocks	III-1-14		Labeled File Processing	III-5-11
	EQUIVALENCE Classes	III-1-15		Programming Notes	III-5-12
	Program Statistics	III-1-16			
	Error Messages	III-1-16	6	FORTTRAN – CYBER RECORD	
TS Mode		III-1-17		MANAGER INTERFACE	III-6-1
	Common Blocks	III-1-20		File Information Table Calls	III-6-1
	Entry Points	III-1-20		File Commands	III-6-3
	External References	III-1-20		Updating File Information Table	III-6-3
	Statement Labels	III-1-21		Key Hashing Subroutine	III-6-8
	Variables	III-1-22		Error Checking	III-6-9
	Blocks	III-1-23		Multiple Index Processing	III-6-10
2	FORTTRAN DIAGNOSTICS	III-2-1	7	MASS STORAGE INPUT/OUTPUT	III-7-1
	Compilation Diagnostics	III-2-1		Random File Access	III-7-1
	Special Compilation Diagnostics	III-2-14		Index Key Types	III-7-2
	Compilation Diagnostics, TS Mode	III-2-18		Multi-Level File Indexing	III-7-5
	Execution Diagnostics	III-2-31		Master Index	III-7-6
3	EXECUTION-TIME PROCESSING	III-3-1		Sub-Index	III-7-6
	Error Processing	III-3-1		Mass Storage Subroutine	III-7-9
	Extended Error Processing	III-3-1		OPENMS	III-7-9
	SYSTEM	III-3-1		WRITMS	III-7-9
	SYSTEMC	III-3-2		READMS	III-7-10
	ERRSET	III-3-7		CLOSMS	III-7-11
	Execution-Time Options	III-3-9		STINDX	III-7-11
	File Name Handling	III-3-9		Compatibility with Previous Mass	
	Print Limit Specification	III-3-11		Storage Routines	III-7-12
				Error Messages	III-7-13
4	ARITHMETIC	III-4-1	8	RENAMING CONVENTIONS	III-8-1
	Floating Point Arithmetic	III-4-1		Register Names	III-8-1
	Overflow (+ $\infty$ or - $\infty$ )	III-4-3		External Procedure Names	
	Underflow (+0 or -0)	III-4-3		(Processor Supplied)	III-8-1
	Indefinite Result	III-4-4		Call-by-Value	III-8-1
	Nonstandard Floating Point			Call-by-Name	III-8-1
	Arithmetic	III-4-5			
	Integer Arithmetic	III-4-7	9	PROGRAM AND MEMORY	
	Double Precision	III-4-7		STRUCTURE	III-9-1
	Complex	III-4-8		Subroutine and Function Structure	III-9-2
	Logical and Masking	III-4-8		Main Program Structure	III-9-3
	Arithmetic Errors	III-4-8		Memory Structure	III-9-3
5	EXECUTION-TIME INPUT/OUTPUT	III-5-1	10	INTERMIXED COMPASS	
	File and Record Definitions	III-5-1		SUBPROGRAMS	III-10-1
	Structure of Input/Output Files	III-5-2		Call by Name and Call by Value	III-10-1
	Sequential Files	III-5-2		Call by Name Sequence	III-10-1
	Random Files	III-5-6		Call by Value Sequence	III-10-2

	COMPASS Subprograms	III-10-2	Compilation and Two Executions with Overlays	III-13-11
	Entry Point	III-10-5		
	Restrictions on Using Library Function Names	III-10-5	14 COMPILATION MODES AND OPTIMIZATION	III-14-1
11	TERMINAL I/O WITH FORTRAN	III-11-1	Object Code Optimization	III-14-2
			Source Code Optimization	III-14-3
12	LISTINGS	III-12-1	15 TIME-SHARING FORTRAN	III-15-1
	FORTRAN Listing Control	III-12-2	Source Listing Format	III-15-1
	DMPX	III-12-3	Sequenced Line Format	III-15-2
13	SAMPLE DECK STRUCTURES	III-13-1	16 FORTRAN – SORT/MERGE	
	FORTRAN Source Program with Control Cards	III-13-1	INTERFACE	III-16-1
	Compilation Only	III-13-2	SORT	III-16-1
	TS Mode Compilation Only	III-13-2	SORTB	III-16-1
	Compilation and Execution	III-13-3	SORTP	III-16-1
	FORTRAN Compilation with COMPASS Assembly and Execution	III-13-4	MERGE	III-16-2
	Compile and Execute with FORTRAN Subroutine and COMPASS Subprogram	III-13-5	FILE	III-16-2
	Compile and Produce Binary Cards	III-13-6	KEY	III-16-3
	Load and Execute Binary Program	III-13-7	Sequence	III-16-3
	Compile and Execute with Relocatable Binary Deck	III-13-8	Equate	III-16-4
	Compile Once and Execute with Different Data Decks	III-13-9	Options	III-16-4
	Preparation of Overlays	III-13-10	TAPE	III-16-5
			Owncode	III-16-5
			END	III-16-6
			A STANDARD CHARACTER SETS	A-1
			INDEX	Index-1

## TABLES

### PART I

3-1	Mixed Type Arithmetic Expressions with + - * / Operators	I-3-5	7-2	Procedure and Subprogram Interrelationships	I-7-2
7-1	Differences Between a Function and Subroutine Subprogram	I-7-1	8-1	Intrinsic Functions	I-8-2
			8-2	Basic External Functions	I-8-7

### PART III

5-1	Defaults for FIT Fields under FORTRAN Extended	III-5-3
-----	---	---------

## FIGURES

### PART I

1-1 Program PASCAL	I-1-4	13-2 External Debugging Deck	I-13-21
13-1 Example of Interspersed Debugging Statements	I-13-20	13-3 Internal Debugging Deck	I-13-22
		13-4 External Deck on Separate File	I-13-23



# STATEMENT FORMS

---

The following symbols are used in the descriptions of FORTRAN Extended statements:

v	variable or array element
sn	statement label
iv	integer variable
m	unsigned integer or octal constant or integer variable
name	symbolic name
u	input/output unit: 1- or 2-digit decimal integer constant, integer variable with value of: 0-99, or a Hollerith value which is the filename left justified with zero fill
fn	format designator
iolist	input/output list

Other forms are defined individually in the following list of statements.

	Page Numbers
<b>ASSIGNMENT STATEMENTS</b>	
v = arithmetic expression	I-4-1
logical v = logical or relational expression	I-4-5
v = masking expression	I-4-5
<b>MULTIPLE ASSIGNMENT</b>	
$v_1 = v_2 = \dots v_n = \text{expression}$	I-4-6
<b>CONTROL STATEMENTS</b>	
GO TO sn	I-5-1
GO TO (sn <sub>1</sub> , ..., sn <sub>m</sub> ), iv	I-5-2
GO TO (sn <sub>1</sub> , ..., sn <sub>m</sub> ), expression	I-5-2

	Page Numbers
GO TO iv, (sn <sub>1</sub> , ..., sn <sub>m</sub> )	I-5-4
GO TO iv (sn <sub>1</sub> , ..., sn <sub>m</sub> )	I-5-4
ASSIGN sn TO iv	I-5-3
IF (arithmetic or masking expression) sn <sub>1</sub> , sn <sub>2</sub> , sn <sub>3</sub>	I-5-5
IF (arithmetic or masking expression) sn <sub>1</sub> , sn <sub>2</sub>	I-5-5
IF (logical or relational expression) stat	I-5-6
IF (logical or relational expression) sn <sub>1</sub> , sn <sub>2</sub>	I-5-7
DO sn iv = m <sub>1</sub> , m <sub>2</sub> , m <sub>3</sub>	I-5-7
DO sn iv = m <sub>1</sub> , m <sub>2</sub>	I-5-7
CONTINUE	I-5-13
PAUSE	I-5-14
PAUSE n	I-5-14
PAUSE ≠c...c≠	I-5-14
STOP	I-5-14
STOP n	I-5-14
STOP ≠c...c≠	I-5-14
END	I-5-15
<b>TYPE DECLARATION</b>	
INTEGER name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2
TYPE INTEGER name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2

	Page Numbers
REAL name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2
TYPE REAL name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2
COMPLEX name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2
TYPE COMPLEX name <sub>1</sub> , ..., name <sub>n</sub>	I-6-2
DOUBLE PRECISION name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
DOUBLE name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
TYPE DOUBLE PRECISION name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
TYPE DOUBLE name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
LOGICAL name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
TYPE LOGICAL name <sub>1</sub> , ..., name <sub>n</sub>	I-6-3
IMPLICIT type <sub>1</sub> (ac) , ..., type <sub>n</sub> (ac)	I-6-3
<p>(ac) is a single alphabetic character or range of characters represented by the first and last character separated by a minus sign.</p>	
<b>EXTERNAL DECLARATION</b>	
EXTERNAL name <sub>1</sub> , ..., name <sub>n</sub>	I-6-16
<b>STORAGE ALLOCATION</b>	
type name <sub>1</sub> (d <sub>i</sub> )	I-6-1
TYPE type name <sub>1</sub> (d <sub>i</sub> )	I-6-1
DIMENSION name <sub>1</sub> (d <sub>1</sub> ), ..., name <sub>n</sub> (d <sub>n</sub> )	I-6-5
d <sub>i</sub>	array declarator, one to three integer constants; or in a subprogram, one to three integer variables
type	INTEGER, REAL, COMPLEX, DOUBLE, DOUBLE PRECISION or LOGICAL

COMMON  $v_1, \dots, v_n$  I-6-6

COMMON/blkname<sub>1</sub>/ $v_1, \dots, v_n \dots$  /blkname<sub>n</sub>/ $v_1, \dots, v_n$  I-6-6

COMMON//  $v_1, \dots, v_n$  I-6-6

blkname      symbolic name or 1 - 7 digits

//            blank common

DATA vlist<sub>1</sub>/dlist<sub>1</sub>/, . . . , vlist<sub>n</sub>/dlist<sub>n</sub>/ I-6-19

DATA (vlist=dlist), . . . , (vlist=dlist) I-6-19

vlist          list of array names, array elements, variable names, or implied DO list,  
separated by commas

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

- constant
- (constant list)
- rf\*constant
- rf\*(constant list)
- rf(constant list)

constant list      list of constants separated by commas

rf                integer constant. The constant or constant list is repeated  
the number of times indicated by rf

EQUIVALENCE (glist<sub>1</sub>), . . . , (glist<sub>n</sub>) I-6-10

LEVEL n, a<sub>1</sub>, . . . , a<sub>n</sub> I-6-15

n                unsigned integer 1, 2 or 3

a                variable, array element, array name

## MAIN PROGRAMS

Page  
Numbers

PROGRAM name

I-7-2

PROGRAM name(fpar<sub>1</sub>, fpar<sub>2</sub>, . . . , fpar<sub>k</sub>)

I-7-2

## SUBPROGRAMS

FUNCTION name (p<sub>1</sub>, . . . , p<sub>n</sub>)

I-7-8

type FUNCTION name (p<sub>1</sub>, . . . , p<sub>n</sub>)

I-7-8

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

SUBROUTINE name (p<sub>1</sub>, . . . , p<sub>n</sub>)

I-7-6

SUBROUTINE name

I-7-6

SUBROUTINE name (p<sub>1</sub>, . . . , p<sub>n</sub>), RETURNS (b<sub>1</sub>, . . . , b<sub>m</sub>)

I-7-6

SUBROUTINE name, RETURNS (b<sub>1</sub>, . . . , b<sub>m</sub>)

I-7-6

## ENTRY POINT

ENTRY name

I-7-18

## STATEMENT FUNCTIONS

name (p<sub>1</sub>, . . . , p<sub>n</sub>) = expression

I-7-10

## SUBPROGRAM CONTROL STATEMENTS

CALL name

I-7-16

CALL name (p<sub>1</sub>, . . . , p<sub>n</sub>)

I-7-16

CALL name (p<sub>1</sub>, . . . , p<sub>n</sub>), RETURNS (b<sub>1</sub>, . . . , b<sub>m</sub>)

I-7-16

CALL name, RETURNS (b<sub>1</sub>, . . . , b<sub>m</sub>)

I-7-16

RETURN

I-5-15

RETURN i

I-5-15

i            is a dummy argument in a RETURNS list

**SPECIFICATION SUBPROGRAMS**

BLOCK DATA I-7-5

BLOCK DATA name I-7-5

**INPUT/OUTPUT**

PRINT fn,iolist I-9-3

PRINT fn I-9-3

PRINT (u,fn) iolist I-9-3

PRINT\*,iolist I-9-3

PRINT (u,fn) I-9-3

PRINT (u,\*) iolist I-9-3

PUNCH fn,iolist I-9-4

PUNCH fn I-9-4

PUNCH (u,fn) iolist I-9-4

PUNCH\*,iolist I-9-4

PUNCH (u,fn) I-9-4

PUNCH (u,\*) iolist I-9-4

WRITE (u,fn) iolist I-9-5

WRITE (u,fn) I-9-5

WRITE fn,iolist I-9-5

WRITE fn I-9-5

WRITE (u) iolist I-9-6

WRITE (u) I-9-6

WRITE (u,\*) iolist I-9-7

WRITE\*,iolist I-9-7

	Page Numbers
READ (u,fn)iolist	I-9-7
READ (u,fn)	I-9-7
READ fn,iolist	I-9-8
READ (u) iolist	I-9-8
READ (u)	I-9-8
READ (u,*) iolist	I-9-9
READ*,iolist	I-9-9
BUFFER IN (u,p) (a,b)	I-9-13
BUFFER OUT (u,p) (a,b)	I-9-15
a	first word of data block to be transferred
b	last word of data block to be transferred
p	integer constant or integer variable. zero = even parity, nonzero = odd parity
NAMELIST/group name <sub>1</sub> /a <sub>1</sub> , ..., a <sub>n</sub> /.../group name <sub>n</sub> /a <sub>1</sub> , ..., a <sub>n</sub>	I-9-16
READ (u,group name)	I-9-17
WRITE (u,group name)	I-9-19
a <sub>i</sub>	array names or variables
group name	symbolic name identifying the group a <sub>1</sub> , ..., a <sub>n</sub>
 <b>INTERNAL TRANSFER OF DATA</b>	
ENCODE (c,fn,v) iolist	I-9-22
DECODE (c,fn,v) iolist	I-9-25
v	starting location of record. Variable or array name
c	length of record in characters. Unsigned integer constant or simple integer variable

## FILE MANIPULATION

REWIND u	I-9-12
BACKSPACE u	I-9-12
ENDFILE u	I-9-13

## FORMAT SPECIFICATION

sn FORMAT ( $fs_1, \dots, fs_n$ )	I-10-5
-----------------------------------	--------

$fs_i$  one or more field specifications separated by commas and/or grouped by parentheses

## DATA CONVERSION

srEw.d	Single precision floating point with exponent	I-10-9,11
srEw.dEe	Floating point with specified exponent length	I-10-9,11
srEw.dDe	Floating point with specified exponent length	I-10-9,11
srFw.d	Single precision floating point without exponent	I-10-13,14
srGw.d	Single precision floating point with or without exponent	I-10-15
srDw.d	Double precision floating point with exponent	I-10-16,17
rlw	Decimal integer conversion	I-10-8
rlw.z	Integer with specified minimum digits	I-10-8
rLw	Logical conversion	I-10-22
rAw	Alphanumeric conversion	I-10-19,20
rRw	Alphanumeric conversion	I-10-21
rOw	Octal integer conversion	I-10-17,18
rOw.z	Integer with specified minimum digits	I-10-19
rZw	Hexadecimal conversion	I-10-19



	Page Numbers
srVw.d    Variable type conversion	I-10-35
s    optional scale factor of the form: nP	
r    optional repetition factor	
w    integer constant indicating field width	
d    integer constant indicating digits to right of decimal point	
e    integer indicating digits in exponent field	
z    integer specifying minimum number of digits	
nX        Intraline spacing	I-10-24
nH ... } * ... * } ≠ ... ≠ }	Hollerith I-10-26
/        Format field separator; indicates end of FORTRAN record	I-10-29
Tn       Column tabulation	I-10-34
V        Display code substitution	I-10-35
=        Numeric substitution	I-10-36
FORTRAN Control Card	I-11-1

## OVERLAYS

CALL OVERLAY (fname,i,j,recall,k)	I-12-5
i        primary overlay number	
j        secondary overlay number	
recall    if 6HRECALL is specified, the overlay is not reloaded if it is already in memory	
k        L format Hollerith constant: name of library from which overlay is to be loaded	
any other non-zero value: overlay loaded from global library set	
OVERLAY (fname,i,j,Cn)	I-12-4
i        primary overlay number, octal	
j        secondary overlay number, octal	
Cn       n is a 6-digit octal number indicating start of load relative to blank common	

**DEBUG**

C\$ DEBUG I-13-24

C\$ DEBUG (name<sub>1</sub>, ..., name<sub>n</sub>) I-13-24

C\$ AREA bounds<sub>1</sub>, ..., bounds<sub>n</sub> } within program unit  
 C\$ DEBUG

C\$ AREA/name<sub>1</sub>/bounds<sub>1</sub>, ..., bounds<sub>n</sub>, ... /name<sub>n</sub>/bounds<sub>1</sub>, ..., bounds<sub>n</sub> } external  
 C\$ DEBUG (name<sub>1</sub>, ..., name<sub>n</sub>) } debug deck  
 or

C\$ DEBUG

- bounds (n<sub>1</sub>,n<sub>2</sub>) n<sub>1</sub> initial line position  
   n<sub>2</sub> terminal line position
- (n<sub>3</sub>) n<sub>3</sub> single line position to be debugged
- (n<sub>1</sub>,\*) n<sub>1</sub> initial line position  
   \* last line of program
- (\*,n<sub>2</sub>) \* first line of program  
   n<sub>2</sub> terminal line position
- (\*,\*) \* first line of program  
   \* last line of program

C\$ ARRAYS (a<sub>1</sub>, ..., a<sub>n</sub>) I-13-4

C\$ ARRAYS I-13-4

a<sub>i</sub> array names

C\$ CALLS (s<sub>1</sub>, ..., s<sub>n</sub>) I-13-6

C\$ CALLS I-13-6

s<sub>i</sub> subroutine names

C\$ FUNCS (f<sub>1</sub>, ..., f<sub>n</sub>) I-13-8

C\$ FUNCS I-13-8

f<sub>i</sub> function name

C\$ GOTOS I-13-15

C\$ NOGO I-13-18

C\$ STORES ( $c_1, \dots, c_n$ ) I-13-11

$c_i$  variable name  
 variable name .relational operator. constant  
 variable name .relational operator. variable name  
 variable name .checking operator.

checking operators:

RANGE out of range  
 INDEF indefinite  
 VALID out of range or indefinite

C\$ TRACE (lv) I-13-16

C\$ TRACE I-13-16

lv level number:  
 0 tracing outside DO loops  
 n tracing up to and including level n in DO nest

C\$ OFF I-13-28

C\$ OFF ( $x_1, \dots, x_n$ ) I-13-28

$x_i$  any debug option

COMPASS SUBPROGRAM IDENTIFICATION

IDENT name (in column 11) III-10-2

END (in column 11) III-10-2



---

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

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

## FORTRAN CHARACTER SET

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

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

## FORTRAN STATEMENTS

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

## CONTINUATION

Statements are coded in columns 7-72. If a statement is longer than 66 columns, it can be continued on as many as 19 lines. A character other than blank or zero in column 6 indicates a continuation line. Column 1 can contain any character other than C, \*, or \$; columns 2, 3, 4, and 5 can contain any character. Any statement except a comment can be continued, including the END statement.

## STATEMENT SEPARATOR

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

```

7
ACUM=24.$I=0 $ IDIFF=1970-1626

```

is the same as

```

7
ACUM = 24.
I = 0
IDIFF = 1970-1626

```

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

## STATEMENT LABELS

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

## COMMENTS

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

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

## COLUMNS 73-80

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

## BLANK LINES

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

## DATA

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





## CONSTANTS

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

### INTEGER CONSTANT

$$\boxed{n_1 n_2 \dots n_m}$$

$n$  is a numeric digit

$1 \leq m \leq 18$  decimal digits

Examples:

237      -74      +136772      0      -0024

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

Examples of invalid integer constants:

46.            (decimal point not allowed)

23A            (letter not allowed)

7,200          (comma not allowed)

When an integer constant is used as a subscript, as the index in a DO statement, or as an implied DO, the maximum value is  $2^{17}-2$  ( $2^{17}-2 = 131\,070$ ), and the minimum is 1.

Integers used in multiplication and division should not have a value greater than  $2^{48}-1$ . The result of integer multiplication or division should be less than  $2^{48}-1$ . If an operand or the result is larger than  $2^{48}-1$  ( $2^{48}-1 = 281\,474\,976\,710\,655$ ), the result is unpredictable; no diagnostic is provided. The resultant maximum value of conversion from real to integer or integer to real numbers is  $2^{48}-1$ . If the value exceeds  $2^{48}-1$ , the high-order bits are lost and no diagnostic is provided. For integer addition and subtraction, the full 60-bit word is used.

## REAL CONSTANT

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

n                    Coefficient ≤ 15 decimal digits

E ± s                Exponent, the + sign is optional

s                    Base 10 scale factor

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

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

Examples:

7.5      -3.22      +4000.      23798.14      .5      - .72      42.E1      700.E-2

Examples of invalid real constants:

3,50.            (comma not allowed)

2.5A            (letter not allowed)

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

Examples:

42.E1            ( $42. \times 10^1 = 420.$ )

.00028E+5        ( $.00028 \times 10^5 = 28.$ )

6.205E12        ( $6.205 \times 10^{12} = 6205000000000.$ )

8.0E+6            ( $8. \times 10^6 = 8000000.$ )

700.E-2          ( $700. \times 10^{-2} = 7.$ )

7E20             ( $7. \times 10^{20} = 70\,000\,000\,000\,000\,000\,000.$ )

Example of invalid real constants:

7.2E3.4          exponent not an integer

## DOUBLE PRECISION CONSTANT

$n.nD\pm s$	$.nD\pm s$	$n.D\pm s$	$nD\pm s$
-------------	------------	------------	-----------

n	Coefficient
$D\pm s$	Exponent, if s is positive the + sign is optional
s	Base 10 scale factor

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

Examples:

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

Examples of invalid double precision constants:

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

## COMPLEX CONSTANT

$(r1, r2)$

r1                    Real part

r2                    Imaginary part

Each part has the same range as a real constant.

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

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

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

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

Examples of invalid complex constants:

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

Real constants which form the complex constant may range from  $10^{-293}$  to  $10^{+322}$ .

## OCTAL CONSTANT

$n_1 \dots n_m B$

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

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

Examples:

777777B

52525252B

500127345B

Invalid octal constants:

892777B

8 and 9 are non-octal digits

7700000007777752525252B

exceeds 20 digits

O7766

O not allowed

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

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

Examples:

BAT = (I\*5252B) .OR. JAY

masking expression

J = MAXO (I, 1000B, J, K+40B)

octal constant used as parameter in function

NAME = I .AND. 77700000B

masking expression

J = (5252B + N) / K

arithmetic expression

DIMENSION BUF(1000B)

dimension specification

When an octal constant is used in an expression, it assumes the type of the dominant operand of the expression (Table 3-1, section 3).

## HOLLERITH CONSTANT

nHf	nLf
nRf	≠f≠

n	Unsigned decimal integer representing number of characters in string. Must be greater than zero, and not more than 10 when used in an expression.
f	String of characters
≠	String delimiter
H	Left justified with blank fill
L	Left justified with binary zero fill
R	Right justified with binary zero fill

```

5 | 7
-----
PROGRAM HOLL (OUTPUT)
A = 6HABCDEF
B = 6LABCDEF
C = 6RABCDEF
D = ≠ABCDEF≠
PRINT 1, A,A,B,B,C,C,D,D
1 FORMAT (024,A15)
STOP
END

```

Stored Internally:

Display Code:

01020304050655555555	ABCDEF	H format
01020304050600000000	ABCDEF::::	L format
00000000010203040506	:::ABCDEF	R format
01020304050655555555	ABCDEF	≠ format

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

5HLABEL                      ≠LABEL≠

nHf

The integer n represents the number of characters in the string f including spaces (or blanks). Spaces are significant only after the H, L, or R in a Hollerith constant.

18HTHIS IS A CONSTANT

7HTHE END

19HRESULT NUMBER THREE

Hollerith constants may be used in arithmetic expressions, DATA and FORMAT statements, as arguments in subprogram calls, and as list items in an output list of an input/output statement. If a Hollerith constant is used as an operand in an arithmetic operation, an informative diagnostic is given.

In an expression, a Hollerith constant is limited to 10 characters.

A Hollerith string delimited by the paired symbols ≠ ≠ can be used anywhere the H form of the Hollerith constant can be used. For example,

```
IF(V.EQ.≠YES≠) Y=Y+1.

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

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

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

The symbol ≠ can be represented within the string by two successive ≠ symbols.

An empty string such as 0H or ≠≠ is not permitted.

When the number of characters in a Hollerith constant is less than 10, the computer word is left justified with blank fill. If it is more than 10, but not a multiple of 10, only the last computer word is left justified with blank fill.

Examples:

```
      7
      |-----|
1    | READ 1,NAME
      |
1    | FORMAT (A7)
      |
      | IF(NAME .EQ. 4HJOAN) GO TO 20
```

```
      7
      |-----|
1000| WRITE (6,1000)
      |
1000| FORMAT (1X, ≠NO COUNTRY THAT HAS BEEN THOROUGHLY EXPLORED IS
      |
1000| S INFESTED WITH DRAGONS.≠)
```

nRf and nLf

A Hollerith constant of the form R or L is limited to 10 characters and cannot be used in a FORMAT statement.

## LOGICAL CONSTANT

A logical constant takes the forms:

.TRUE. or .T. representing the value true

.FALSE. or .F. representing the value false

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

Examples:

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



## VARIABLES

A variable represents a quantity whose value can be varied; this value can be changed repeatedly during program execution. Variables are identified by a symbolic name of one to seven letters or digits, beginning with a letter. A variable is associated with a storage location; whenever a variable is used, it references the value currently in that location.

A variable can have its type specified in a type statement (see section I-6) as integer, real, double precision, complex, or logical. In the absence of an explicit declaration, the type is implied by the first character of the name: I, J, K, L, M, and N imply type integer and any other letter implies type real, unless IMPLICIT statements (see section I-6) are used to change this normal implicit type.

Example:

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

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

## DEFAULT TYPING OF VARIABLES

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

## INTEGER VARIABLES

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

The value range is  $-2^{59}-1$  to  $2^{59}-1$ . When an integer variable is used as a subscript, the maximum value is  $2^{17}-1$ . The resultant absolute value of conversion from integer to real, or real to integer must be less than  $2^{48}$ . The operands, as well as the result, of an integer multiply or division must be less than  $2^{48}$  in absolute value. If this value is exceeded, the results are unpredictable. The resultant absolute value of integer addition or subtraction must be less than  $2^{59}$ .

An integer variable occupies one word of memory.

Examples:

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

## REAL VARIABLES

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

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

Examples:

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

## DOUBLE PRECISION VARIABLES

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

Double precision variables occupy two consecutive words of memory.

Example:

```
DOUBLE PRECISION OMEGA,X,IOTA  
IMPLICIT DOUBLE PRECISION(A)
```

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

## COMPLEX VARIABLES

Complex variables must be typed by a type declaration. A complex variable occupies two words of memory. Each word contains a real number and each number can range from  $10^{-293}$  to  $10^{+322}$ .

Example:

```
COMPLEX ZERA,MU,LAMBDA
```

## LOGICAL VARIABLES

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

Example:

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

## ARRAYS

A FORTRAN array is a set of elements identified by a single name composed of one to seven letters and digits beginning with a letter. Each array element is referenced by the array name and a subscript. The type of the array elements is determined by the array name in the same manner as the type of a variable is determined by the variable name (see Variables in this section). The array name and its dimensions must be declared in a DIMENSION or COMMON statement or a type declaration. Arrays can have one, two, or three dimensions.

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

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

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

Example1:

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

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

Example2:

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

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

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

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

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

Example3:

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

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

## SUBSCRIPTS

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

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

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

The amount of storage allocated to arrays is discussed under DIMENSION declarations in section I-6.

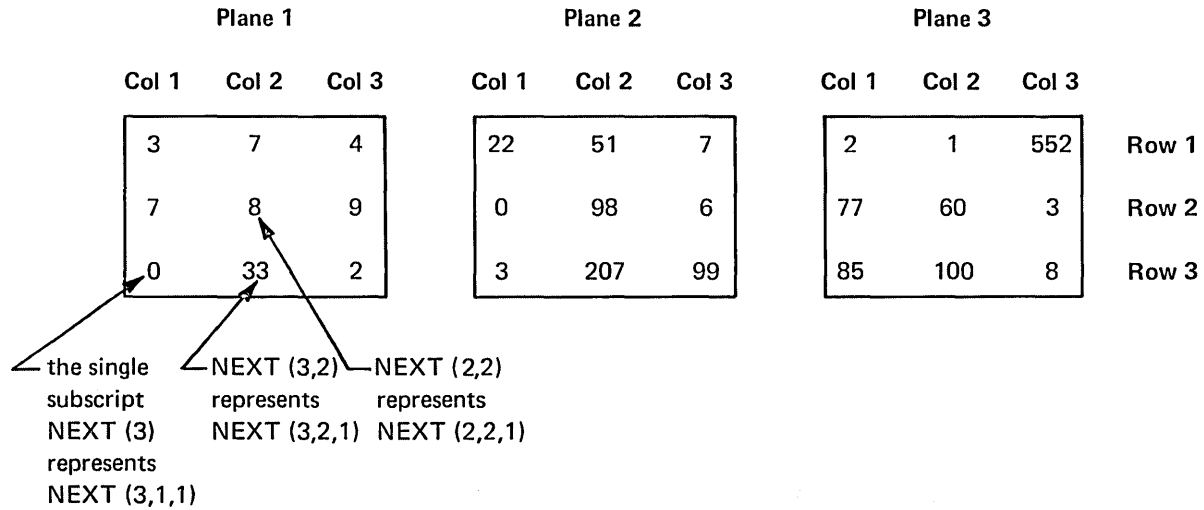
Valid subscript forms:

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

Invalid subscript forms:

```
ATLAS(0)      zero subscript causes a reference outside of the array
D(1 .GE. K)   relational or logical expression illegal
A(,I) or A(I,,K) only trailing subscript expressions can be omitted
```

Example:



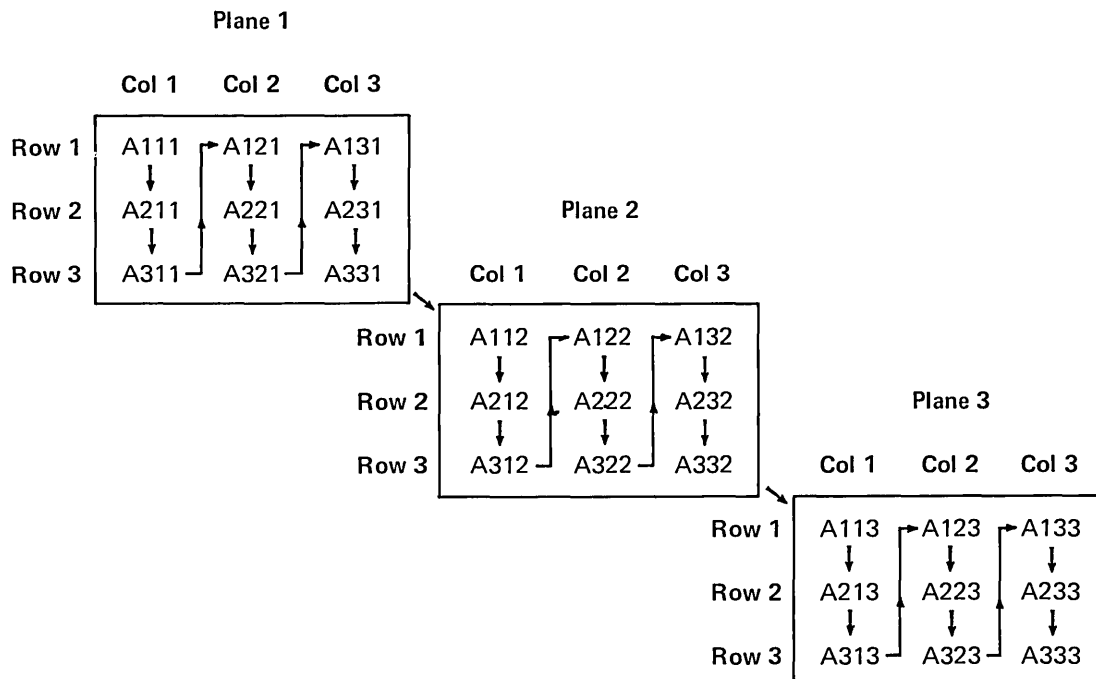
In the three-dimensional array NEXT when only one subscript is shown, the remaining subscripts are assumed to be one.

## ARRAY STRUCTURE

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

Example:

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



The array is stored in linear sequence as follows:

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

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

Number of Dimensions	Array Dimension	Subscript	Location of Element Relative to Starting Location
1	ALPHA(K)	ALPHA(k)	$(k-1) \times E$
2	ALPHA(K,M)	ALPHA(k,m)	$(k-1 + K \times (m-1)) \times E$
3	ALPHA(K,M,N)	ALPHA(k,m,n)	$(k-1 + K \times (m-1 + M \times (n-1))) \times E$

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

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



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

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

Examples:

	<b>Subscript</b>	<b>Location of Element Relative to Starting Location</b>
INTEGER ALPHA (3)	ALPHA(2)	$(2-1) \times 1 = 1$
REAL ALPHA (3,3)	ALPHA(3,1)	$(3-1+3 \times (1-1)) \times 1 = 2$
REAL ALPHA (3,3,3)	ALPHA(3,2,1)	$(3-1+3 \times (2-1+3 \times (1-1))) \times 1 = 5$



---

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

## ARITHMETIC EXPRESSIONS

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

$(A-B)*F+C/D**E$  is a valid arithmetic expression

FORTRAN arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

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

$X+Y$	$X-Y$
$X*Y$	$X/Y$
$-X$	$X**Y$
$+X$	

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

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

## EVALUATION OF EXPRESSIONS

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

1. References to external functions are evaluated.
2. Arithmetic statement functions and intrinsic functions are expanded.
3. Subexpressions delimited by parentheses are evaluated, beginning with the innermost subexpressions.
4. Subexpressions defined by arithmetic, relational, and logical operators are evaluated according to the following precedence hierarchy:

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

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

Unary addition and subtraction are treated as operations on an implied zero. For example,  $+2$  is treated as  $0+2$ , and  $-3$  is treated as  $0-3$ .

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

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

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

Zero-value quantity raised to a zero-value exponent

Infinite or indefinite operand (section 4, part 3)

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

If the error traceback option is selected on the FTN control card (section 11), the first three conditions will produce informative diagnostics during execution. If the traceback option is not selected, a mode error message is printed (section 4, part 3).

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

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

Each left parenthesis must have a corresponding right parenthesis.

Example:

$(F - (X * Y)$	Incorrect, right parenthesis missing
$(F - (X * Y))$	Correct

Examples:

In the expression  $A-B*C$

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

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

D is raised to the power of E.

A is divided by B.

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

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

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

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

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

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

Example:

$$I = 4 \quad J = 3 \quad K = 2$$

$$J/K*I \quad I*J/K$$

$$3/2*4 = 4 \quad 4*3/2 = 6$$

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

Example:

$$N = 24 \quad M = 27 \quad K = 2$$

$$N/M*K$$

$$24/27*2 = 0$$

Examples of valid expressions:

A

3.14159

B + 16.427

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

-(C + DELTA \* AERO)

$(-B - \text{SQRT}(B**2 - (4*A*C))) / (2.0*A)$

GROSS - (TAX\*0.04)

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

### TYPE OF ARITHMETIC EXPRESSIONS

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

Complex

Double Precision

Real

Integer

Table 3-1. Mixed Type Arithmetic Expressions with + - \* / Operators

1st operand \ 2nd operand	Integer	Real	Double Precision	Complex	Octal or Hollerith Constant
Integer	Integer	Real	Double Precision	Complex	Integer
Real	Real	Real	Double Precision	Complex	Real
Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Precision
Complex	Complex	Complex	Complex	Complex	Complex
Octal or Hollerith Constant	Integer	Real	Double Precision	Complex	Integer

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

When an octal or Hollerith constant is used, type is not converted. When these constants are the only operands in an expression, the result of the expression is type integer.

## EXPONENTIATION

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

Base	Exponent
Integer	Integer, Real, Double Precision, Complex
Real	Integer, Real, Double Precision, Complex
Double Precision	Integer, Real, Double Precision, Complex
Complex	Integer

The exponentiation is evaluated from left to right. The expression  $A^{**}B^{**}C$  is, in effect,  $((A^{**}B)^{**}C)$

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

Type of A	Type of B	Type of Result of $A^{**}B$
Integer	Integer Real Double Complex	Integer Real Double Complex
Real	Integer Real Double Complex	Real Real Double Complex
Double	Integer Real Double Complex	Double Double Double Complex
Complex	Integer	Complex

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

$B^{**}2.$

$B^{**}N$

$B^{**}(2*N-1)$

$(A+B)^{**}(-J)$

A negative exponent must be enclosed in parentheses:

$A^{**}(-B)$

$NSUM^{**}(-J)$



Examples:

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

## RELATIONAL EXPRESSIONS

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

$a_1, a_2$           Arithmetic or masking expression

op                  Relational operator

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

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

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

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

`X+Y .GT. 5.3`

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

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

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

Examples:

`J.LT.ITEM`

`580.2 .GT. VAR`

`B .GE. (2.7,5.9E3)` real part of complex number is used in evaluation

`E.EQ..5`

`(I) .EQ. (J(K))`

`C.LT. 1.5D4`

most significant part of double precision number is used in evaluation

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

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

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

`A .GT. B`

`A .GT. (B)`

`(A) .GT. B`

`(A) .GT. (B)`

Examples:

REAL A  
A.GT.720

AMT .LT. (1.,6.55)

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

DOUBLE PRECISION BILL, PAY  
BILL .LT. PAY

(I).EQ.(N\*J)

A+B.GE.Z\*\*2

B.LE.3.754

300.+B.EQ.A-Z

Z.LT.35.3D+5

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

Examples of invalid expressions:

A .GT. 720 .LE. 900

2 relational operators must not appear in a relational expression

B .LE. 3.754 .EQ. C

## LOGICAL EXPRESSIONS

$L_1 \text{ op } L_2 \text{ op } L_3 \text{ op } \dots L_n$
---

$L_1 \dots L_n$       logical operand or relational expression

op                  logical operator

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

Logical operators:

.NOT. or .N.                  logical negation

.AND. or .A.                  logical multiplication

.OR. or .O.                  inclusive OR

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

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

.NOT.p                      If p is true, .NOT.p has the value false. If p is false, .NOT.p has the value true.

p.AND.q                    If p and q are both true, p.AND.q has the value true. Otherwise, false.

p.OR.q                      If either p or q, or both, are true then p.OR.q has the value true. If both p and q are false, then p.OR.q has the value false.

Truth Table

p	q	p .AND. q	p .OR. q	.NOT. p
1	1	1	1	0
1	0	0	1	0
0	1	0	1	1
0	0	0	0	1

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

.NOT.

.AND.

.OR.

Example:

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

Output:

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

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

.NOT. p

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

p .AND..NOT. q

p .OR..NOT. q

p .AND.(.NOT. q )

p .OR.(.NOT. q )

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

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

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

.NOT.L

.NOT.(X .GT. Y)

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

(L) .AND. M

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

.AND. P

.AND. must be preceded by a logical expression

K .EQ. 1 .OR. 2

.OR. must be followed by a logical expression

P .AND. .OR.R

.AND. always must be separated from .OR. by a logical expression

Examples:

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

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

Examples:

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

## MASKING EXPRESSIONS

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

Examples:

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

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

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

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

Examples:

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

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

Examples:

TAX .AND. INT

.NOT. 55

734 .OR. 82

A .AND. 77B

B .OR. C

M .AND. .NOT. 77B

Extract the low order 6 bits of A

Logical sum of the contents of B and C

Clear the low order 6 bits of M.

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

J = 0101011101 and I = 1100110101

J .AND. I

The bit-by-bit logical product is formed

J 0101011101

I 1100110101

0100010101

Result after masking

J .OR. I

The bit-by-bit logical sum is formed

J 0101011101

I 1100110101

1101111101

Result after masking



.NOT. Complement the operand

.NOT. I

I 1100110101

0011001010                      Result after masking

.NOT. may appear with .AND. and .OR. only as follows:

masking expression .AND. .NOT. masking expression

masking expression .OR. .NOT. masking expression

masking expression .AND. (.NOT. masking expression)

masking expression .OR. (.NOT. masking expression)

If an expression contains masking operators of equal precedence, the expression is evaluated from left to right.

A .AND. B .AND. C

A .AND. B is evaluated before B .AND. C

Using the following values:

A	77770000000000000000	octal constant
D	000000007777777777	octal constant
B	00000000000000001763	octal form of integer constant
C	20045000000000000000	octal form of real constant

Masking operations produce the following octal results:

.NOT. A	is	00007777777777777777
A .AND. C	is	20040000000000000000
A .AND. .NOT. C	is	57730000000000000000
B .OR. .NOT. D	is	77777770000000001763

Invalid example:

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

Example:

```
PROGRAM MASK (INPUT,OUTPUT)
1  FORMAT (1H1,5X,4HNAME,///)
   PRINT 1
2  FORMAT (3A10,I1)
3  READ 2,LNAME,FNAME,ISTATE,KSTOP
   IF(KSTOP.EQ.1)STOP

C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD
   IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.77770000000000
11  K00000B)) GO TO 3
   FORMAT(5X,2A10)
10  PRINT 11,LNAME,FNAME
   GO TO 3
   END
```

---

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

$v = \text{expression}$

$v$  is a variable or an array element

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

## ARITHMETIC ASSIGNMENT STATEMENTS

$v = \text{arithmetic expression}$
------------------------------------

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

Examples:

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

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

Complex

Double Precision

Real

Integer

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

### CONVERSION TO INTEGER

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

## CONVERSION TO REAL

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

## CONVERSION TO DOUBLE PRECISION

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

## CONVERSION TO DOUBLE PRECISION (CONTINUED)

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

## CONVERSION TO COMPLEX

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

## LOGICAL ASSIGNMENT

Logical variable or array element = Logical or relational expression

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

Examples:

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

LOG2 is assigned the value .FALSE. because I $\neq$ 0

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

NSUM is assigned the value .FALSE.

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

## MASKING ASSIGNMENT

v = masking expression

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

Examples:

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

```

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

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

```

## MULTIPLE ASSIGNMENT

$$v_1 = v_2 = \dots v_n = \text{expression}$$

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

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

```
Y = Z
```

```
X = Y
```

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

Examples:

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

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

Multiple assignment is legal in all types of assignment statements.



---

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

ASSIGN	PAUSE
GO TO	STOP
IF	END
DO	RETURN
CONTINUE	

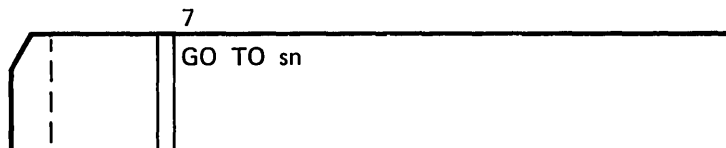
Control must be transferred to an executable statement only.

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

## GO TO STATEMENT

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

### UNCONDITIONAL GO TO STATEMENT



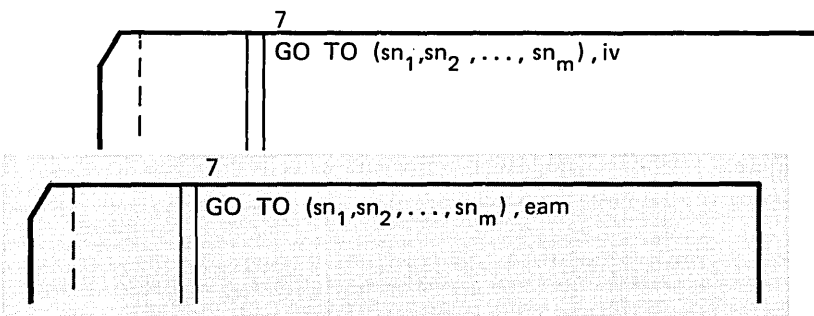
sn is a label of an executable statement.

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

Example:

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

## COMPUTED GO TO STATEMENT



sn<sub>i</sub> is a label on an executable statement.

iv is an integer variable.

eam is an arithmetic or masking expression.

The computed GO TO statement transfers control to one of the statements referenced in the parentheses. If the variable iv has a value of one, control transfers to the statement labeled sn<sub>1</sub>; if the value is i, control transfers to the statement labeled sn<sub>i</sub>.

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

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

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

Example 1:

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

The next statement executed is:

10 if L = 1

20 if L = 2

30 if L = 3

20 if L = 4

Example 2:

K=2

GO TO(100,150,300),K

Statement 150 is executed next.

K=2

X=4.6

.

.

.

GO TO(10,110,11,12,13),X/K Control transfers to statement 110, since the integer value of the expression X/K equals 2.

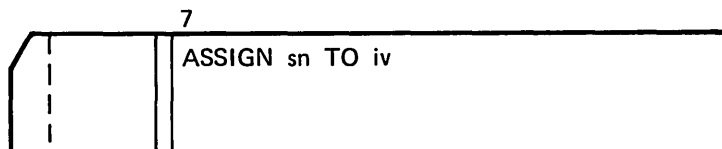
Example 3:

M=4

GO TO (100,200,300),M

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

## ASSIGN STATEMENT



sn is a label of an executable statement.

iv is an integer variable.

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

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

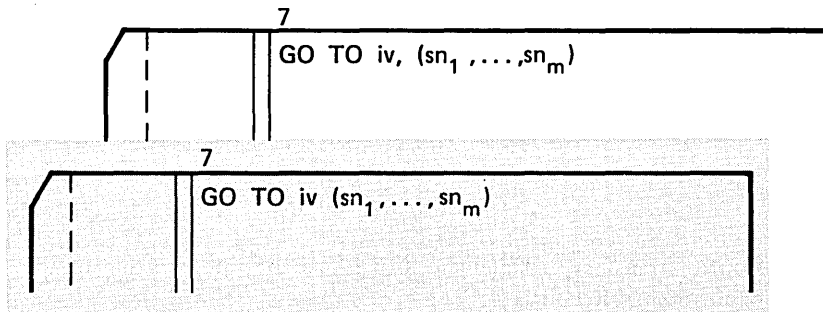
Example:

```

ASSIGN 10 TO LSWITCH
GO TO LSWITCH(5,10,15,20)           Control transfers to statement 10

```

## ASSIGNED GO TO STATEMENT



iv is an integer variable.

(sn<sub>1</sub>, ..., sn<sub>m</sub>) is a list of all the statement labels to which control can be passed by this assigned GO TO. Upon execution of the assigned GO TO, iv must be assigned to one of the labels in the list.

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

Example:

```

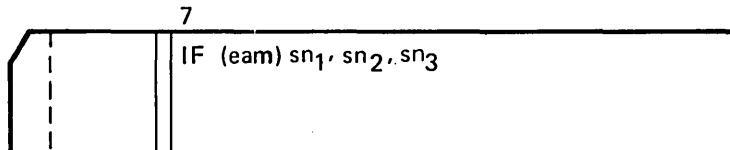
ASSIGN 50 TO JUMP
10 GO TO JUMP,(20,30,40,50)   Statement 50 is executed immediately after statement 10.
.
20 CONTINUE
.
30 CAT=ZERO+HAT
.
.
40 CAT=10.1-3.
.
.
50 CAT=25.2+7.3

```

## ARITHMETIC IF STATEMENT

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

### THREE-BRANCH ARITHMETIC IF STATEMENT



eam is an arithmetic or masking expression.

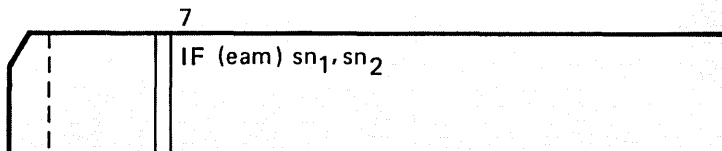
sn<sub>1</sub>, sn<sub>2</sub>, sn<sub>3</sub> are labels on executable statements.

The three-branch IF statement transfers control to the statement labeled sn<sub>1</sub> if the value of the expression is less than zero, to the statement labeled sn<sub>2</sub> if it is equal to zero, or to the statement labeled sn<sub>3</sub> if it is greater than zero.

Example:

```
PROGRAM IF (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
  READ (5,100) I,J,K,N
100 FORMAT (10X,4I4)
  IF(I-N) 3,4,6
  3 ISUM=J+K
  6 CALL ERROR1
  PRINT 2, ISUM
  2 FORMAT (I10)
  4 STOP
  END
```

### TWO-BRANCH ARITHMETIC IF STATEMENT



eam is an arithmetic or masking expression.

sn<sub>1</sub>, sn<sub>2</sub> are labels on executable statements.

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

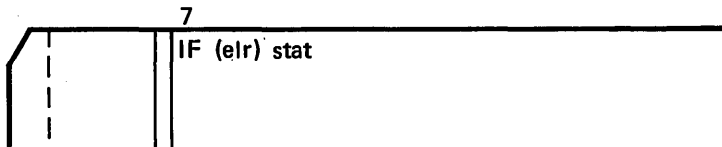
Example:

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

## LOGICAL IF STATEMENT

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

### STANDARD-FORM LOGICAL IF STATEMENT



elr is a logical or relational expression.

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

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

Examples:

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

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

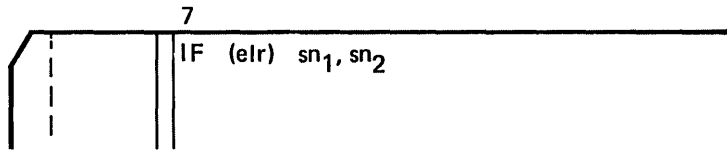
```
      IF (A.LE. 2.5) CASH=150.
70 B=A+C-TEMP
```

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

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

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

## TWO-BRANCH LOGICAL IF STATEMENT



elr is a logical or relational expression.

sn<sub>1</sub>, sn<sub>2</sub> are labels on executable statements.

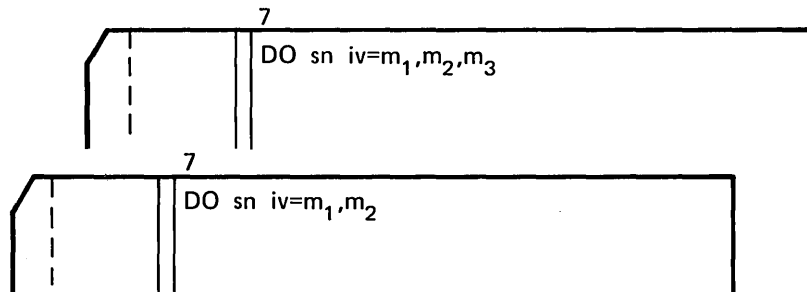
The two-branch logical IF allows for transfer of control to one of two executable statements. If the value of the logical or relational expression is true, control is transferred to the statement labeled sn<sub>1</sub>. If the value of the expression is false, control is transferred to the statement labeled sn<sub>2</sub>.

Example:

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

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

## DO STATEMENT



sn Terminal statement label; an executable statement that must physically follow and reside in the same program unit as its associated DO statement. The terminal statement must not be any arithmetic or two-branch logical IF, a GO TO, RETURN, END, STOP, PAUSE, or another DO statement.

iv Control variable; an integer variable.

m <sub>1</sub>	Initial parameter.	}	Indexing parameters: unsigned integer or octal constants or integer variables with positive non-zero values at execution such that neither $m_1+m_3$ nor $m_2+m_3$ is larger than $2^{17}-1$ . If the indexing parameters exceed these constraints, the results are unpredictable. If $m_3$ is not specified, its value is assumed to be 1.
m <sub>2</sub>	Terminal parameter.		
m <sub>3</sub>	Incrementation parameter.		

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

## DO LOOPS

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

1. iv is assigned the value of  $m_1$ .
2. The range of the DO loop is executed.
3. iv is incremented by the value of  $m_3$ .
4. iv is compared with  $m_2$ . If the value of iv is less than or equal to the value of  $m_2$ , the sequence of operations starting at step 2 is repeated. If the value of iv is greater than the value of  $m_2$ , the DO is said to have been satisfied, the control variable becomes undefined, and control passes to the statement following sn. If  $m_1$  is greater than or equal to  $m_2$ , the range of the DO loop is executed once.

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

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

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

Example 1:

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

The statements following DO up to and including statement 10 are executed four times. The DO loop is executed with I equal to 1, 4, 7, 10. Statement 300 is then executed.



Example 2:

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

Statement 10 is executed five times, whether or not Z = EQUATE is executed. Statement 6 is executed only after the DO loop is satisfied.

Example 3:

```
IVAR = 9
.
.
DO 20 I = 1,200
IF (I-IVAR) 20,10,10
20 CONTINUE
10 IN = I
```

An exit from the range of the DO is made to statement 10 when the value of the control variable I is equal to IVAR. The value of the integer variable IN becomes 9.

Example 4:

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

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

## NESTED DO LOOPS

When a DO loop entirely contains another DO loop, the grouping is called a DO nest. DO loops can be nested to 50 levels. The range of a DO statement can include other DO statements providing the range of each inner DO is entirely within the range of the containing DO statement.

The last statement of an inner DO loop must be either the same as the last statement of the outer DO loop or must occur before it. If more than one DO loop has the same terminal statement, a transfer to that statement can be made only from within the range (or extended range) of the innermost DO, and the label cannot be referenced in any GO TO or IF statement in the nest except in the range of the innermost DO.

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

A transfer from the range of an outer DO into the range of an inner DO loop is not allowed; however, a transfer out of the range of an inner DO into the range of an outer DO is allowed because such a transfer is within the range of the outer DO loop.



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



Example 1:

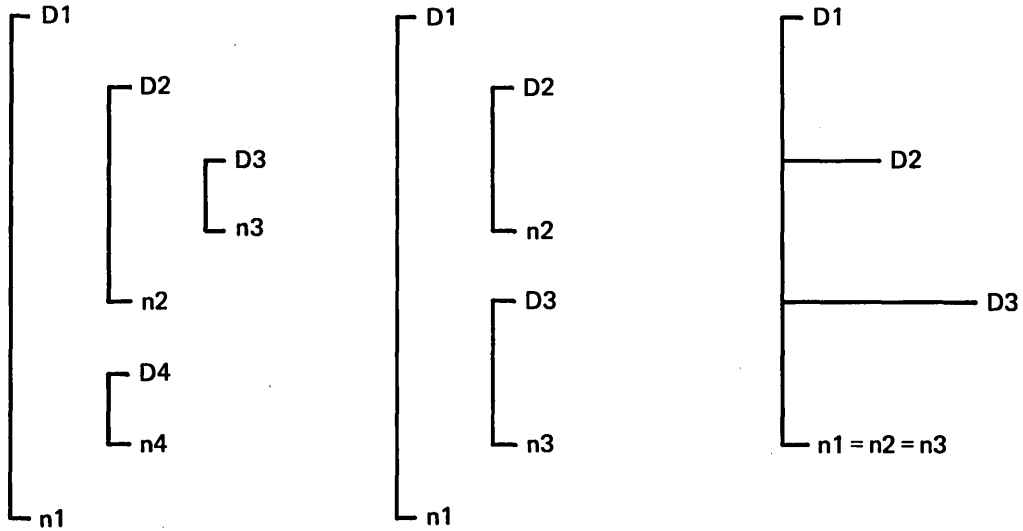
```

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

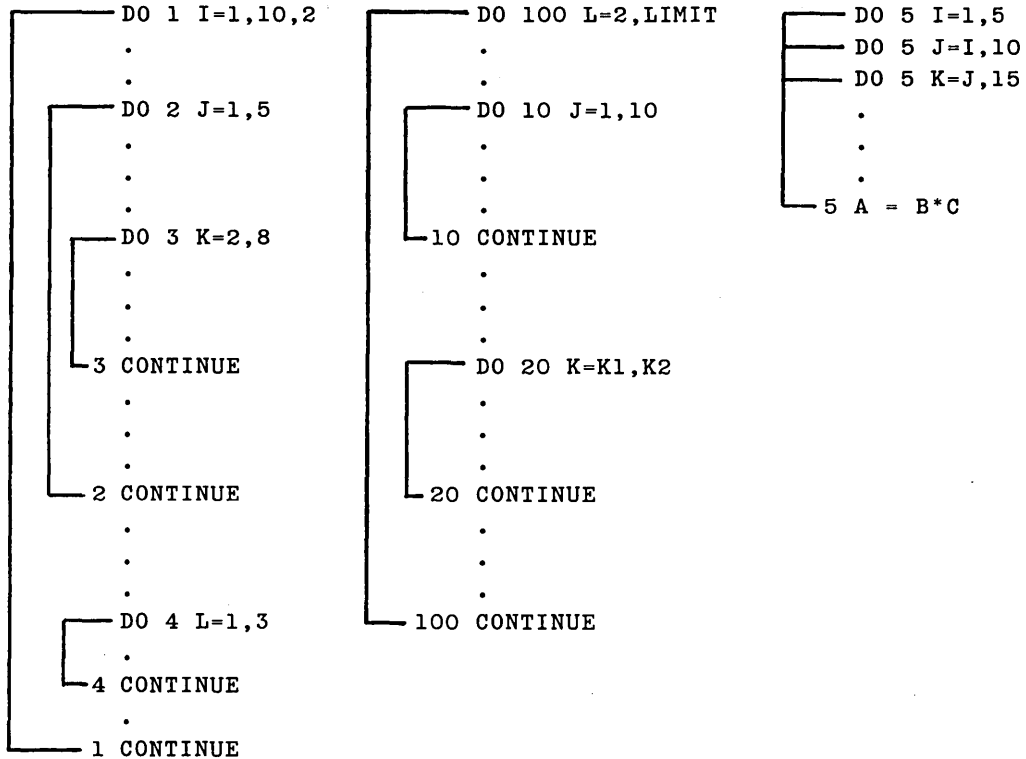
```

This example sets arrays A and B to zero.

Example 2:



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



Example 3:

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

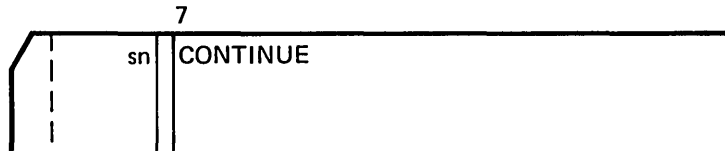
Since statement 10 is the terminal statement for more than one DO loop, it can be referenced in a GO TO or IF statement in the range of the innermost DO. If 10 is referenced in one of the outer loops, control is transferred out of the range with undefined results.

Example 4:

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

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

## CONTINUE STATEMENT



sn is a statement label.

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

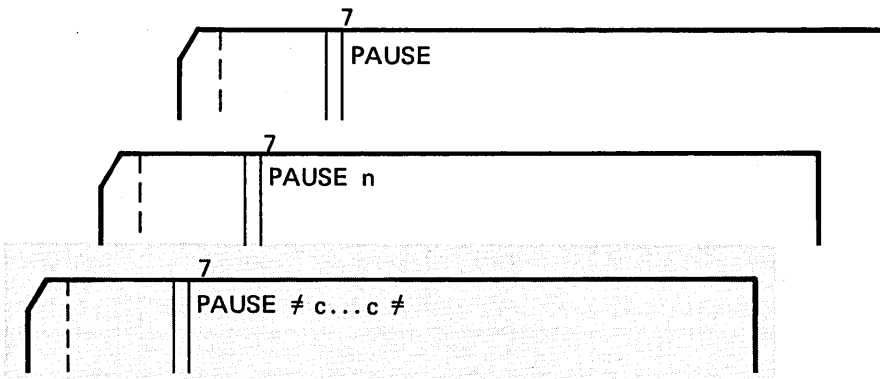
Example 1:

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

Example 2:

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

## PAUSE STATEMENT

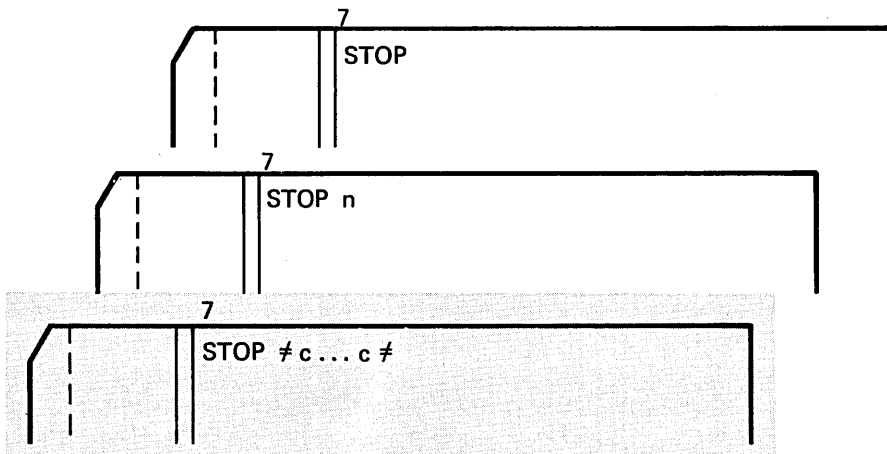


n is a string of 1-5 octal digits.

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

When a PAUSE statement is encountered during execution, the program halts and PAUSE n, or c...c, appears as a dayfile message on the display console. The operator can continue or terminate the program with an entry from the console. If the program is not terminated, it continues with the next statement. If n is omitted, blanks are implied.

## STOP STATEMENT



n is a string of 1-5 octal digits.

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

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

## END STATEMENT



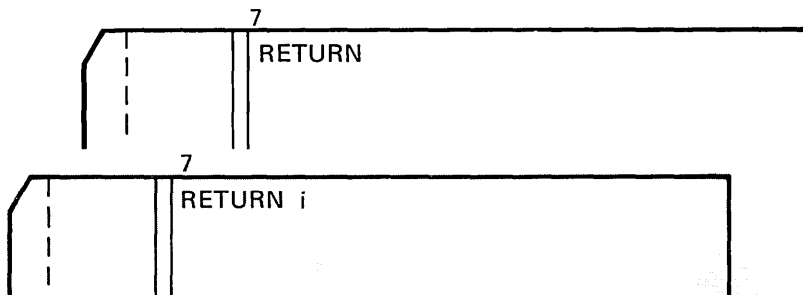
The END statement indicates the end of the program unit to the compiler. Every program unit must physically terminate with an END statement. The END statement can follow a \$ statement separator, be labeled, and be continued. If control flows into or branches to an END statement, it is treated as if a RETURN statement had preceded the END statement.

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

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

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

## RETURN STATEMENT



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

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

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

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

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

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

Example 1:

```

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

```

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


Example 2:

Calling Program	Subprogram
.	.
.	.
.	.
CALL PGM1(A,B,C),	SUBROUTINE PGM1(X,Y,Z),
XRETURNS (5,10)	XRETURNS (M,N)
.	U=X**Y
.	X=Z+X*Y
5 B=SQRT(A*C)	20 IF (U+X) 25, 30, 35
.	25 RETURN M      Return is to statement 5 in calling program.
.	30 RETURN N      Return is to statement 10 in calling program.
10 CALL PGM2 (D,E)	35 Z=Z+(X*Y)
.	RETURN      Return is to statement following CALL PGM1.
.	END
.	
.	
.	

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



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

<p>IMPLICIT</p> <p>Type</p> <p>DIMENSION</p> <p>COMMON</p> <p>EQUIVALENCE</p> <p>EXTERNAL</p> <p>LEVEL</p> <p>DATA</p>		<p>The IMPLICIT statement must precede other specification statements.</p> <p>If any of these statements appears after the first executable statement or statement function definition, the specification statement is ignored and a fatal diagnostic is printed.</p> <p>The DATA statement must follow all other specification statements except statement function definitions and FORMAT statements (see section III-9).</p>
--	---	---

## TYPE STATEMENTS

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

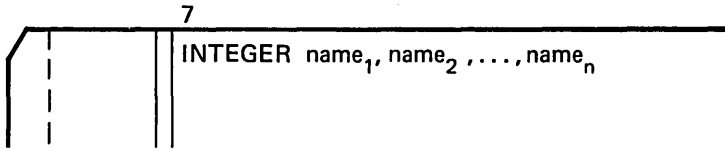
In the absence of an explicit type statement, the type of a symbolic name is implied by the first character of the name: I, J, K, L, M, or N imply type integer and any other letter implies type real, unless an IMPLICIT statement is used to change this normal implied type.

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

## EXPLICIT TYPE DECLARATIONS

Five explicit type statements can be declared: INTEGER, REAL, COMPLEX, DOUBLE PRECISION, and LOGICAL.

### INTEGER



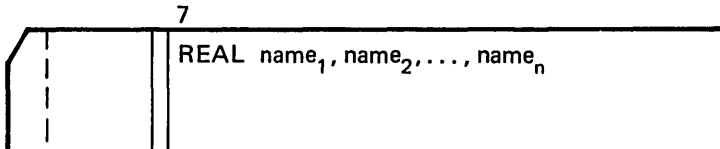
The symbolic names listed are declared as type integer.

Example:

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

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

### REAL



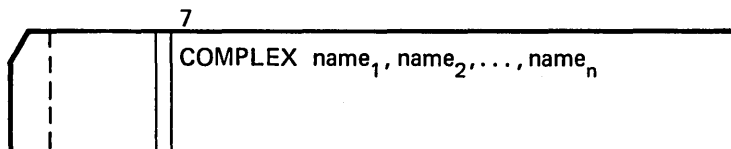
The symbolic names listed are declared as type real.

Example:

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

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

### COMPLEX



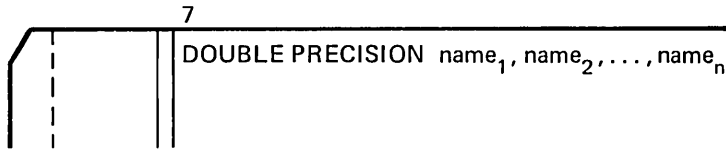
The symbolic names listed are declared as type complex.

Example:

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

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

## DOUBLE PRECISION



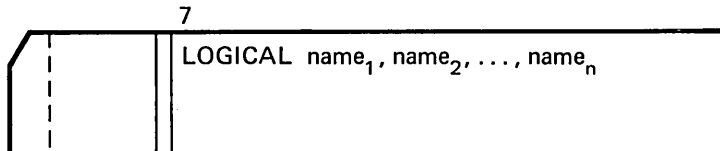
The symbolic names listed are declared as type double precision. DOUBLE can be used instead of DOUBLE PRECISION.

Example:

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

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

## LOGICAL



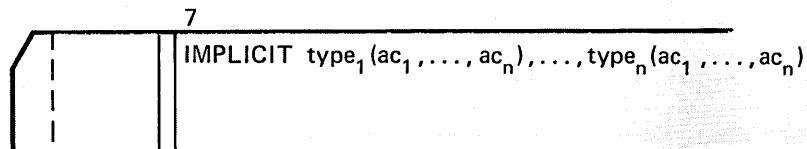
The symbolic names listed are declared as type logical.

Example:

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

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

## IMPLICIT TYPE STATEMENT



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

ac Single alphabetic character, or range of characters represented by the first and last character separated by a minus sign. ac must be enclosed in parentheses.

This statement specifies the type of variables or array elements beginning with the letters ac. Only one IMPLICIT statement may appear in a program unit, and it must precede other specification statements. An IMPLICIT statement in a FUNCTION or SUBROUTINE subprogram affects the type associated with dummy arguments and the function name, as well as other variables in the subprogram. An IMPLICIT statement cannot be used to dimension an array.

Explicit typing of a variable name or array element in a type statement or FUNCTION statement overrides an IMPLICIT specification.

Example 1:

```
IMPLICIT INTEGER(A-D,R)
REAL ASUM
ASUM = BOR + ROR * ANEXT
DECK = CROWN + B
```

The variables BOR, ROR, ANEXT, DECK, CROWN and B are of type integer; ASUM is type real.

Example 2:

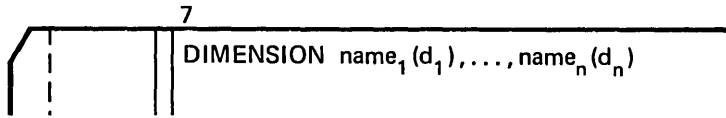
```
PROGRAM COME (OUTPUT,TAPE6=OUTPUT)
IMPLICIT INTEGER (A-F,H)
DIMENSION E(3,4)
COMMON A(1),B,C,D, F,G,H
EQUIVALENCE (A,E,I)
NAMELIST/VLIST/A,B,C,D,E,F,G,H,I

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

WRITE (6,VLIST)
STOP
END
```

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

## DIMENSION STATEMENT



$d_i$             Array declarator, 1-3 integer constants. In a subprogram DIMENSION statement, they can be integer variables.

$name_i$         Symbolic name of an array.

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

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

Example:

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

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

Example:

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

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

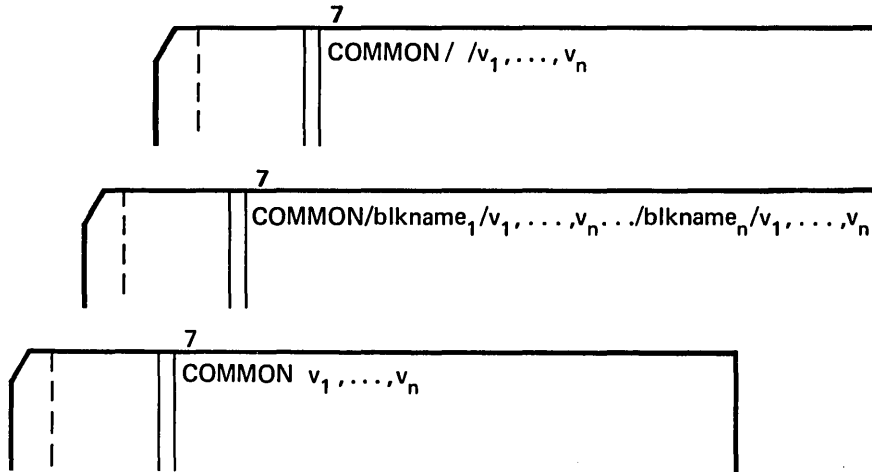
```
REAL NIL (6,2,2)
```

Example:

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

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

## I COMMON STATEMENT



blkname	Block name or number enclosed in slashes. A block name is a symbolic name. A block number is 1-7 digits; it must not contain any alphabetic characters. Leading zeros are ignored. 0 is a valid block number. The same block name or number can appear more than once in a COMMON statement or a program unit; the loader links all variables in blocks having the same name or number into a single labeled common block.
$v_1, \dots, v_n$	Variables or array names which can be followed by constant subscripts that declare the dimensions. The variable or array names are assigned to blkname. The COMMON statement can contain one or more block specifications.
//	Denotes a blank common block. If blank common is the first block in the statement, slashes can be omitted.

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

COMMON is a non-executable statement. See section III-9 for proper location of COMMON statements relative to other statements in the program unit. The COMMON specification provides up to 125 storage blocks that can be referenced by more than one subprogram. A block of common storage can be labeled by a name or a number. A COMMON statement without a name or number refers to a blank common block. Variables and array elements can appear in both COMMON and EQUIVALENCE statements. A common block of storage can be extended by an EQUIVALENCE statement; however, no common block can exceed 131,071 words.

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

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

The length of a common block, other than blank common, must not be increased by a subprogram using the block unless the subprogram is loaded first by the operating system loader.

Example:

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

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

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

The COMMON statement may contain one or more block specifications:

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

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

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

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

These statements have the same effect as the single statement:

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

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

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

Examples:

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

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

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

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

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

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

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

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

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

If a program unit does not use all locations reserved in a common block, unused variables can be inserted in the COMMON declaration in the subprogram to ensure proper correspondence of common areas.



Example:

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

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

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

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

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

```
COMMON/SUM/D subprogram
```

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

Example:

```
PROGRAM MAIN  
COMPLEX C  
COMMON/TEST/C(20)/36/A,B,Z
```

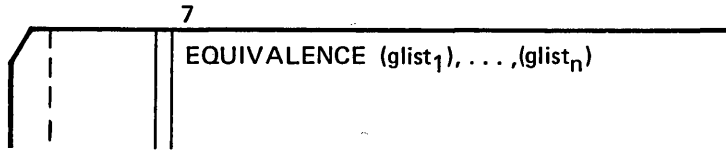
The block named TEST consists of 40 computer words. The length of the block numbered 36 is three computer words.

The subprogram may use different names as in:

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

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

## EQUIVALENCE STATEMENT



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

Array elements must have integer constant subscripts. Dummy arguments must not appear in an equivalence statement. Equivalenced variables must be assigned to the same level of storage.

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

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

Example:

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

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

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

Example:

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

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

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

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

Example:

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

These statements establish the following equivalence groups:

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

and the following equivalence classes:

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

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

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

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

means the same as the statements:

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

A logical, integer, or real entity equivalenced to a double precision or complex entity shares the same location as the real or most significant part of the complex or double precision entity.

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

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

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

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

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

Example:

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

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

Two or more arrays can share the same storage locations.

Example:

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

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

Lengths of arrays need not be equal.

Examples:

```
DIMENSION ZER01(10,5),ZER02(3,3)
EQUIVALENCE (ZER01,ZER02)           is a legal EQUIVALENCE statement

EQUIVALENCE (ITEM,TEMP)
```

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

Example:

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

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

WRITE (6,VLIST)
STOP
END
```

Output from Program COME:

```
$VLIST
A      = 1,
B      = 2,
C      = 3,
D      = 4,
E      = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
F      = 5,
G      = 0.0,
H      = 7,
I      = 1,
$END
```

An explanation of this example appears in part 2.

## EQUIVALENCE AND COMMON

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

Example:

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

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

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

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

Example:

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

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

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

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

Examples:

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

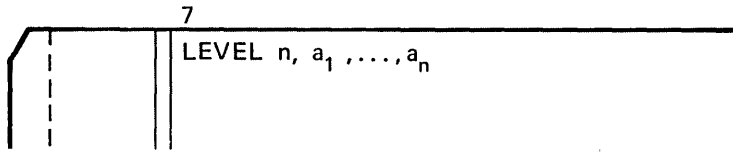
```
COMMON /HAT/ A(4),C /X/ Y,Z
EQUIVALENCE (C,Y)          illegal
```

As stated in section I-2, indexing outside of array bounds is prohibited. Since the compiler attempts to minimize the size of equivalence classes in common blocks to the smallest subset of the block that includes all members named in associated EQUIVALENCE statements, all members of a common block will not necessarily be considered as one array. The programming practice of intentionally referencing locations outside a known array may produce unintentional results as shown in the following example.

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

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

## LEVEL STATEMENT



- $a_1, \dots, a_n$  List of variables or array names separated by commas
- $n$  Unsigned integer 1, 2, or 3 indicating level to which list is to be allocated.
- § { 1 Small core memory resident (SCM)
- 2 Large core memory resident (LCM). Directly addressable (or word addressable)
- 3 Large core memory resident, accessed by block transfer to or from small core memory through MOVLEV subroutine call
- ‡ { 1 Central memory resident
- 2 Central memory resident
- 3 Extended core storage resident, accessed by block transfer to or from central memory through MOVLEV subroutine call

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

No dimension or type information may be included in the LEVEL statement.

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

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73 and 74, CYBER 170, and 6000 Series computers.

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

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

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

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

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

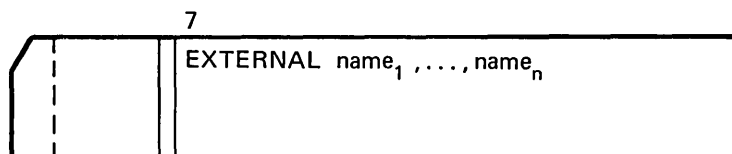
If a variable or array name appears in an EQUIVALENCE and a LEVEL statement, the equivalenced variables must all be allocated to the same level.

Example:

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

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

## EXTERNAL STATEMENT



name<sub>1</sub>,...,name<sub>n</sub>

Subprogram names

Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program.



Any name used as an actual argument in a call is assumed to be a variable or array unless it appears in an EXTERNAL statement. An EXTERNAL statement must be used even if the subprogram concerned is a standard system function, such as SQRT. However, an EXTERNAL statement is not required for intrinsic functions used as actual arguments. If an intrinsic function name appears in an EXTERNAL statement, the user must supply the function.

Example:

Calling Program	Subprogram
EXTERNAL SIN, SQRT	SUBROUTINE SUBRT (A,B,C)
CALL SUBRT(2.0,SIN,RESULT)	X=A+3.14159/2.
WRITE (6,100) RESULT	C=B(X)
100 FORMAT (F7.3)	RETURN
CALL SUBRT(2.0,SQRT,RESULT)	END
WRITE (6,100)RESULT	
STOP	
END	

First the sine, then the square root are computed; and in each case, the value is returned in RESULT. The EXTERNAL statement must precede the first executable statement, and always appears in the calling program. (It may not be used with statement functions.)

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

Example:

Calling Program	Subprogram
CALL SUBRT(SIN(X),RESULT)	SUBROUTINE SUBRT(A,B)
	.
	.
	.
	B=A
	.
	.
	.
	END

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

Example:

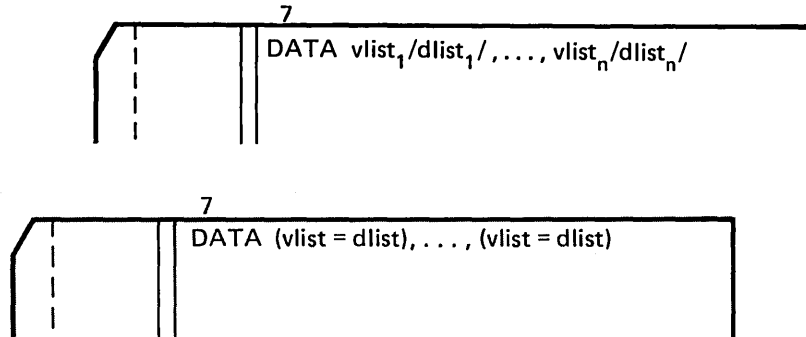
```
PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
COMMON X(4,3)
REAL Y(6)
EXTERNAL MULT, AVG
NAMELIST/V/X,Y,AA,AM
CALL SET(Y,6,0.)
CALL IOTA(X,12)
CALL INC(X,12,-5.)
AA=PVAL(12,AVG)
AM=PVAL(12,MULT)
WRITE(6,V)
STOP
END
```

```
FUNCTION AVG(J)
C  AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
COMMON A(100)
AVG=0.
DO 1 I = 1,J
1  AVG=AVG+A(I)
AVG=AVG/FLOAT(J)
RETURN
END
```

```
REAL FUNCTION MULT(J)
COMMON ARRAY(12)
MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
RETURN
E N D
```

An explanation of this example appears in part 2.

## DATA STATEMENT



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

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

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

constant list

List of constants separated by commas.

rf

Integer constant. The constant or constant list is repeated the number of times indicated by rf.

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

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

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

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

Example:

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

The following octal constants are stored in ARRAY B:

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

When a Hollerith specification is used in a DATA statement, it should not exceed 10 characters. For example, to store the following values in an array A:

Location	Contents
A(1)	1234567890
A(2)	ABCDEFGHIJ
A(3)	KLMNOPQRST
A(4)	UVWXYZ + - *

the following statements should be used:

```
DIMENSION A(4)
DATA A/10H1234567890,10HABCDEFGHIJ,10HKLMNOPQRST,10HUVWXYZ+- */
```

The following statements would not product the desired result:

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

They would initialize:

Location	Contents
A(1)	1234567890
A(2)	KLMNOPQRST
A(3)	UVWXYZ + - *
A(4)	undefined

## IMPLIED DO IN DATA LIST

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

(varlist, i=m<sub>1</sub>, m<sub>2</sub>, m<sub>3</sub>)

where:

varlist            an array element or another implied DO. If it is an array element, its subscript expressions must be of the form

$$M*i\pm N$$

where M and N are unsigned integer constants.

i                    a simple integer variable called the index variable

m<sub>1</sub>, m<sub>2</sub>, m<sub>3</sub>        unsigned integer constants specifying the initial value, terminal value, and increment, respectively, for the index variable; if m<sub>3</sub> and the preceding comma are omitted, the value of m<sub>3</sub> is assumed to be 1.

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

When an implied DO is encountered in a DATA statement, the elements in its range are initialized for index variable i with the value m<sub>1</sub>. The index variable is then increased by m<sub>3</sub> and, if i is less than or equal to m<sub>2</sub>, the range of varlist is initialized for the new value of i. This procedure continues until the value of the index variable exceeds m<sub>2</sub>.

Example 1:

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

The values stored in array ANARRAY are:

ANARRAY(1)	1.
	2.
	3.
	2.5
	2.5
	2.5
	2.5
	2.5
	2.5
	2.5
ANARRAY(10)	2.5

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

Example 2:

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

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

Example 3:

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

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

Example 4:

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

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

ARRAY AMASS:

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

ARRAY A

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

ARRAY B:

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

Example 5:

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

Example 6:

2\*(1.0, 2.0) Means repeat the real constants 1.0 and 2.0 twice

2\*((1.0, 2.0)) Means repeat the complex constant (1.0, 2.0) twice

Example 6 illustrates the use of repeat specifications with real and complex constants. When a repeat specification is used with complex constants, it is necessary to ensure that the parentheses that are part of the complex constant are not confused with the parentheses enclosing the constant list.

Example 7:

```

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

```

\$CUT

```

I      = 5,
L      = T,
X      = .31415926536E+01,
Z1     = (.21E+01,-.3E+01),
A      = .1E+01, .2E+01, .1E+01, .2E+01,
Z      = (.1E+01,-.15E+01), (.1E+01,-.15E+01), (.1E+01,-.15E+01),
$END

```

Example 8:

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

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

```
INTEGER ARAY(5)
```

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

```
COMMON/BOX/ARAY4(3,4,5)
```

```
DATA (ARAY4(1,3,5)=22.5)
```

The statements:

```
DIMENSION D3(4),POQ(5,5)
```

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

**Initialize:**

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

and set the entire array POQ to zero.

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



# PROGRAMS, SUBPROGRAMS, AND PROCEDURES I-7

---

A program unit consists of FORTRAN statements, with optional comments, terminated by an END statement. A main program is a program unit that does not begin with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. A subprogram is a program unit that begins with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. An executable program contains one main program with or without subprograms. A program unit containing no FORTRAN statements other than an END statement is considered a null program; it is diagnosed and ignored.

A subprogram is defined separately and can be compiled independently of a main program. If the subprogram begins with a SUBROUTINE or FUNCTION statement, it is a procedure subprogram and can exchange no, one, or more values through a list of arguments, through common, or both. If the subprogram begins with a BLOCK DATA statement, it is a specification subprogram.

A procedure is a procedure subprogram, statement function, intrinsic function, or basic external function. Intrinsic functions and basic external functions are FORTRAN supplied procedures and are available to any programmer. Statement functions and procedure subprograms are supplied by the programmer.

The differences between function and subroutine specification and use are summarized in table 7-1.

Table 7-1. Differences Between a Function and Subroutine Subprogram

	Function	Subroutine
How Used	The name appearing in an expression is used as the reference.	A CALL statement is used as the reference.
Arguments	One or more arguments must be included.	Arguments need not be present.
How Typed	Name is typed implicitly by first letter or explicitly by the type designation appearing before the word FUNCTION.	No type is associated with the name.

Functions return a single value through the function name. Function subprograms defined by the programmer also can return values through a list of arguments, through common, or both.

Table 7-2 summarizes the terminology of the overlapping categories of procedures and subprograms.

Table 7-2. Procedure and Subprogram Interrelationships

	Statement Function	Intrinsic Function	Basic External Function	Function Subprogram	Subroutine Subprogram	Block Data Subprogram
Procedure	yes	yes	yes	yes	yes	no
External procedure	no	no	yes	yes	yes	N/A
Subprogram	no	no	no	yes	yes	yes
Function	yes	yes	yes	yes	no	no
External function	no	no	yes	yes	N/A	N/A
Who defines	user	compiler	compiler	user	user	user
Where defined	within program unit	compiler	library	external to program unit	external to program unit	external to program unit

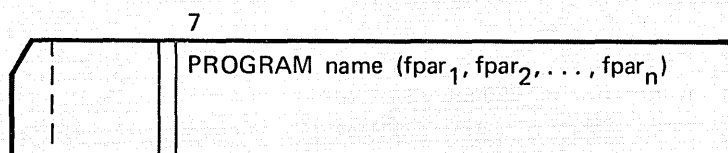
N/A = not applicable

Programmer written procedures (statement functions, function subprograms, and subroutine subprograms) are discussed below as a group. FORTRAN supplied procedures (intrinsic functions and basic external functions) are discussed in detail in section I-8. The only subprogram that is not a procedure is the block data subprogram. Since it is not executable, it is discussed separately.

## MAIN PROGRAMS

A main program can contain any FORTRAN statements except FUNCTION, SUBROUTINE, or BLOCK DATA; it should have a PROGRAM statement, at least one executable statement, and an END statement. One main program is required in any executable FORTRAN program unit.

### PROGRAM STATEMENT FORMAT



**name** Must be a unique symbolic name within the main program and cannot be used as a subprogram name.

**fpar<sub>i</sub>** The fpar can be any of the following forms:

file	File name (1-6 letters or digits beginning with a letter) for each I/O file required by the main program or its subprograms; the maximum number of file names is 50.
file=n	n <sup>†</sup> is a decimal number specifying the buffer length; default length is 2002 octal words.
file=/r	r is the maximum length in characters for list directed, formatted, and NAMELIST records; default length is 150 characters.
file=n/r	n/r defines both buffer and record lengths.
file <sub>a</sub> =file <sub>b</sub>	File <sub>a</sub> is made equivalent to previously defined file <sub>b</sub> .

In a program structured for overlays, the  $fpar_i$  parameter list is used only in the PROGRAM statement for the main overlay. It is not used in primary and secondary overlay PROGRAM statements.

## PROGRAM STATEMENT USAGE

The PROGRAM statement defines the program name that is used as the entry point name and the object deck name for the loader. Optionally, the PROGRAM statement can declare files that are used in the program and any subprograms that are called. If this statement is omitted from the main program, the program is assumed to have the name START, and two files named INPUT and OUTPUT.

All file names used in standard FORTRAN input/output statements must be listed in the PROGRAM statement. File names referenced by direct call to CYBER Record Manager must not be listed in the PROGRAM statement. If a file name is referenced in a standard FORTRAN input/output statement in a main program, but is not specified in the PROGRAM statement, a warning diagnostic is issued at compile time. If a file name is referenced in a standard FORTRAN input/output statement in a subprogram, but is not specified in the PROGRAM statement of the main program, a diagnostic is issued when the file is used at execution time.

File names on the PROGRAM statement must satisfy the following conditions:

- The file name INPUT must be declared if a READ statement without a logical unit number is included in the program.
- The file name OUTPUT must be declared if a PRINT statement without a logical unit number is included in the program.
- The file name PUNCH must be declared if a PUNCH statement without a logical unit number is included in the program.
- The file name TAPE<sub>u</sub> (u is an integer constant 0-99) must be declared if any input/output statement involving unit u appears in the program. At execution time, if u is a variable, there must be a file name TAPE<sub>u</sub> for each value u may assume.

FORTRAN I/O routines add the characters TAPE as a prefix to the logical unit number to form the file name. TAPE3 is the file name assigned to logical unit 3 and TAPE5 is the file name assigned to logical

<sup>†</sup>n is ignored if specified in a program run under SCOPE 2.1.

unit 5, but TAPE5 and TAPE05 do not specify the same file name. If TAPE05 is used, it can be accessed with FORTRAN I/O statements only by using the display code file name in L format (Input/Output in section I-9 contains details).

TAPEu refers to a file located on rotating mass storage unless specified otherwise in the job deck before the program is executed. The file is temporary unless made permanent by the user.

FORTTRAN I/O statements use the buffer areas established by the file name specified in the PROGRAM statement. The buffer length can appear only with the first reference to the file in the PROGRAM statement. A buffer length of zero should be specified for a file referenced by a BUFFER statement. Since buffered records are transmitted directly into and out of central memory, field length of the program is reduced by at least 2000 (octal) words for each file declared with zero buffer length in the PROGRAM statement.

For files not referenced by BUFFER statements, the following values of n are suggested:<sup>†</sup>

For terminals:            n=number of words in the largest record plus one.

For mass storage :        n $\geq$ 64. Large records and sequential reading/writing execute faster with a larger buffer.

For tapes:	<u>Tape Format</u>	<u>Minimum Value of n</u>
	SCOPE, SI, I, X	128 for formatted. 512 for unformatted.
	S	512 for formatted or unformatted.
	L	$\geq$ maximum block length.

Record length, r, should be specified for files referenced in list-directed input/output statements. When file names are made equivalent, the buffer length and record size specified apply to both files.

Examples:

```
PROGRAM ORB (INPUT,OUTPUT=1000,TAPE1=INPUT,TAPE2=OUTPUT,TAPE4=1000/2000)
```

All input/output statements that reference TAPE1 reference INPUT instead, and all listable output normally recorded on TAPE2 is transmitted to the file named OUTPUT. TAPE4 has a buffer length of 1000 words with a maximum of 2000 characters per record.

```
PROGRAM JIM(INPUT,TAPE19=INPUT)
```

TAPE19=INPUT must be preceded in the same statement by INPUT (or INPUT=buffer length). TAPE19 becomes the name for the file INPUT.

<sup>†</sup>Does not apply to SCOPE 2.1.

```

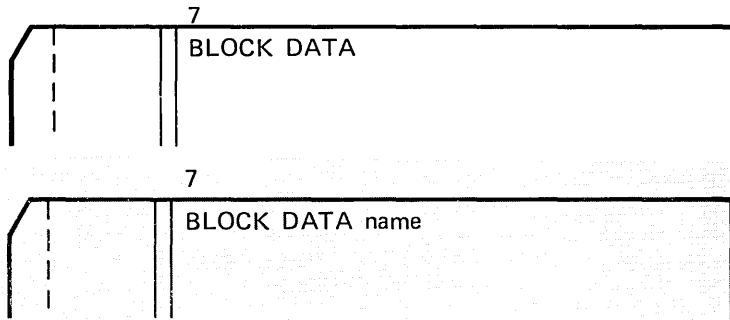
PROGRAM SAMPLE (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
.
.
.
READ(5,100)A,B,C
100 FORMAT (3F7.3)
.
.
.
WRITE(6,200)A,B,C
200 FORMAT (1H1,3F7.3)

```

This statement reads from logical unit 5; it is declared in the PROGRAM statement as TAPE5 which is equivalent to INPUT.

Logical unit 6 is declared as TAPE6 in the PROGRAM statement and equivalent to OUTPUT.

## BLOCK DATA SUBPROGRAM



name identifies the BLOCK DATA subprogram if more than one is compiled.

The block data subprogram is a nonexecutable specification subprogram that can be used to enter data into labeled or numbered common (but not blank common) prior to program execution. The name `BLKDAT` is assigned to the block data subprogram if it is not named by the user.

The block data subprogram contains only `IMPLICIT`, `LEVEL`, type, `DIMENSION`, `COMMON`, `EQUIVALENCE`, `DATA`, and `END` statements. Any executable statements are ignored and a warning is issued. All `DATA` statements must follow the specification statements. Data can be entered into more than one block of common in a block data program.

Example:

```

BLOCK DATA ANAME
COMMON/CAT/X,Y,Z/DEF/R,S,T
COMPLEX X,Y
DATA X,Y/2*((1.0,2.7))/,R/7.6543/
END

```

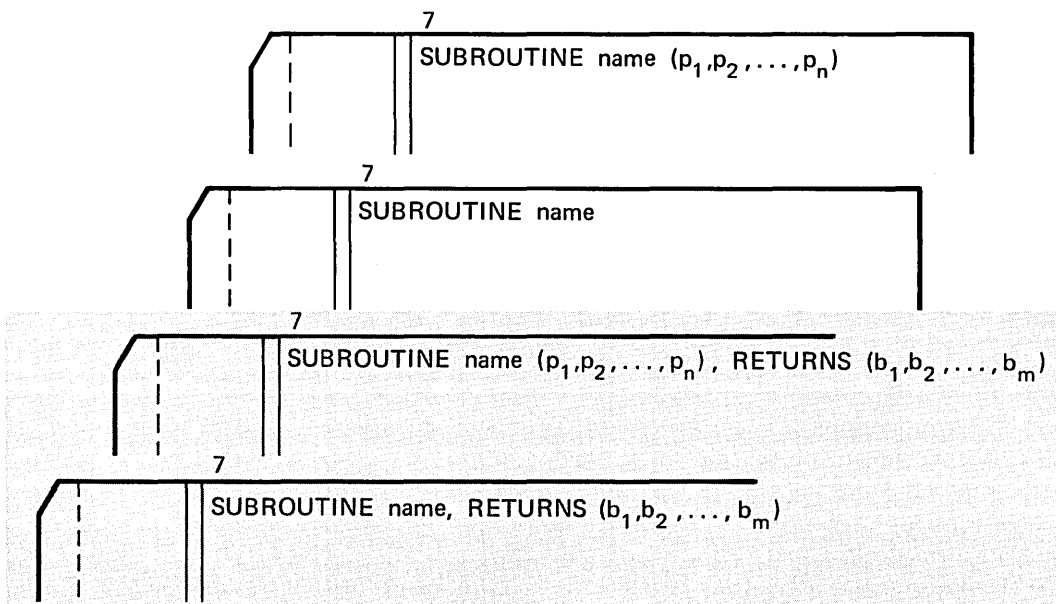
Z is in block CAT and S and T are in DEF, although no initial data values are defined for them.

## PROCEDURES

The category of procedure to be used is determined by its particular capabilities and the needs of the program being written. If the program requires the evaluation of a standard mathematical function, a FORTRAN supplied intrinsic function or a basic external function can be used. If a single computation is needed repeatedly, a user-written statement function is included in the program. If a number of statements are required to obtain a single result, a function subprogram is written. If a number of calculations are required to obtain several values, a subroutine is written.

Each procedure discussion contains a definition, description, and examples. Procedure Communication (later in this section) contains details on how to use procedures and how procedures use arguments or common to communicate.

## SUBROUTINE SUBPROGRAMS



**name** Symbolic name of the subroutine.

**p<sub>1</sub>, ..., p<sub>n</sub>** Dummy arguments that must agree in order, number, type, and LEVEL with the actual arguments passed to the subprogram at execution time.

**b<sub>1</sub>, ..., b<sub>m</sub>** Dummy statement label arguments that must agree in order, number, and LEVEL with the actual statement labels passed to the subroutine at execution time.

The argument lists are optional and limited to a maximum of 63 parameters.

A subroutine subprogram is executed when a CALL statement is encountered in a program unit. A subroutine subprogram must not directly or indirectly call itself. The subroutine subprogram communicates with the calling program unit through a list of arguments passed with the CALL statement or through common. Calling a Subroutine Subprogram later in this section contains more CALL statement details.

The SUBROUTINE statement contains the symbolic name that is used as the main entry point of the subprogram. (The ENTRY statement specifies an alternate entry point in the subprogram.) The subprogram name is not used to return results to the calling program, does not determine the type, and must not appear in any other statement in the same subprogram.

Subroutine subprograms can contain any statements except PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. They begin with a SUBROUTINE statement, should have at least one RETURN statement, and end with an END statement. If control flows into the END statement, then a RETURN is implied. Control is returned to the calling program when a RETURN, RETURN i or END is encountered.

Dummy arguments which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. If an array name without subscripts is used as an actual argument in a CALL statement and the corresponding dummy argument is not declared an array in the subprogram, the first element of the array is used in the subprogram. Adjustable dimensions are permitted in subroutine subprograms (details are given later in this section under Using Arrays).

The RETURNS list allows control to be returned to the calling program somewhere other than at the executable statement immediately following the CALL statement. The CALL statement specifies actual statement labels to replace the dummy statement label arguments in the RETURNS list. The actual statement labels must correspond in order and number with the dummy statement label arguments. The dummy statement label argument i is the statement to which control transfers when RETURN i is executed.

The RETURN statement in section I-5 and the CALL statement in this section give further details.

Example 1:

Calling Program	Subprogram
.	SUBROUTINE ERROR1
.	WRITE (6,1)
.	1 FORMAT (5X,*NUMBER IS OUT OF RANGE*)
IF (A-B) 10,20,20	RETURN
10 CALL ERROR1	END
20 RESULT=(A*CAT) +375.2-ZERO	
.	
.	
.	

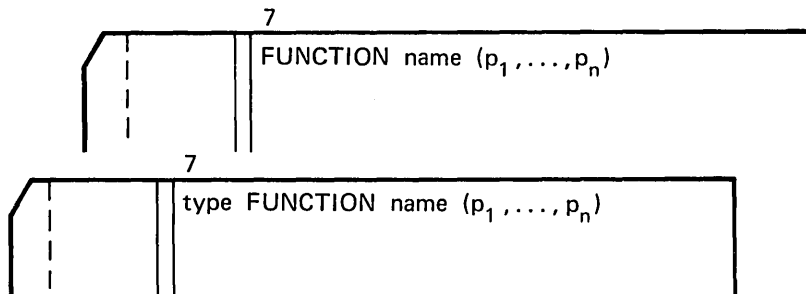
The subroutine ERROR1 is called and executed if A-B is less than zero. Control returns to statement 20. This example also illustrates that arguments need not be used.

**Example 2:**

Calling Program	Subprogram
.	
.	
.	SUBROUTINE PGM1 (X,Y,Z),
CALL PGM1 (A,B,C),	XRETURNS (M,N)
XRETURNS (5,10)	U=X**Y
.	X=Z+X*Y
.	20 IF (U+X) 25, 30, 35
.	25 RETURN M      Return is to statement 5 in calling program
.	30 RETURN N      Return is to statement 10 in calling program
5 B=SQRT(A*C)	35 Z=Z+(X*Y)
.	RETURN      Return is to statement following CALL PGM1
.	END
10 CALL PGM2 (D,E)	
.	
.	
.	

This example illustrates the use of the RETURNS list as well as the use of the normal RETURN statement.

**FUNCTION SUBPROGRAM**



- name      Symbolic name of the subprogram.
- $p_1, \dots, p_n$       Dummy arguments that should agree in order, number, and type with the actual arguments in the calling program. At least one argument is required; a maximum of 63 is allowed.
- type      The type may be REAL, INTEGER, DOUBLE, DOUBLE PRECISION, COMPLEX, or LOGICAL.

A function subprogram performs a set of calculations when its name appears in an arithmetic, logical, or masking expression in a referencing program unit. Execution of the function subprogram must result in a single value being defined for the function name. A function subprogram can modify the value of one or more of its arguments or store data in common.



Dummy arguments which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. If an array name without subscripts is used as an actual argument in the function reference and the corresponding dummy argument has not been declared an array in the subprogram, the first element of the array is used in the subprogram. Adjustable dimensions are permitted in function subprograms (details are given in Using Arrays later in this section).

The FUNCTION statement contains the subprogram symbolic name that is used as the entry point when the function is referenced. (See Referencing a Function later in this section for more details.) The function name must not appear in any nonexecutable statements other than the FUNCTION statement in the subprogram. The type of the function name must be the same in the referencing program and the referenced function subprogram. When type is omitted, the type of the function result is determined by the first character of the function name.

The function subprogram can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, another FUNCTION statement, or any statement that directly or indirectly references the function being defined. The function subprogram begins with a FUNCTION statement, should have at least one RETURN statement, and has an END statement that is treated as a RETURN if executed. Control is returned to the referencing program when either a RETURN or END is encountered. A RETURN i in a function subprogram causes a fatal error at compilation time.

A function subprogram can have the same name as that of an intrinsic or basic external function supplied by FORTRAN. Section I-8 defines the conditions under which programmer supplied routines override the FORTRAN supplied routines.

Example:

Calling Program	Subprogram
.	
.	
.	
DIMENSION ARY (5,5)	FUNCTION DIAG (A,N)
.	DIMENSION A(5,5)
.	DIAG=A(1,1)
.	DO 70 I=1,N
10 RES=DIAG(ARY,5)**2	70 DIAG=DIAG*A(I,I)
.	RETURN
.	END
.	

The statement labeled 10 contains the reference to function DIAG. The statement labeled 70 sets the function name to a value. At the end of the function subprogram execution, RES will have the value of DIAG squared.

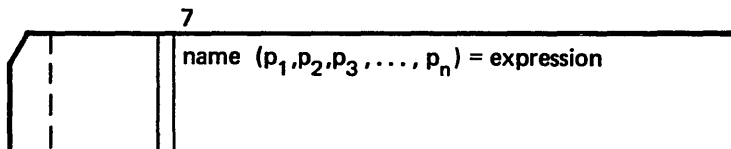
## BASIC EXTERNAL FUNCTION

A basic external function is a predefined procedure included with the system. Section I-8 contains further details.

## INTRINSIC FUNCTION

An intrinsic function is a compiler-defined procedure that is inserted in the referencing program at compile time. Section I-8 contains further details.

## STATEMENT FUNCTION



- name** Type of the function is determined by the type of the function name.
- p<sub>1</sub>, ..., p<sub>n</sub>** Dummy arguments must be simple variable names. At least one argument is required; a maximum of 63 is allowed. These arguments should agree in order, number, type, and LEVEL with the actual arguments used in the function reference.
- expression** Any arithmetic, masking, relational, or logical expression may be used. It may contain references to intrinsic or basic external functions, statement functions, or function subprograms. Names in the expression that do not represent arguments are normal variables having the same value as they have outside the function.

A statement function is a user-defined, single-statement computation and applies only to the program unit containing the definition. Since the statement function only defines the function, the value is computed when the function is referenced and the actual arguments are substituted for the dummy arguments in the definition.

During compilation, the statement function definition is retained by the compiler. Whenever the function is referenced, instructions are generated in-line to evaluate the function (as opposed to FUNCTION subprograms for which an external procedure is used at each reference). The expansion of a statement function is the same as writing the expression in place of the reference. Thus the statement function does not reduce execution speed or efficiency.

Statement function names must not appear in DIMENSION, EQUIVALENCE, COMMON or EXTERNAL statements; they can appear in a type declaration but cannot be dimensioned. Statement function names must not appear as actual or dummy arguments. If the function name is type logical, the expression must be logical. For other types, if the function names and expression differ, conversion is performed as part of the function.

A statement function must precede the first executable statement and it must follow all specification statements. A statement function must not reference itself either directly or indirectly.

Examples:

Statement Function Definitions	Statement Function References
ADD(X,Y,C,D)=X+Y+C+D	RES1=GROSS-ADD(TAX,FICA,INS,RES3)
AVERGE(O,P,Q,R)=(O+P+Q+R)/4	GRADE=AVERGE(TEST1,TEST2,TEST3, TEST4)+MID
LOGICAL A,B,EQV EQV(A,B)=(A.AND.B).OR. (.NOT.A.AND..NOT.B)	TEST=EQV(MAX,MIN).AND.ZED
COMPLEX Z Z(X,Y)=(1.,0.)*EXP(X)*COS(Y) +(0.,1.)*EXP(X)*SIN(Y)	RESULT=(Z(BETZ,GAMMA(I+K))**2-1.) /SQRT(TWOPIE)

Example 1:

The statement function can be used to substitute a FORTRAN supplied function name in a program containing an alternate name for this function.

```
SINF(X)=SIN(X)      Statement function definition.  
.  
.  
.  
A=SINF(3.0+B)+7.    Statement function reference.
```

The above sequence generates exactly the same object code as:

```
A=SIN(3.0+B)+7.
```

Example 2:

To compute one root of the quadratic equation  $ax^2+bx+c=0$ , given values of a, b and c, an arithmetic statement function can be defined as follows:

```
ROOT (A,B,C)=(-B+SQRT(B*B-4.*A*C))/(2.0*A)
```

When the function is used in an expression, actual arguments are substituted for the dummy arguments A, B, C.

```
RESA = ROOT (6.5,7.,1.)
```

is equivalent to writing:

```
RESA = (-7.+SQRT(7.*7.-4.0*6.5*1.0))/(2.0*6.5)
```

Wherever the statement function ROOT (A, B, C) is referenced, the definition of that function — in this case  $(-B+SQRT(B*B-4.*A*C))/(2.*A)$  — is evaluated using the current values of the arguments A, B, C.

## PROCEDURE COMMUNICATION

The procedures defined by a statement function or a procedure subprogram are executed when they are referenced in a program unit.

### PASSING VALUES TO A PROCEDURE

Values can be passed between a calling program unit and a procedure as actual arguments in an argument list or through common. Arrays with adjustable dimensions can be used to pass values of arguments.

Arguments passed to a procedure must agree with the procedure definition in order, number, type, length, and memory residence (See LEVEL in section I-6).

### USING ARGUMENTS

Arguments used for communication between procedures are either actual or dummy (formal). The arguments appearing in a subroutine CALL statement or a function reference are the actual arguments. The corresponding dummy arguments appear in the SUBROUTINE or FUNCTION statement. If a RETURNS list is used, the actual statement label arguments appear in the CALL statement and the dummy statement label arguments appear in the SUBROUTINE and RETURN statements.

The actual arguments (such as constants, arithmetic expressions, logical expressions, variables, and array names) allowed for a particular procedure are given in the discussion of the procedure reference.

Dummy arguments are used as variable, array or external procedure subprogram names within the subprogram and can be used to return values to the calling program. The dummy arguments are replaced by the actual arguments when the procedure is executed. Since all names are local to the program unit containing them, the same dummy argument name can be used in more than one program unit. A dummy argument must not appear in COMMON, EQUIVALENCE, or DATA statements within a program unit.

Dummy arguments representing array names must appear within the subprogram in a DIMENSION or type statement giving dimension information. If dummy arguments are not dimensioned, they cannot be referenced as an array in a subprogram.

In a subprogram, the definition of a dummy argument that is associated with a constant actual argument or an entity in a common block in the same subprogram is prohibited. If a subprogram reference causes two dummy arguments to be associated, the definition of either in the referenced subprogram is prohibited.

Example 1:

Calling Program	Subprogram
.	
.	
.	
W(I, J) = FA + FB - GRATER(C - D, 3 * AX / BX)	FUNCTION GRATER(A, B) IF (A .GT. B) 1, 2
.	1 GRATER = A - B
.	RETURN
.	2 GRATER = A + B
.	RETURN
.	END

This example shows the normal use of arguments in a function subprogram. The actual argument C-D is used in place of the dummy argument A and 3\*AX/BX is substituted for dummy argument B when the function subprogram is executed.

Example 2:

CALL SUBA(1.5)	SUBROUTINE SUBA(R)
	IF (R.NE.0) R = 0

This example contains a prohibited definition of a dummy argument, R, which is associated with a constant actual argument.

Example 3:

CALL SUBB (X, X)	SUBROUTINE SUBB (A, B)
	.
	.
	.
	A = Y
	Z = B

This example contains a prohibited definition of a dummy argument, A, which has been previously associated with another dummy argument, B, in the referencing program unit.

Example 4:

COMMON X	SUBROUTINE SUBC (B)
CALL SUBC (X)	COMMON A
.	.
.	.
.	.
	A = Y
	Z = B

This example contains a prohibited definition of a dummy argument, B, which is associated with an entity in common, A, in the same subprogram.

## USING COMMON

Common can be used to transfer values between a calling program unit and a subprogram. Passing values through common is more efficient than passing values through arguments in a CALL statement or function reference.

The definition of a dummy argument in a subprogram that is associated with an entity in a common block in the same subprogram is prohibited.

Example:

```
PROGRAM CMN (INPUT,OUTPUT)
COMMON NED (10)
READ 3,NED
3 FORMAT (10I3)
CALL JAVG
STOP
END
SUBROUTINE JAVG
C THIS SUBROUTINE COMPUTES THE AVERAGE OF THE FIRST 10 ELEMENTS IN
C COMMON
COMMON N(10)
ISTORE = 0
DO 1 I = 1,10
1 ISTORE = ISTORE + N(I)
ISTORE = ISTORE/10
PRINT 2,ISTORE
2 FORMAT (*1AVERAGE = *,I10)
RETURN
END

AVERAGE =          45
```

The array NED in program CMN and the array N in subroutine JAVG share the same locations in common. NED(1) shares the same location with N(1), NED(2) with N(2), etc. The values read into locations NED(1) through NED(10) are available to subroutine JAVG. JAVG computes and prints the average of these values.

## USING ARRAYS

The array dimensions in a subprogram must be the same as those in the calling routine if the subscripts are to agree between the called and calling program units. If a dummy argument is not dimensioned, it cannot be referenced as an array in the subprogram.

If any of the entries in a subscript of a type or DIMENSION statement is an integer variable name, the array is called an adjustable array. The variable names are called adjustable dimensions. Such an array can only appear in a procedure subprogram. The dummy argument list of the subprogram must contain the array name and the integer variable names that represent the adjustable dimensions. The values of the actual arguments that represent array dimensions in the argument list of the reference must be defined prior to calling the subprogram and cannot be redefined during execution of the subprogram. The absolute

size of the actual array may not be exceeded. For every array appearing in an executable program, there must be at least one constant array dimension associated through subprogram references.

In a subprogram, an array name that appears in a COMMON statement must have fixed dimension specifications.

## REFERENCING A FUNCTION

A function is referenced when the name appears in an expression. A function must not directly or indirectly reference itself. The reference can appear anywhere in an expression that an operand can be used.

When a statement function or intrinsic function is referenced, instructions are generated in-line to evaluate the function. The value is computed with the actual arguments substituted for the dummy arguments in the definition.

When a function subprogram or a basic external function is referenced, control is transferred to the function subprogram and the values of the actual arguments are substituted for the dummy arguments. Control is returned to the referencing program unit when a RETURN is encountered.

Actual arguments in a function subprogram reference may be an arithmetic or logical expression, constant (including Hollerith), variable, array name, array element name, subroutine subprogram name, external function name (not intrinsic function or statement function), or function reference (the function reference is a special case of an arithmetic expression).

Example:

### Calling Program

```
Z=A+B-JOE(3.*P,Q-1)
```

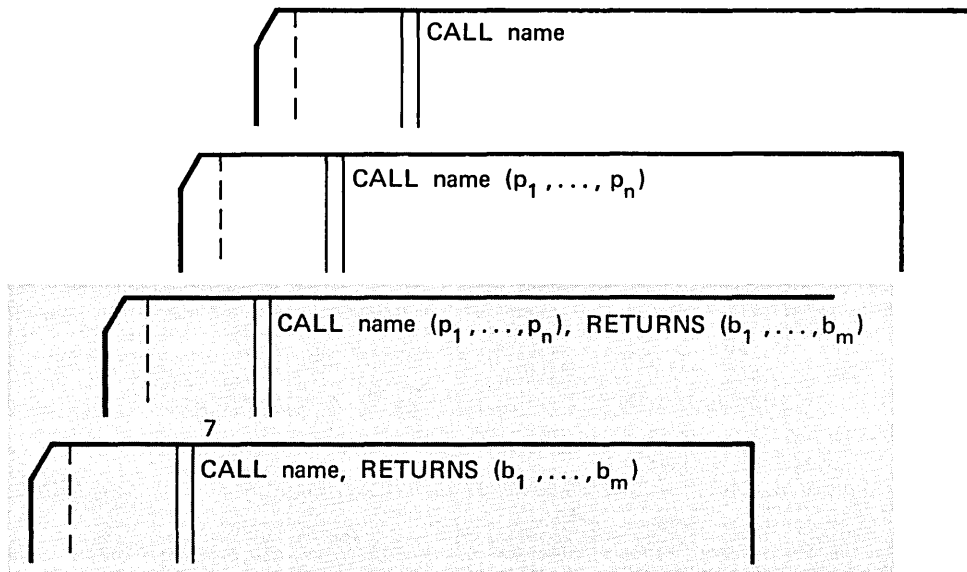
```
.  
. .  
. .  
R=S+JAM(Q,2.5*P)  
. .  
. .  
. .
```

### Function Subprogram

```
FUNCTION JOE(X,Y)  
10 JOE=X+Y  
RETURN  
ENTRY JAM  
IF(X.GT.Y)10,20  
20 JOE=X-Y  
RETURN  
END
```

Function subprogram JOE is executed as a result of its name appearing in another program unit.

## CALLING A SUBROUTINE SUBPROGRAM



name Name of subroutine called.

$p_1, \dots, p_n$  Actual arguments which must correspond in order, number, type, and LEVEL with those specified in the SUBROUTINE statement.

$b_1, \dots, b_m$  Actual statement labels in the calling program unit that correspond in order and number with the dummy statement label arguments in the SUBROUTINE statement. This specification can be omitted if control returns to the statement immediately following the CALL statement.

The total number of arguments must not exceed 63.

A subroutine subprogram is executed when a CALL statement is encountered in a program unit. The CALL statement transfers control to the subroutine and either a RETURN or a RETURN i in the subroutine returns control to the calling program unit. If a RETURN is encountered, control is transferred to the first executable statement following the CALL statement. If RETURN i is encountered, control is transferred to the statement corresponding to i in the RETURNS list. (The RETURN statement in section I-5 and Subroutine Subprogram in this section contain further details on the RETURNS list.)

The CALL statement can contain actual arguments and statement labels. They must correspond in order, number, type, and memory level to those in the subroutine subprogram definition.

The name in the CALL statement can be an alternate entry point in a subroutine subprogram, as specified in an ENTRY statement (described later in this section), or a subroutine name. The subroutine name must not appear in any specification statement in the calling program except an EXTERNAL statement.

Actual arguments in a subroutine subprogram call can be any of the following: arithmetic or logical expression, constant, variable, array name, array element name, subroutine subprogram name, basic external



function name (not an intrinsic or statement function name), function reference (the function reference is a special case of an arithmetic expression).

Example 1:

Calling Program	Subprogram
DO 5 I = 1,20	SUBROUTINE GRATER (A,B)
.	IF (A.GT.B) 1,2
.	1 B = A - B
5 CALL GRATER (STACK(I),TEMP(I))	RETURN
.	2 B = A + B
.	RETURN
	END

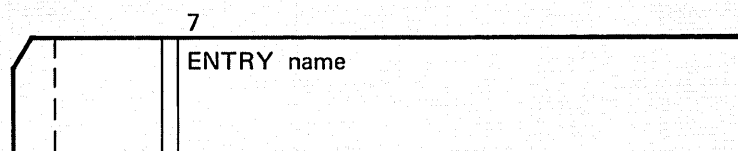
The subroutine subprogram GRATER is called 20 times since the CALL statement as the last statement in a DO loop causes looping to continue until the DO loop terminal parameter, 20, is satisfied.

Example 2:

Calling Program	Subroutine Subprogram
PROGRAM MAIN(INPUT,OUTPUT)	
.	
.	
10 CALL XCOMP(A,B,C),	SUBROUTINE XCOMP (B1,B2,G),
XRETURNS(101,102,103,104)	XRETURNS(A1,A2,A3,A4)
.	
.	IF(B1*B2-4.159)10,20,30
101 CONTINUE	10 CONTINUE
.	.
.	.
GO TO 10	RETURN A1           Return to 101
102 CONTINUE	20 CONTINUE
.	.
.	.
GO TO 10	RETURN A2           Return to 102
103 CONTINUE	30 CONTINUE
.	.
.	.
GO TO 10	IF (B1)40,50
104 CONTINUE	40 RETURN A3        Return to 103
END	50 RETURN A4        Return to 104
	END

The values of A, B, and C in the CALL statement replace B1, B2, and G in the SUBROUTINE statement for use in the subprogram XCOMP. Statement numbers 101, 102, 103, and 104 replace A1, A2, A3, and A4 in the subprogram and RETURN i statements.

## USING THE ENTRY STATEMENT



name is an entry point in a procedure subprogram.

The ENTRY statement defines an alternate entry point, which is other than the first executable statement, in a procedure subprogram. The ENTRY statement can appear anywhere an executable statement can appear in the subprogram except within the range of a DO where it is ignored and a warning diagnostic is issued. A procedure subprogram can contain any number of ENTRY statements. The first executable statement following ENTRY becomes the alternate entry point to the subprogram. ENTRY statements cannot be labeled and cause a fatal-to-execution error in a main program unit.

In the subprogram, the entry name can appear only in the ENTRY statement and each name must appear in a separate statement. A function entry name must be the same type as the name in the FUNCTION statement, and it must be unique within the program.

In the calling program, the reference to the entry name is made just as if reference were being made to the function subprogram or subroutine subprogram in which the entry name is contained. The name can appear in an EXTERNAL statement, and if it is a function subprogram entry name, in a type statement.

The dummy arguments, if any, appearing with the FUNCTION statement or SUBROUTINE statement do not appear with the ENTRY statement, but are assumed to be the same as for the main entry point.

In a function subprogram, the value of the function is the last value assigned to the name of the function, regardless of which ENTRY statement was used to enter the subprogram. The function name is used to return results to the calling program even though the reference was through an entry name.

Example 1:

Calling Program	Subroutine Subprogram
COMMON SET1 (25)	SUBROUTINE CLEAR (ARAY)
.	DIMENSION ARAY (25)
.	DO 100 I = 1,25 ← Main entry point
CALL CLEAR (SET1)	100 ARAY (I) = 0.0
.	ENTRY FILL
.	3 READ 2, VALUE, IPLACE ← Alternate entry point
.	2 FORMAT (10X, F7.2, I4)
CALL FILL (SET1)	ARAY (IPLACE) = VALUE
.	IF (IPLACE .GT. 24) RETURN
.	GO TO 3
.	END

At some point in the calling program, a call is made to the subroutine: CALL CLEAR (SET1). The array SET1 is set to zero and values are read into the array. Later in the program, a call is made again to the subroutine CLEAR; but this time it is entered at the entry point FILL. When FILL is called, further values are read into the array SET1 without first setting the array to zero.

Example 2:

**Calling Program**

```
RESULT=FSHUN(X,Y,Z)
RES2=FRED(R,S,T)
```

**Subprogram**

```
FUNCTION FSHUN(A,B,C)
3 FSHUN=A*B/C**2
RETURN
ENTRY FRED
IF(A .LE. 702.) GO TO 3
FSHUN=(C+A)/B
RETURN
END
```

When the FUNCTION FSHUN is entered at the beginning of the function, or through the ENTRY FRED, the result will be returned to the calling program through the function name FSHUN.

Example 3:

```
FUNCTION CAT(A,B)
.
.
.
DOG=10.+3.2
ENTRY DOG
```

The ENTRY name DOG is not valid because it has been used as a variable.



FORTRAN Extended provides certain procedures that are of general utility or difficult to express in FORTRAN; they are referenced in the same way as user-written procedures. The three classes of FORTRAN Extended supplied procedures are: intrinsic functions, basic external functions, and utility subprograms.

## INTRINSIC FUNCTIONS

An intrinsic function is a compiler-defined procedure that returns a single value. It is inserted in the referencing program at compile time. The form of the intrinsic function reference is the same as the statement function reference outlined in section I-7.

When a variable, array, or statement function is defined with the same name as that of an intrinsic function, the user-supplied definition prevails.

When a function subprogram is defined with the same name as that of an intrinsic function, the user definition prevails only if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that changes the type associated with the intrinsic function.

In a calling program unit, if the name of an intrinsic function appears either in an EXTERNAL statement or in an explicit type statement that changes the type associated with the function, the user must supply a function subprogram with the name of that function.

Table 8-1 lists the intrinsic functions provided by FORTRAN Extended. The results of functions with the type listed as no mode assume the type of the expression in which they are used, unless that type is logical (in which case the function result remains typeless).

Table 8-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Absolute Value	$ A $	1	ABS IABS DABS	Real Integer Double	Real Integer Double	Y=ABS(X) J=IABS(I) DOUBLE A,B B=DABS(A)
Truncation	Sign of A times largest integer $\leq  A $ for $ A  \leq 2^{48}-1$	1	AINT INT IDINT	Real Real Double	Real Integer Integer	Y=AINT(X) I=INT(X) DOUBLE Z J=IDINT(Z)
Remainder- ing † (see note)	$A1 \text{ (mod } A2)$	2	AMOD MOD ††	Real Integer	Real Integer	B=AMOD(A1,A2) J=MOD(I1,I2)
Choosing largest value	$\text{Max}(A1, A2, \dots)$	2 - 63	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double	X=AMAX0(I,J,K) A=AMAX1(X,Y,Z) L=MAX0(I,J,K,N) I=MAX1(A,B) DOUBLE W,X,Y,Z W=DMAX1(X,Y,Z)
Choosing smallest value	$\text{Min}(A1, A2, \dots)$	2 - 63	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double	Y=AMIN0(I,J) Z=AMIN1(X,Y) L=MIN0(I,J) J=MIN1(X,Y) DOUBLE A,B,C C=DMIN1(A,B)
Float	Conversion from integer to real	1	FLOAT	Integer	Real	X1=FLOAT(I)

† MOD or AMOD (a,b) is defined as  $a-[a/b]b$ , where  $[X]$  is the largest integer that does not exceed the magnitude of X with sign the same as X. The results are not defined when the second argument is zero.

†† The arguments of MOD must each be less than or equal to  $2^{47}-1$ .

Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Fix	Conversion from real to integer Same as INT	1	IFIX	Real	Integer	IY=IFIX(Y)
Transfer of Sign	Sign of A2†† with  A1	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double	Z=SIGN(X,Y) J=ISIGN(I1,I2) DOUBLE X,Y,Z Z=DSIGN(X,Y)
Positive Difference	If A1>A2 then A1-A2. If A1 ≤ A2 then 0.	2	DIM IDIM	Real Integer	Real Integer	A=DIM(C,D) J=IDIM(I1,I2)
Logical Product	Bit-by-bit logical AND of A <sub>1</sub> through A <sub>n</sub>	2 - 63	AND	any type †	no mode	A=AND(X,Y,Z)
Logical Sum	Bit-by-bit logical OR of A <sub>1</sub> through A <sub>n</sub>	2 - 63	OR	any type †	no mode	A=OR(X,Y,Z)
Exclusive OR	Bit-by-bit Exclusive OR of A <sub>1</sub> through A <sub>n</sub>	2 - 63	XOR	any type †	no mode	A=XOR(X,Y,Z)
Complement	Bit-by-bit Boolean complement of A	1	COMPL	any type †	no mode	B=COMPL(A)

† For a double precision or complex argument, only the high order or real part is used.

†† For functions SIGN, ISIGN, and DSIGN, the sign of the second argument is defined as positive when the value of that argument is +0 and negative when the value is -0.

Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Shift	Shift A1, A2 bit positions: left circular if A2 is positive; right with sign extension, and end off if A2 is negative. $0 \leq  A2  \leq 60^\dagger$	2	SHIFT	A1: any type $\dagger\dagger$ A2: integer	no mode	B=SHIFT(A,I)
Mask	Form mask of A1 bits set to 1 starting at the left of the word. $0 \leq A1 \leq 60^\dagger$	1	MASK	Integer	no mode	A=MASK(I)
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real	DOUBLE Y X=SNGL(Y)
Obtain Real Part of Complex Argument		1	REAL	Complex	Real	COMPLEX A B=REAL(A)

$\dagger$ MASK and SHIFT are undefined for arguments outside these bounds.

$\dagger\dagger$  For a double precision or complex argument, only the most significant or real part is used.



Table 8-1. Intrinsic Functions (Continued)

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real	COMPLEX A D=AIMAG(A)
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double	DOUBLE Y Y=DBLE(X)
Express Two Real Arguments In Complex Form	$A1+A2i$ (where $i^2 = -1$ )	2	CMPLX	Real	Complex	COMPLEX C C=CMPLX(A1,A2)
Obtain Conjugate of a Complex Argument	$a-bi$ (where $A=a+bi$ )	1	CONJG	Complex	Complex	COMPLEX X,Y Y=CONJG(X)
Random Number Generator	Returns values uniformly distributed over the range (0,1); dummy argument is ignored.	1	RANF	any type	Real	Y=RANF(A)
Obtain address of a variable, array element, or entry point of external subprogram	Argument is the name of a variable, array element, or external subprogram	1	LOCF	any type	Integer	J=LOCF(Q)

## BASIC EXTERNAL FUNCTIONS

A basic external function is a predefined procedure included with the FORTRAN Common Library. These procedures are used to evaluate standard mathematical functions such as sine, cosine, square root, etc. A basic external function is referenced by the appearance of the function name with appropriate arguments in an expression.

A basic external function ordinarily is called by value; however, it is called by name if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the function, or if option T, D, or OPT=0 is specified on the FTN control card. (Section III-10 contains a description of Call By Value and Call By Name.)

When a variable, array, or statement function is defined with the same name as that of a basic external function, the user definition overrides the system definition.

When a FUNCTION subprogram is defined with the same name as that of a basic external function, the user definition overrides the library definition only if, in the calling program unit, the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the library function, or if option T, D, or OPT=0 is specified on the FTN control card.

Table 8-2 lists the basic external functions.

Arguments for which a result is not mathematically defined, or those of a type other than that specified, should not be used. Arguments of the trigonometric functions are in radians, and the inverse trigonometric functions return principal values.

If the name of the function appears either in an EXTERNAL statement or in an explicit type statement that overrides the type associated with the library function, or if option T, D, or OPT=0 is specified on the FTN control card, the arguments of all external functions are checked to ensure that they are neither indefinite nor infinite and fall within the limits listed in the Definition column of table 8-1. Argument checking is provided unconditionally for the following functions: EXP, ALOG, ALOG10, SIN, COS, SQRT, ATAN, ATAN2, ASIN, ACOS, CABS, SINH, and COSH. An informative diagnostic is provided when an argument is found to be invalid.

Table 8-2. Basic External Functions

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Exponential	$e^A$ $-675.84 \leq A \leq 741.67$	1	EXP	Real	Real	Z=EXP(Y) DOUBLE X,Y Y=DEXP(X) COMPLEX A,B B=CEXP(A)
		1	DEXP	Double	Double	
	$e^{(X+iY)}$ $-675.84 \leq X \leq 741.67$ $ Y  \leq \pi \times 2^{46}$	1	CEXP	Complex	Complex	
Natural Logarithm	$\log_e(A)$ $A > 0$	1	ALOG	Real	Real	Z=ALOG(Y) DOUBLE X,Y Y=DLOG(X) COMPLEX A,B B=CLOG(A)
		1	DLOG	Double	Double	
	$\log_e(X+iY)$ $X^2+Y^2 \neq 0$	1	CLOG <sup>†</sup>	Complex	Complex	
Common Logarithm	$\log_{10}(A)$ $A > 0$	1	ALOG10 DLOG10	Real Double	Real Double	B=ALOG10(A) DOUBLE D,E E=DLOG10(D)
Trigono- metric Sine	$\sin(A)$ $ A  \leq \pi \times 2^{46}$	1	SIN	Real	Real	Y=SIN(X) DOUBLE D,E E=DSIN(D) COMPLEX CC,F CC=CSIN(F)
		1	DSIN	Double	Double	
	$\sin(X+iY)$ $ X  \leq \pi \times 2^{46}$ $ Y  \leq 741.67$	1	CSIN	Complex	Complex	
Trigono- metric Cosine	$\cos(A)$ $ A  \leq \pi \times 2^{46}$	1	COS	Real	Real	X=COS(Y) DOUBLE D,E E=DCOS(D) COMPLEX CC,F CC=CCOS(F)
		1	DCOS	Double	Double	
	$\cos(X+iY)$ $ X  \leq \pi \times 2^{46}$ $ Y  \leq 741.67$	1	CCOS	Complex	Complex	
Hyperbolic Tangent	$\tanh(A)$ $ A  \leq 741.67$	1	TANH	Real	Real	B=TANH(A)
Hyperbolic Sine	$\sinh(A)$ $ A  \leq 741.67$	1	SINH	Real	Real	B=SINH(A)
Hyperbolic Cosine	$\cosh(A)$ $ A  \leq 741.67$	1	COSH	Real	Real	B=COSH(A)

<sup>†</sup>CLOG returns values with imaginary parts in the range  $(-\pi, \pi]$ . For  $x < 0$ , therefore, CLOG( $x+i0$ ) returns an imaginary part with a value  $=+\pi$ ; CLOG( $x+i0^+$ ) returns an imaginary part with a value  $\approx +\pi$ ; and CLOG( $x-i0^+$ ) returns an imaginary part with a value  $\approx -\pi$ .

Table 8-2. Basic External Functions (Continued)

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function	Example
Square Root	$(A)^{1/2}$ $A \geq 0$	1	SQRT	Real	Real	Y=SQRT(X) DOUBLE D,E E=DSQRT(D) COMPLEX CC,F CC=CSQRT(F)
		1	DSQRT	Double	Double	
		1	CSQRT <sup>†</sup>	Complex	Complex	
Arctangent	arctan (A)	1	ATAN <sup>††</sup>	Real	Real	Y=ATAN(X) DOUBLE D,E E=DATAN(D) B=ATAN2(A1,A2) DOUBLE D,D1,D2 D=DATAN2(D1,D2)
		1	DATAN <sup>††</sup>	Double	Double	
	arctan (A1/A2) $A1^2+A2^2 \neq 0$	2	ATAN2 <sup>†††</sup>	Real	Real	
		2	DATAN2 <sup>†††</sup>	Double	Double	
Remaindering	A1 (mod A2)	2	DMOD <sup>§</sup>	Double	Double	DOUBLE DM,D1,D2 DM=DMOD(D1,D2)
Modulus	$\sqrt{a^2+b^2}$ A=a+bi	1	CABS	Complex	Real	COMPLEX C CM=CABS(C)
Arccosine	arccos (A) $ A  \leq 1$	1	ACOS <sup>§§</sup>	Real	Real	X=ACOS(Y)
Arcsine	arcsin (A) $ A  \leq 1$	1	ASIN <sup>§§§</sup>	Real	Real	X=ASIN(Y)
Trigonometric Tangent	tan (A) $ A  \leq \pi \times 2^{46}$	1	TAN	Real	Real	X=TAN(Y)

<sup>†</sup>CSQRT returns values in the right half plane.

<sup>††</sup>ATAN and DATAN return values in the range  $(-\frac{\pi}{2}, \frac{\pi}{2})$ .

<sup>†††</sup>ATAN2 and DATAN2 return values in the range  $(-\pi, \pi]$ . For  $x < 0$ , therefore, ATAN2(0,x) returns a value =  $+\pi$ ; ATAN2(0<sup>+</sup>,x) returns a value  $\approx +\pi$ ; and ATAN2(0<sup>-</sup>,x) returns a value  $\approx -\pi$ .

<sup>§</sup> The function DMOD (a,b) is defined as  $a - [a/b]b$ , where  $[X]$  is the largest integer that does not exceed the magnitude of X with sign the same as X; the result is not defined when the second argument is zero.

<sup>§§</sup> ACOS returns values in the range  $[0, \pi]$ . or <sup>§§§</sup> ASIN returns values in the range  $(-\frac{\pi}{2}, \frac{\pi}{2})$ .

## ADDITIONAL UTILITY SUBPROGRAMS

The utility subprograms described below are supplied by the system and are always called by name (section III-10 gives more details). A user-supplied subprogram with the same name as a library subprogram overrides the library subprogram.

In the definitions listed under the routines:

i and n are integer variables, constants, or expressions.

j is an integer variable.

a and b are variable or array names of any type.

u is a unit number or file name (nLx . . . x).

H is a Hollerith specification.

## OPERATING SYSTEM INTERFACE ROUTINES

### DATE(a) or CALL DATE(a)<sup>†</sup>

The current date is returned as the value of argument a in the form 10Hbmm/dd/yyb (unless it is changed at installation option), where b denotes a blank, mm is the number of the month, dd is the number of the day within the month, and yy is the year. The value returned is Hollerith data and can be output using an A format specification (PROGRAM LIBS, section II).

The type of the function DATE is real; thus if J and K are integer variables as in:

J = DATE(K)

J will not be useful because the value returned will have been converted from real to integer.

### JDATE(a) or CALL JDATE(a)<sup>‡</sup>

The current date is returned as the value of argument a in the form 5Ryyddd, where yy is the year and ddd is the number of the day within the year. The value returned is Hollerith data and can be output using an R format specification.

---

<sup>†</sup>These routines can be used as functions or subroutines. The value is returned via the argument and the normal function return.

<sup>‡</sup>This routine is not available under SCOPE 2.1.

### **SECOND(t) or CALL SECOND(t)<sup>†</sup>**

The central processor time is returned from start-of-job in seconds as a real number, usually accurate to two decimal places. t is a real variable.

Example:                   DPTIM = SECOND (CP)

### **TIME(a) or CALL TIME(a)<sup>†</sup>**

The current reading of the system clock is returned as the value of argument a in the form 10Hbhh.mm.ss. , where b denotes a blank, and hh, mm, and ss are the number of hours, minutes, and seconds, respectively. The value returned is Hollerith data and can be output using an A format specification (PROGRAM LIBS, section II).

The type of the function TIME is real; thus if J and K are integer variables in the following statement, J is not useful because the value returned will have been converted from real to integer.

Example:                   J = TIME(K)

### **CALL DISPLA (H,k)**

A name and a value are placed in the dayfile. H is a Hollerith specification of not more than 80 characters; k is a real or integer variable or expression and is displayed as an integer or real value.

Example:                   CALL DISPLA (7H TIME =, STOP-START)

### **CALL REMARK (H)**

Places a message in the dayfile. Under SCOPE 2.1, the maximum message length is 90 characters displayed on one line. Under KRONOS 2.1, NOS 1.0, and SCOPE 3.4, the maximum message length is 80 characters displayed 40 characters per line. A message exceeding the maximum length is truncated. A message shorter than the maximum must have all zeros in the lower 12 bits of the last word. These zeros are automatically supplied when a Hollerith constant is used as the parameter.

Example:                   CALL REMARK (9HLAST DECK)

### **CALL SLITE(i)**

Sense light i is turned on. If i = 0, all sense lights are turned off. If i is other than 0-6, an informative diagnostic is printed and sense lights are not changed.

---

<sup>†</sup>These routines can be used as functions or subroutines. The value is returned via the argument and the normal function return.

### CALL SLITET(i,j)

Sense light *i* is tested. If sense light *i* is on, *j* = 1; if sense light *i* is off, *j* = 2. If *i* is other than 1-6, an informative diagnostic is printed, all sense lights remain unchanged, and *j* = 2. Execution turns off sense light *i* if it is on.

(Note: Logical variables generally provide a more efficient method of testing a condition than do calls to SLITE or SLITET.)

### CALL SSWTCH(i,j)

If sense switch *i* is on, *j* is set to 1; if sense switch *i* is off, *j* is set to 2. *i* is 1 to 6. If *i* is out of range, an informative diagnostic is printed, and *j* is set to 2. The sense switches are set or reset by the computer operator or by a SWITCH control card.

### CALL OVERLAY(fname, primary, secondary, recall,k)

See section I-12.

### CALL EXIT

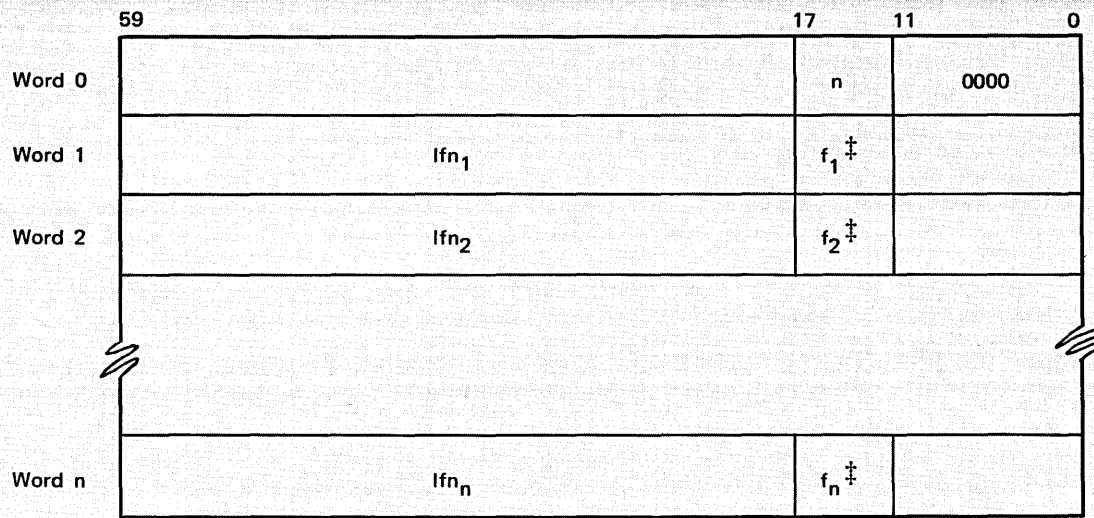
Program execution is terminated and control is returned to the operating system. (Note: use of the STOP statement is preferable to CALL EXIT.)

### CALL CHEKPTX(filelist)<sup>†</sup>

A checkpoint dump of the files specified by filelist is taken:

filelist	Is one of the following:
0	Checkpoint dump all files.
filearray	Checkpoint files specified in integer array filearray, which has the following structure:

<sup>†</sup>Available only on SCOPE 3.4 and SCOPE 2.1.



n      Number of files in following list, to a maximum of 42 (decimal).

lfn<sub>i</sub>      Name (in left justified display code) of user mass storage files to be processed.

f<sub>n</sub>      Octal number indicating specific manner in which lfn is to be processed.

- 0      Mass storage file is copied from beginning-of-information to its position at check-point time, and only that portion will be available at restart. The file is positioned at the latter point.
- 1      Mass storage file is copied from its position at check point time to end-of-information, and only that portion will be available at restart. The file is positioned at the former point.
- 2      Mass storage file is copied from beginning-of-information to end-of-information; the entire file will be available at restart time. The file is positioned at the point at which the checkpoint was taken.
- 3      The last operation on the file determines how the mass storage file is copied.

‡ Does not apply to SCOPE 2.1.



**Example:**

```
.  
. .  
. .  
DIMENSION IFILES(4)  
IFILES(1) = 30000B  
IFILES(2) = 5LTAPE1 .OR. 10000B  
IFILES(3) = 5LTAPE2 .OR. 30000B  
IFILES(4) = 5LTAPE3  
. .  
. .  
CALL CHECKPTX(IFILES)
```

For more information, refer to the SCOPE 3.4 Reference Manual or the SCOPE 2 Reference Manual.

**CALL RECOVER(name,flags, checksum)<sup>†</sup>**

name	Name of subroutine to be executed if flagged conditions occur.
flags	Octal value for conditions under which recovery code is to be executed, as outlined below. Conditions can be combined as desired, with octal values up to 177 allowed.
001	Arithmetic mode error.
002	PP call or auto-recall error.
004	Time or storage limit exceeded.
010	Operator drop, kill, or rerun.
020	System abort.
040	CP abort.
100	Normal termination.

checksum Last word address of recovery code to be checksummed; 0 if no checksum is desired.

The RECOVER subroutine allows a user program to gain control at the time that normal or abnormal job termination procedures would otherwise occur. Initialization of RECOVER at the beginning of a program establishes the conditions under which control is to be regained and specifies the address of user recovery code. If the stated condition occurs during program execution, control returns to the user code. If necessary, the system increases the CP time limit, I/O time limit, or mass storage time limit to provide an installation defined minimum of time and mass storage for RECOVER processing. No limit is increased more than once in a job. RECOVER can be called more than once during program initialization to reference different user recovery subroutines. These calls to RECOVER can use different combinations of conditions for the same or different user recovery subroutines.

If the checksum parameter is zero, no checksum is done.

<sup>†</sup> Is not available on SCOPE 2.1.

If one of the user's selected error conditions occurs, RECOVER gains control, performs internal tasks, and then transfers control to the user's recovery subroutine(s). The following three arguments are passed to the user's recovery subroutine:

1. A 17-word integer array. The first 16 words are an image of the exchange package; the seventeenth word is the contents of RA+1.
2. A flag that upon return, determines the type of program termination. If the user's recovery subroutine sets the flag non-zero, ENDRUN termination occurs upon completion of the last post-processing subroutine. If the flag remains zero, the original error code, as well as the exchange package, are restored and the job continues as if RECOVER had not been called. Altering the exchange package passed as argument 1 prevents the correct completion of the restore, but does not impair system operation.
3. An array, starting at RA+1, that allows a FORTRAN subroutine to access all of the user's field length.

Example:

```
PROGRAM MAIN(INPUT,OUTPUT)
EXTERNAL REPREV,CHKSUM
.
.
.
CALL RECOVER(REPREV,72B,LOCF(CHKSUM))
.
.
.
STOP
END
SUBROUTINE REPREV(IXCHNG,IFLAG,IPLDLN)
DIMENSION IXCHNG(17), IPLDLN(40000B)
IFLAG = 1
PRINT 10, IXCHNG, (IPLDLN(I), I=0,100B)
10 FORMAT (3(6X, O20))
RETURN
ENTRY CHKSUM ←————— determines end of code to be checksummed
END
```

## DEBUGGINGS AIDS

**CALL DUMP** ( $a_1, b_1, f_1, \dots, a_n, b_n, f_n$ )

**CALL PDUMP** ( $a_1, b_1, f_1, \dots, a_n, b_n, f_n$ )

Dumps main memory on the OUTPUT file in the indicated format, PDUMP returns control to the calling program; DUMP terminates program execution.  $a_i$  and  $b_i$  specify the beginning and the end of the storage area to be dumped.  $1 \leq n \leq 20$ .  $f$  is a format indicator, as follows:

$f = 0$ or $3$	octal dump
$f = 1$	real dump
$f = 2$	integer dump

For  $f$  values 0-3,  $a_i$  and  $b_i$  are the first and last words dumped. If 4 is added to any  $f$  value, the contents of  $a_i$  and  $b_i$  are used as the addresses of the first and last words dumped within the job's field length. An ASSIGN statement or the LOCF function can be used to get addresses for the  $a_i$  and  $b_i$  parameters.

Examples:	<b>CALL PDUMP</b> (A(1),A(100), 1)	Dumps from A(1) to A(100) as real numbers
	<b>CALL PDUMP</b> (0, 1000B, 4)	Dumps from location 0 to 1000B in octal

## CALL STRACE

Provides traceback information from the subroutine calling STRACE back to the main program. Traceback information is written to the file DEBUG. To obtain traceback information interspersed with the source program, DEBUG should be equivalenced to OUTPUT in the PROGRAM statement. (Refer to STRACE, section I-13.)

## LEGVAR(a)

Checks the value of variable  $a$ . Returns the result -1 if variable is indefinite, +1 if out of range, and 0 if normal. Variable  $a$  is type real; result is type integer.

## CALL SYSTEM (errnum,mesg)

errnum is an error number, mesg is an error message. Refer to section III-3 for further information.

## CALL SYSTEMC(errnum, speclist)

SYSTEMC allows for non-standard processing of library detected errors. Refer to section III-3 for further information.

### CALL ERRSET(num,lim)

For error numbers 78 and 79 on formatted, list directed, and NAMELIST reads, this subroutine sets maximum number of errors, lim, allowed in input data before termination. Error count is kept in location num. Refer to section III-3 for further information.

## RANDOM NUMBER GENERATOR

### RANF (n)<sup>†</sup>

Random number generator. Returns values uniformly distributed over the range (0,1); the values 0 and 1 are excluded. n is a dummy argument which is ignored. Result is type real.

### CALL RANSET(n)

Initializes seed of RANF. n is a bit pattern. Bit  $2^0$  will be set to 1 (forced odd), and bits  $2^{59} - 2^{48}$  will be set to 1717 octal.

### CALL RANGET(n)

Obtains current seed of RANF between 0 and 1. n is a symbolic name to receive the seed. It is not necessarily normalized. The value returned may be passed to RANSET at a later time to regenerate the same sequence of random numbers.

## MASS STORAGE INPUT/OUTPUT

Refer to section III-7 for further information on the following routines:

### CALL OPENMS (u,ix,lngth,t)

Opens mass storage file and informs Record Manager that file u is word addressable. If an existing file is opened, the master index is read into the area specified by the program. u is the unit designator. ix is the first word address of the index in central memory. lngth is the length of the index buffer; for a name index,  $\text{lngth} \geq 2 * (\text{number of records in file}) + 1$ ; for a number index,  $\text{lngth} \geq \text{number of records in file} + 1$ . t = 1: file is referenced through a name index; t = 0: file is referenced through a number index.

Example:       PROGRAM MS1 (TAPE3)  
                  DIMENSION INDEX (11), DATA (25)  
                  CALL OPENMS (3,INDEX,11,0)

---

<sup>†</sup>RANF is an intrinsic function.

### CALL STINDX (u,ix,lngth,t)

Changes index in central memory from master to subindex. u,ix,lngth,t are the same as for OPENMS. If t is omitted, whether the subindex is considered to be a name or a number index is determined by the value of t specified in the most recent call to STINDX or OPENMS.

Example:       CALL STINDX (2,SUBIX,10,0)

### CALL CLOSMS (u)

Writes index from central memory to file and closes file. CLOSMS should be called before job termination.

Example:       CALL CLOSMS (7)

### CALL READMS (u,fwa,n,k)

Transmits data from mass storage to central memory. u is the unit designator, fwa is the central memory address of the first word of the data area. n is the number of central memory words transferred. Number index k has limits  $1 \leq k \leq \text{lngth} - 1$ . Name index k = any 60-bit quantity except  $\pm 0$ .

Example:       CALL READMS(3,DATA,25,6)

### CALL WRITMS(u,fwa,n,k,r,s)

Transmits data from central memory to mass storage. u,fwa,n,k are the same as for READMS. r = +1: rewrites in place however, it does not rewrite, and fatal error is printed, if new record length exceeds old record length. r = -1 rewrites in place if space is available, otherwise writes at end of information. r = 0 no rewrite; writes at end of information. The r parameter can be omitted if the s parameter is omitted. The default value for r is 0 (normal write).

s = 1 writes subindex marker flag in index control word for this record. s = 0 does not write subindex marker flag in index control word for this record. The s parameter can be omitted; its default value is 0.

The s parameter is included for future random file editing routines. Current routines do not test the flag, but user should include this parameter in new programs, when appropriate, to facilitate transition to a future edit capability.

Example:       CALL WRITMS (3,DATA,25,NRKEY,1,1)

## INPUT/OUTPUT STATUS CHECKING

FORTRAN Extended provides the capability of checking for an end-of-file or a parity error condition following read operations via the functions UNIT, EOF, and IOCHEC.

Any of the following conditions encountered during a read returns an end-of-file status via the functions UNIT or EOF:

End-of-section (in the case of file INPUT only)

End-of-partition

End-of-information

Non-deleted W format flag record

Embedded tape mark

Terminating double tape mark

Terminating end-of-file label

Embedded zero length level 17 block

The functions UNIT and IOCHEC return a parity error indication for every record within or spanning a block containing a parity error; however, such an indication does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the Record Manager input/output routines.

§ Parity status can be checked on write operations that access mass storage files when the write check option has been specified on the REQUEST card for the file (SCOPE 2.1 Reference Manual). Write parity errors for other types of devices (such as staged/on-line tape) are detected by the operating system, and a message to this effect is written in the dayfile.

### UNIT(u)

The UNIT function is used to check the status of a BUFFER IN or BUFFER OUT operation for an end-of-file or parity error condition on logical unit u. The function returns the following values:

- 1. Unit ready, no end-of-file or parity error encountered on the previous operation
- +0. Unit ready, end-of-file encountered on the previous operation
- +1. Unit ready, parity error encountered on the previous operation

Example: IF (UNIT(5)) 12,14,16

Control transfers to the statement labeled 12, 14 or 16 if the value returned was -1., 0., or +1., respectively.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. If the UNIT function references a logical unit referenced by input/output statements other than BUFFER IN or BUFFER OUT, the status returned always indicates unit ready and no error (-1.).

§ Applies only to SCOPE 2.1.

## EOF(u)

The EOF function is used to test for an end-of-file condition on unit *u* following a formatted, list-directed, NAMELIST, or unformatted read. Zero is returned if no end-of-file is encountered, or a non-zero value if end-of-file is encountered.

Example:       IF (EOF(5)) 10,20

returns control to the statement labeled 10 if the previous read encountered an end-of-file; otherwise, control goes to statement 20.

If an end-of-file is encountered, EOF clears the indicator before returning control.

The EOF function returns a zero value following read or write operations on random access files, and also following write operations on all types of files, regardless of whether an end-of-file condition has been detected; therefore, the EOF function should not be used in those circumstances.

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

## IOCHEC(u)

The IOCHEC function tests for parity error on unit *u* following a formatted, list-directed, NAMELIST, or unformatted read. The value zero is returned if no error has been detected.

Example:       J = IOCHEC(6)  
              IF (J) 15,25

zero value would be returned to J if no parity error occurred and non-zero if an error had occurred; control would transfer to the statement labeled 25 or 15 respectively.

If a parity error occurred, IOCHEC would clear the parity indicator before returning. Parity errors are handled in this way regardless of the type of the external device.

## OTHER INPUT/OUTPUT SUBPROGRAMS

### LENGTH(u) or CALL LENGTHX(u,nw,ubc)

Returns information regarding the previous BUFFER IN or READMS call of the file designated by *u*. *nw* or the value of LENGTH is set to the number of 60-bit words read. *ubc* is set to the number of unused bits in the last word of the transfer. *nw*, *ubc*, and value returned are type integer.

Example:       NW = LENGTH(5)  
  
              or  
  
              CALL LENGTHX(5,NW,NUBC)

### **CALL LABEL(u,fwa)**

Sets tape label information for a file. u is the unit number. fwa is the address of the first word of the label information. See section III-5 for further information.

## **ECS/LCM SUBPROGRAMS**

### **CALL MOVLEV (a,b,n)**

Transfers n consecutive words of data between a and b. a and b are variables or array elements; n is an integer constant. a is the starting address of the data to be moved and b is the starting address of the receiving location.

Example:        `CALL MOVLEV(A,B,1000)`

No conversion is done by MOVLEV. If data from a real variable is moved to an integer type receiving field, the data remains real.

Example:        `CALL MOVLEV (A, I, 1000)`

After the move, I does not contain the integer equivalent of A.

Example:        `DOUBLE PRECISION D1(500), D2(500)`

`CALL MOVLEV (D1, D2, 1000)`

Since D1 is defined as double precision, n should be set to 1000 to move the entire D1 array.

### **CALL READEC(a,b,n)**

Transfers data from extended core storage to central memory.

a is a simple variable or array element located in central memory. b is a simple variable or array element located in an extended core storage block or LCM block. n is an integer constant or expression. n consecutive words of data are transferred beginning with a in central memory and b in extended core storage.

### **CALL WRITEC(a,b,n)**

Transfers data from central memory to extended core storage or LCM.

No type conversion is done.

`LEVEL 3,B`

`CALL READEC(A,B,10)`

`CALL WRITEC(A,B,10)`



## TERMINAL INTERFACE

Refer to section III-11 for further information on the following routines:

### CALL CONNEC(fd) or CALL CONNEC(fd,cs)

Associate file fd with terminal for input/output operations using the character set specified by cs.

### CALL DISCON(fd)

Disassociate file fd from terminal.

## CYBER RECORD MANAGER INTERFACE

These routines, interfacing with CYBER Record Manager, provide an alternative to standard FORTRAN I/O; they should not be used with files referenced by standard FORTRAN I/O routines. Refer to section III-6 for further information.

## SORT/MERGE INTERFACE

These routines, interfacing with Sort/Merge, provide an extended capability for processing data records in a FORTRAN program. Refer to section III-16 for further information.



---

To input or output data, the following information is required:

Unit number of the input/output device

List of FORTRAN variables to receive input data or from which results are to be output.

Layout or format of data

READ, WRITE, PRINT, or PUNCH statements specify the input or output device and the list. The form of data is designated by the FORMAT statement.

Data can be formatted or unformatted or list directed. In formatted mode, display code character strings are converted and transferred according to a FORMAT statement. In unformatted mode, data is transferred in the form in which it normally appears in storage, no conversion takes place, and no FORMAT statement is used. In list directed mode, display code character strings are converted and transferred according to the type of the list items.

Input/output control statements are discussed below. Input/output lists and the FORMAT statements are covered in section 10.

The following definitions apply to all input/output statements:

- u            Input/output unit; the operating system associates this unit with an internal file name which may be:
  - Integer constant of one or two digits (leading zeros are discarded). The compiler associates these numbers with file names of the type TAPEu, where u is the file designator (refer to PROGRAM statement, section 7).
  - Simple integer variable name with a value of:
    - 1 - 99, or
    - A display code file name (L format, left justified with binary zero fill). This is the internal logical file name.
- fn            Format designator; a FORMAT statement number or the name of an array containing the format specification. The statement number must identify a FORMAT statement in the program unit containing the input/output statement.
- iolist        Input/output list specifying items to be transmitted (section I-10).

All information is considered to be a file or part of a file. Local to a given job, a file is identified by a logical file name (the internal file named, u). All control card references to a file identify it by the logical file name. The internal central memory representation of a logical file name consists of its literal value in display code, left justified and zero filled.

Several file names are given special significance. When one of these names is used, the following automatic disposition is made, unless the user has defined an alternate disposition:

Card input is assigned to the file INPUT.

Data in the file OUTPUT is assigned to the printer.

Data in the file PUNCH is assigned to the card punch as coded card output.

Data in the file PUNCHB is output on the card punch as binary card output.

## **FORTRAN RECORD LENGTH**

For cards, formatted logical record length cannot exceed 80-characters and for print files, 137 characters. Other files are limited to 150 characters unless the maximum record length is specified on the PROGRAM statement (see section I-7).

The length of an unformatted FORTRAN logical record is determined by the length of the input/output list, and can be any size.

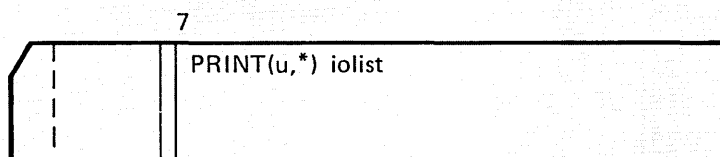
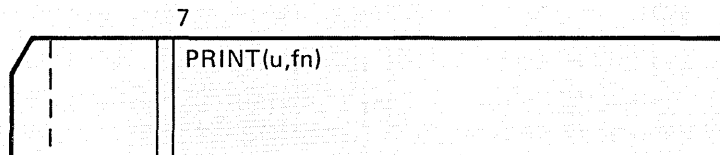
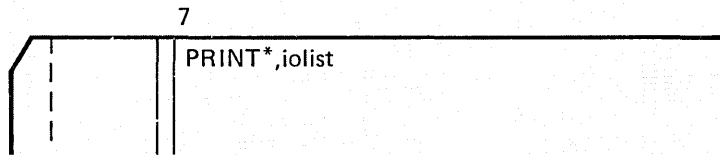
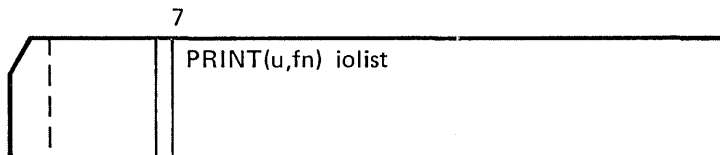
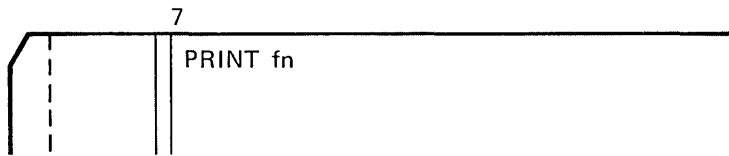
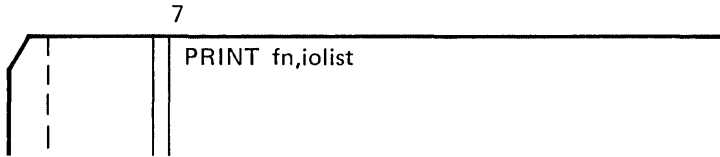
## **CARRIAGE CONTROL**

The record length of print files is limited to a maximum of 137 characters. The first character of the record is the carriage control character and is never printed. The second character of the record appears in the first print position.

The printer control characters are listed in section I-10. For off-line printing, printer control is determined by the installation printer routine.

## OUTPUT STATEMENTS

### PRINT



This statement transfers information from the storage locations named in the input/output list to the file named OUTPUT or the file specified by u, according to the specification in the format designator, fn or \*. If the user has not specified an alternate assignment, the file OUTPUT is sent to the printer.

```

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

```

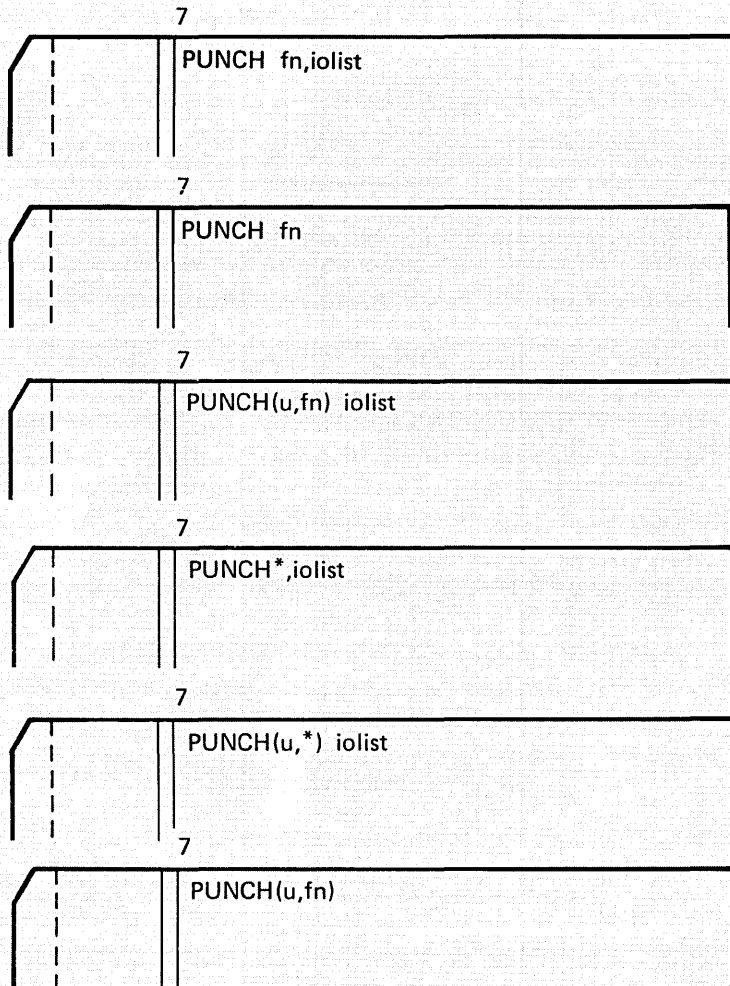
The iolist can be omitted. For example,

```

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

```

## PUNCH



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

```

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

```

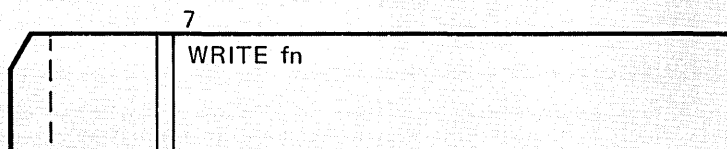
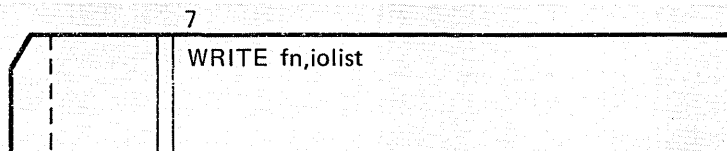
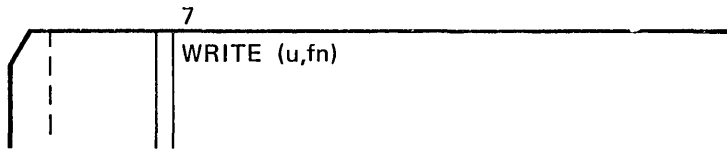
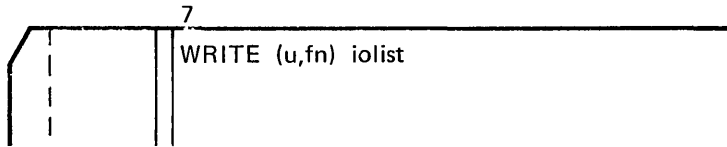
The iolist can be omitted. For example,

```

PUNCH 30
30 FORMAT (10H LAST CARD)

```

### FORMATTED WRITE



The formatted WRITE statement transfers information from the storage locations named in the input/output list to the file named OUTPUT or the file specified by u, according to the FORMAT specification, fn. If the user has not specified an alternate assignment, the file OUTPUT is sent to the printer.

```

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

```

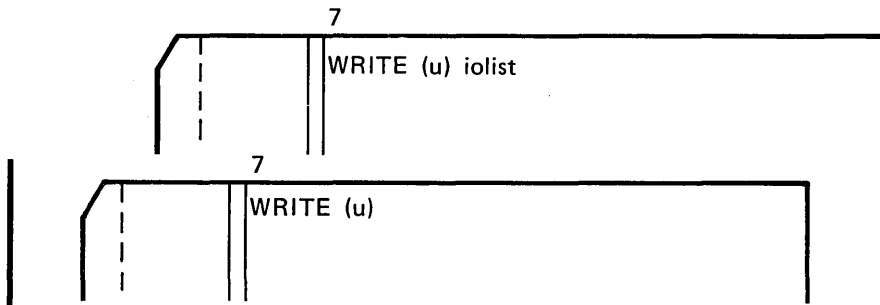
The iolist can be omitted. For example,

```

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

```

### UNFORMATTED WRITE



Example:

```

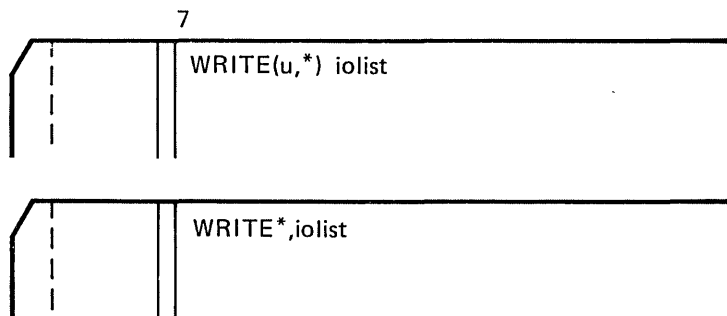
PROGRAM OUT(OUTPUT,TAPE10)
DIMENSION A(260),B(4000)
.
.
.
WRITE (10) A,B
END

```

This statement is used to output binary records. Information is transferred from the list variables, iolist, to the specified output unit, u, with no FORMAT conversion. One record is created by an unformatted WRITE statement. (Refer to section 5, part III). If the list is omitted, the statement writes a null record on the output device. A null record has no data but contains all other properties of a legitimate record.



## LIST DIRECTED WRITE



Data is transferred from storage locations specified by the iolist to unit u in a manner consistent with the list directed input described below.

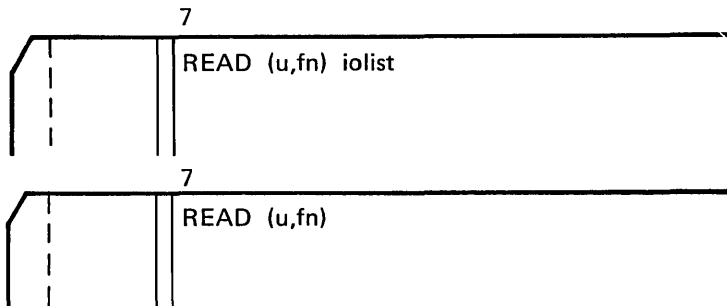
For files referenced in list directed WRITE and PRINT statements, the maximum record length in characters should be specified in the PROGRAM statement (section I-7).

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

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

## INPUT STATEMENTS

### FORMATTED READ



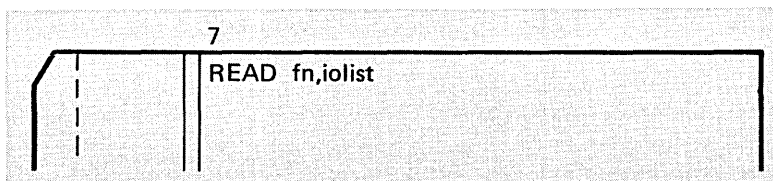
These statements transmit data from unit u to storage locations named in iolist according to FORMAT specification fn. The number of words in the list and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, one or more FORTRAN records will be bypassed. The number of records bypassed is one plus the number of slashes interpreted in the FORMAT statement. Except for information read into H specifications in the FORMAT statement, the data in the records skipped is ignored.

```

PROGRAM IN (INPUT,OUTPUT,TAPE4=INPUT,TAPE7=OUTPUT)
READ (4,200) A,B,C
200 FORMAT (3F7.3)
A = B*C+A
WRITE (7,50) A
50 FORMAT (50X,F7.4)
STOP

```

The user should test for an end-of-file after each READ statement to avoid input/output errors. If an attempt is made to read on unit u and an EOF was encountered on the previous read operation on this unit, execution terminates and an error message is printed. (Refer to section 5, part III, EOF FUNCTION.)



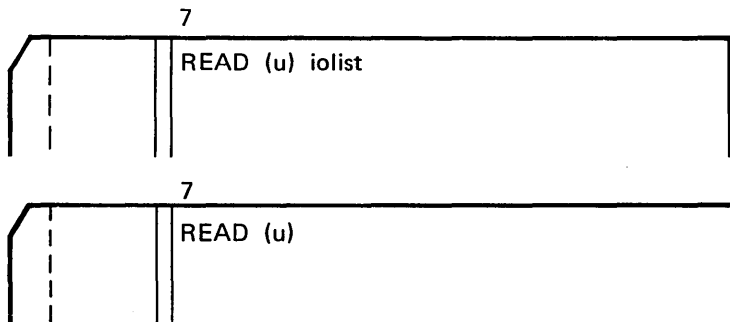
This statement transmits data from the INPUT file to the locations named in iolist. Data is converted in accordance with format specification fn.

```

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

```

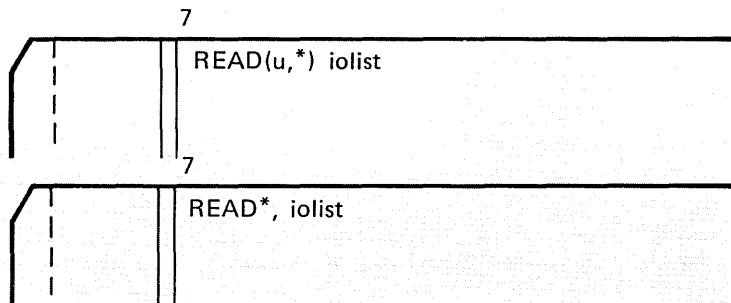
### UNFORMATTED READ



One record (section III-5) of information is transmitted from the specified unit, u, to the storage locations named in iolist. Records must be in binary form; no FORMAT statement is used. The information is transmitted from the designated file in the form in which it exists on the file. If the number of words in the list exceeds the number of words in the record, execution diagnostic results. If the number of locations specified in the iolist is less than the number of words in the logical record, the excess data is ignored. If iolist is omitted, READ (u) spaces over one record.

```
PROGRAM AREAD ( INPUT,OUTPUT,TAPE2 )
READ (2) X,Y,Z
SUM = X+Y+Z/2.
:
END
```

### LIST DIRECTED READ



Data is transmitted from unit u or the file INPUT to the storage locations named in iolist. The input data items are free-form with separators rather than in fixed-size fields.

A list directed READ following a list directed READ that terminated in the middle of a data record continues with the same data record. When a list directed READ follows a formatted READ or a formatted READ follows a list directed READ, a new data record is always used.

For files referenced in list directed READ statements, the maximum record length in characters should be specified in the PROGRAM statement (section I-7).

Example:

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

Input:

13.3, -5.2, .01

Output:

```
$OUT CAT = .133E+02, BIRD = -.5E+01, DOG = .1E-01.
$END
```

## LIST DIRECTED INPUT DATA FORMS

The list directed READ statement is similar to formatted I/O statements except an asterisk replaces the FORMAT statement number. For input statements, the form is:

```
READ *, iolist
```

```
READ(unit,*) iolist
```

Input data consists of a string of values separated by: one or more blanks, a comma or a slash either of which may be preceded or followed by any number of blanks. Also, a line boundary, such as end of record or end of card, serves as a value separator.

To repeat a value, an integer repeat constant is followed by an asterisk and the constant to be repeated. Blanks cannot be embedded in a constant or the specification of a repeated constant.

A null may be input in place of a constant when the value assigned to the corresponding list entity is not to be changed. A null is indicated by the first character in the input string being a comma or by two commas or slashes separated by an arbitrary number of blanks. Nulls may be repeated by specifying an integer repeat count followed by an asterisk and any value separator. A null cannot be used for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, remaining list elements are treated as nulls; when the next input statement is executed for this specified unit, the character following the slash becomes the first input character for the second READ. When the I/O list is exhausted and no slash has been encountered, the next list directed input on the same unit will begin at the following value separator.

Constants in the input stream take the form of FORTRAN constants except: blanks are not allowed within a constant and a decimal point omitted from a real constant is assumed to occur to the right of the right-most digit of the mantissa. Otherwise, each constant must be of the same type as the corresponding list entry, or the job will be terminated. Furthermore, a repeated constant such as 4\*7 should not be used as input data to variables of differing types.

For example:

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

can read correctly:

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

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

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

The only Hollerith constants permitted are those enclosed in the symbol  $\neq$ . They may contain embedded blanks. The paired symbols  $\neq \neq$  can be used to represent a single  $\neq$  within a character constant. A character string cannot be repeated, and it should be read into an integer variable or array. A character constant of less than 10 characters is padded on the right with blanks to fill the word. Only the first 10 characters are used if the constant exceeds 10 characters.

## LIST DIRECTED OUTPUT DATA FORM

List directed output is consistent with the input; however, null values, as well as slashes and repeated constants are not produced. For real or double precision variables with absolute values in the range of  $10^{-6}$  to  $10^9$ , an F format type of conversion is used; otherwise, an output is of the IPE type. Trailing zeros in the mantissa and leading zeros in the exponent are suppressed.

PRINT\*,list

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

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

For example:

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

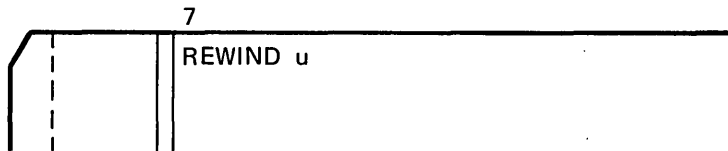
Output:

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

## FILE MANIPULATION STATEMENTS

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

### REWIND

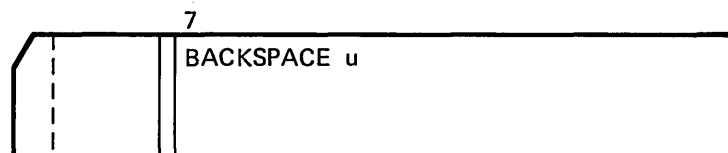


The REWIND operation positions a file so that the next FORTRAN input/output operation references the first record in the file; even though several ENDFILE statements may have been issued to that unit since the last REWIND. A mass storage file is positioned at the beginning of information. If the file is already at beginning of information, the statement acts as a do-nothing statement. (Refer to BACKSPACE/REWIND, section 5, part III for further information.)

Example:

```
REWIND 3
```

### BACKSPACE



Unit *u* is backspaced one logical record. If the file is at beginning of information, this statement acts as a do-nothing statement. A backspace operation should not follow a list directed read on a given file.

§BACKSPACE is permitted for F, S, or W record format or for tape files with one record per block. (Refer to BACKSPACE/REWIND, section 5, part III for further information.)

Example:

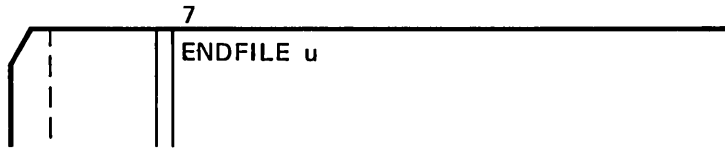
```
DO 1 LUN = 1,10,3  
1 BACKSPACE LUN
```

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

---

§ Applies only to SCOPE 2.1.

## ENDFILE



An end-of-file mark is written on the designated unit.

Issuing an ENDFILE as the first operation on a file establishes the same default record and block types as used for formatted I/O (RT=Z, BT=C).

Meaningful results are not guaranteed if ENDFILE is used on a random access file and subsequently a random file subroutine, such as READMS, is called.

Example:

```
IOUT = 6LOUTPUT
END FILE IOUT
```

End-of-file is written on the file OUTPUT.

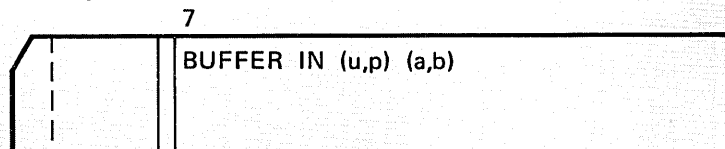
## BUFFER STATEMENTS

The buffer statements and the read/write statements both accomplish data input/output; however, they differ in the following respects:

A buffer control statement initiates data transmission and then returns control to the program so that it can perform other tasks while data transmission is in progress. A read/write statement completes data transmission before returning control to the program.

In a buffer control statement, parity must be specified by a parity indicator. In the read/write control statement, the mode of transmission formatted (display code) or unformatted (binary) is tacitly implied.

The read/write control statements are associated with a list and, if formatted, with a FORMAT statement. The buffer statements are not associated with a list; data is transmitted to or from a block of storage.



- p Integer constant or simple integer variable. Designates parity on 7-track magnetic tape, zero designates even parity; one designates odd parity. p is inoperative for other peripheral devices.
- a First word of record to be transmitted.
- b Last word of record to be transmitted.

The address of b must be greater than or equal to the address of a. In addition, a and b must be either the same variable, or else in the same array, common block, or equivalence class. If a and b are different variables or array elements in a common block without any equivalenced members, optimization may be degraded.

Example:

Given the following specification statements:

```
DIMENSION A(100), B(50), F(50)
COMMON /C/ CA, CB, CC
COMMON /D/ DD, DF
EQUIVALENCE (B,A), (CC,F(25))
```

the following statements are valid:

```
BUFFER IN (,0) (A(2), A(100))
BUFFER IN (1,0) (CA, CC)
BUFFER IN (1,0) (B, A(100))
BUFFER IN (1,0) (CA, F(50))
```

and the following are invalid:

```
BUFFER IN (1,0) (B, F(50))
BUFFER IN (1,0) (CB,DD)
```

Each BUFFER IN statement causes one record of information to be transmitted from unit u to storage locations a through b. A program should not reference either the unit u or the contents of storage locations a through b between the time a BUFFER IN statement is executed and the time a UNIT function (on the same unit) indicates the buffer operation is complete. The length of a BUFFER IN record can be ascertained through either the LENGTH function or the LENGTHX library subroutine (section 8, part I).

1	5	7
		PROGRAM TP (TAPE1,OUTPUT)
		INTEGER REC(512),RNUMB
		REWIND 1
		DO 4 RNUMB = 1,10000
1		BUFFER IN (1,1) (REC(1),REC(512))
2		IF (UNIT(1)) 3,5,5
3		K=LENGTH(1)
		C LENGTH RETURNS NUMBER OF WORDS TRANSFERRED BY BUFFER IN
4		PRINT 100,RNUMB,(REC(I),I=1,K)
100		FORMAT (7HORECORD,I5/(1X,10A10))
5		STOP
		END

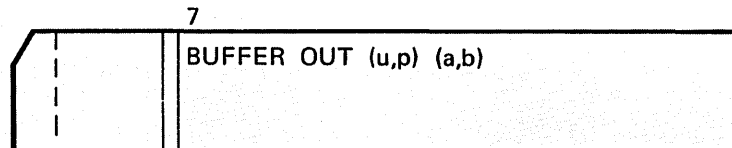


Odd parity information is transferred from logical unit 1 into storage beginning at the first word of the array, REC(1), and extending through the last word of the array, REC(512). The UNIT function tests the status of the buffer operation. If the buffer operation is completed without error, statement 3 is executed. If an EOF or a parity error is encountered, control transfers to statement 5 and the program stops.

Example:

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

Even parity information is transferred from logical unit 1 into storage beginning at the first word of the array, CALC(1), and extending through CALC(50), the last word of the array.



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

Contents of storage locations a through b are written on unit u in even or odd parity.

Examples:

```
BUFFER OUT(2,0)(OUTBUF(1),OUTBUF(4))
```

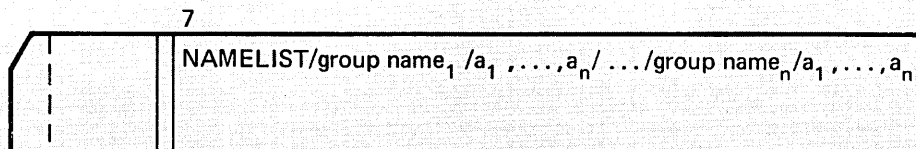
```
DIMENSION ALPHA(100)
```

```
BUFFER OUT(2,1)(ALPHA(1),ALPHA(100))
```

One record is written for each BUFFER OUT statement. Section 5, part III contains further information regarding BUFFER IN/OUT statements.

## NAMELIST

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



group name            Symbolic name which must be enclosed in slashes and must be unique within the program unit.

a<sub>1</sub>, ..., a<sub>n</sub>            List of variables or array names separated by commas.

The NAMELIST group name identifies the succeeding list of variables or array names. Whenever an input or output statement references the NAMELIST name, the complete list of associated variables or array names is read or written.

A NAMELIST group name must be declared in a NAMELIST statement before it is used in an input/output statement. The group name may be declared only once, and it may not be used for any purpose other than a NAMELIST name in the program unit. It may appear in any of the input/output statements in place of the format number:

```
READ (u, group name)  
READ group name  
WRITE (u, group name)  
PRINT group name  
PUNCH group name
```

It may not, however, be used in an ENCODE or DECODE statement in place of the format number. When a NAMELIST group name is used, the list must be omitted from the input/output statement.

A variable or array name may belong to one or more NAMELIST groups.

Data read by a single NAMELIST name READ statement must contain only names listed in the referenced NAMELIST group. A set of data items may consist of any subset of the variable names in the NAMELIST. The value of variables not included in the subset remain unchanged. Variables need not be in the order in which they appear in the defining NAMELIST statement.

```

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

```

Input record

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

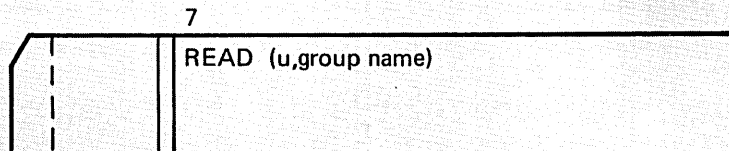
Output

```

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

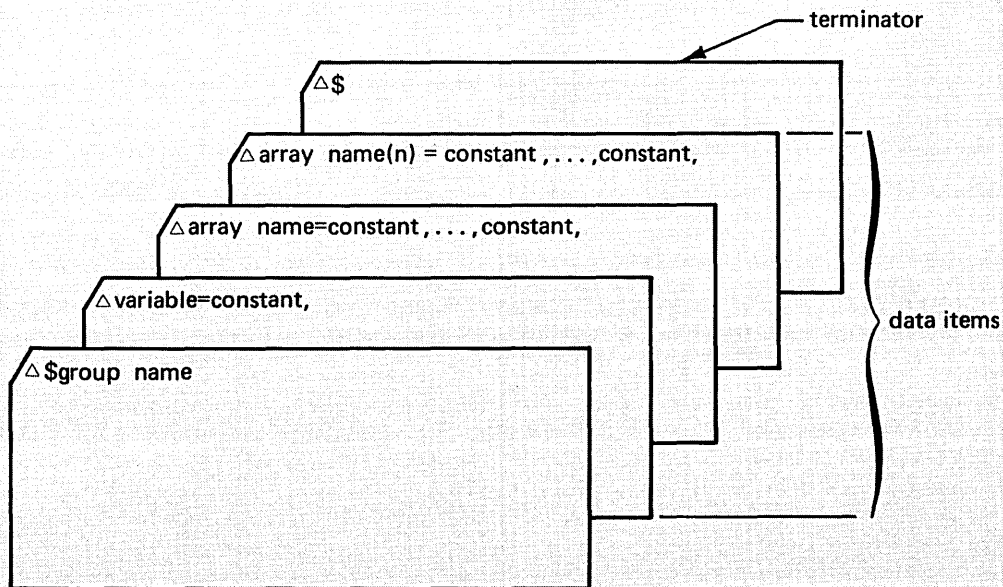
$END

```



When a READ statement references a NAMELIST group name, input data in the format described below is read from the designated file. If the specified group name is not found before end-of-file, a fatal error occurs. If the file is empty, control returns to the statement following the READ; however, a subsequent read on the same file will result in a fatal error. Consequently, a NAMELIST read should be followed by a test for end-of-file (see section III-5).

## INPUT DATA



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

Blanks must not appear:

- Between \$ and NAMELIST group name

- Within array names and variable names

Blanks may be used freely elsewhere.

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

Data items separated by commas may be in three forms:

- variable = constant

- array name = constant,...,constant

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

Omitting a constant constitutes a fatal error.

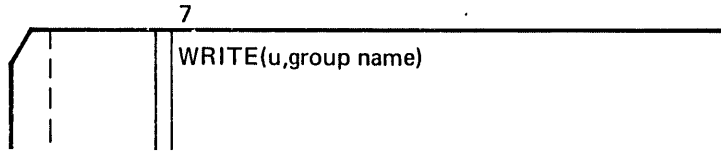
Constants can be preceded by a repetition factor and an asterisk.

Example:

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

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

## OUTPUT



All variables and arrays, and their values, in the list associated with the NAMELIST group name are output on the designated unit, u. They are output in the order of specification in the NAMELIST Statement. Output consists of at least three records. The first record is a \$ in column 2 followed by the group name; the last record is a \$ in column 2 followed by the characters END.

Example:

```
PROGRAM NAME (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
NAMELIST/VALUES/TOTAL,QUANT,COST
DATA QUANT,COST/15.,3.02/
TOTAL = QUANT*COST*1.3
WRITE (6,VALUES)
STOP
END
```

Output

```
$VALUES
TOTAL   = .588899999999999E+02,
QUANT   = .15E+02,
COST    = .302E+01,
$END
```

No data appears in column 1 of any record. If the logical unit referenced is the standard punch unit and a variable crosses column 80, this and following variables are punched on the next card. The maximum length of a record written by a WRITE (u, group name) or PRINT group name statement is 136 characters (unless a smaller maximum record length has been specified in the PROGRAM statement). Logical constants appear as T or F. Elements of an array are output in the order in which they are stored.

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

Example:

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

Output record:

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

This output may be read later in the same program using the following statement:

```
READ(5, ITEMS)
```

## ARRAYS IN NAMELIST

In input data the number of constants, including repetitions, given for an array name should not exceed the number of elements in the array.

Example:

```
INTEGER BAT(10)
NAMELIST/HAT/BAT, DOT
READ (5, HAT)
```

```
  2
┌───┴─── $HAT      BAT=2,3,8*4, DOT=1.05$END
└───┬───
```

The value of DOT becomes 1.05, the array BAT is as follows:

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

Example:

```
DIMENSION GAY(5)
NAMELIST/DAY/GAY, BAY, RAY
READ (5, DAY)
```

Input Record:

```
  2
┌───┴─── $DAY GAY(3)=7.2, GAY(5)=3.0, BAY=2.3, RAY=77.2$
└───┬───
```

array element = constant.....constant

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants need not equal, but may not exceed, the remaining number of elements in the array.

Example:

```
DIMENSION ALPHA (6)
NAMELIST/BETA/ALPHA,DELTA,X,Y
READ (5,BETA)
```

Input record:

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

In storage

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

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

Example:

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

Input Record:

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

Example:

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

Input record:

```
$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,  
12.OE1,13,4*14,Z=(1.,2.),K=16,M=17$
```

produce the following values:

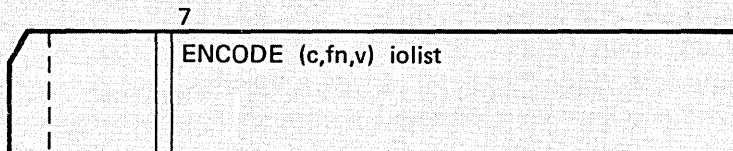
```
I1=1                Y(1,2)=14.0  
I2=2                Y(2,2)=14.0  
I3=3                Y(3,2)=14.0  
Y(3,5)=26.0        Y(1,3)=14.0  
Y(1,1)=11.0        K=16  
Y(2,1)=120.0       M=17  
Y(3,1)=13.0        Z=(1.,2.)    The rest of Y is unchanged.  
L=.TRUE.
```

## ENCODE AND DECODE

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

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

### ENCODE



- v Variable or array name which supplies the starting location of the record to be encoded.
- c Unsigned integer constant or simple integer variable specifying the length of each record.

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

- fn Format designator, statement label or integer variable, which must not be a NAME-LIST group name or an \*
- iolist List of variables to be transmitted to the location specified by v.



Example:

```
5 | 7
PROGRAM ENCDE (OUTPUT)
DIMENSION A(2),ALPHA(4)
DATA A,B,C/10HABCDEFGH IJ,10HKLMNOPQRST,5HUVWXY,7HZ123456/
ENCODE (40,1,ALPHA)A,B,C
1 FORMAT (2A4,A5,A6)
PRINT 2,ALPHA
2 FORMAT (20H1CONTENTS OF ALPHA =,8A10)
STOP
END
```

In memory after ENCODE statement has been executed.

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

ENCODE is a core-to-core transfer of data, which is similar to a formatted WRITE. Data in the iolist, in internal form, is converted under FORMAT specifications, fn, and written in display code into an array or variable.

An integral number of words is allocated for each record created by an ENCODE statement. If c is not a multiple of 10, the record ends before the end of the word is reached; and the remainder of the word is blank filled.

If the list and the format specification transmit more than the number of characters specified per record, an execution error message is printed. If the number of characters transmitted is less than the length specified by c, remaining characters in the record are blank filled.

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

```

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

```

Output:

CONTENTS OF ALPHA =ABCDKLMN

UVWXYZ12345

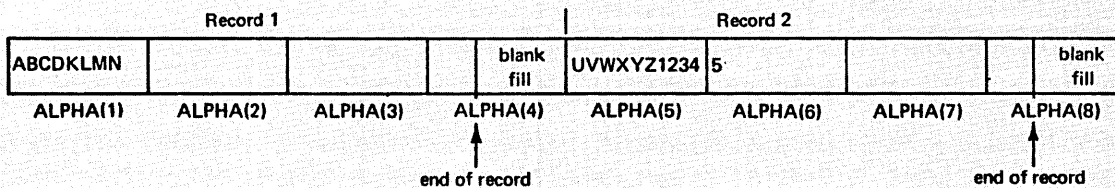
If this same ENCODE statement is altered to:

```

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

```

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



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

The following example illustrates that it is possible to encode an area into itself, and the information previously contained in the area will be destroyed.

```

5 | 7
PROGRAM ENCO2 (OUTPUT)
I=10HBCDEFGHIJK
IA=1H1
ENCODE (8,10,I) I,IA,1
10 FORMAT (A3,A1,R4)
PRINT 11,I
11 FORMAT (A11)
END

```

Printout is:

BCD1HIJKbb

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

```

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

```

M is tested to ensure it is within limits; if it is not, control goes to statement 2, which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters (2A10,I ). This packed FORMAT is stored in SPECMAT. SPECMAT contains (2A10,Im).

A and B will be printed under specification A10, and the quantity J under specification I2, through I9 according to the value of m.

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

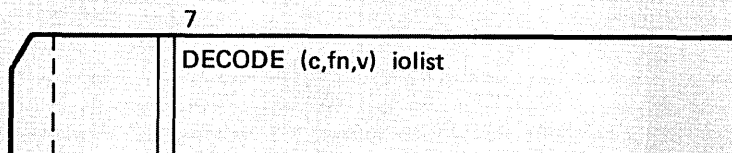
```

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

```

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

## DECODE



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

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

```

5 | 7
PROGRAM ADD (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
DIMENSION CARD (8), INK (77)
2 READ (5,100) KEY1,CARD
100 FORMAT (I1,7A10,A9)
IF (EOF(5)) 80,90
90 IF (KEY1-2) 3,8,3
3 CALL ERROR1
GO TO 2
8 WRITE (6,300) CARD
300 FORMAT (I1,7A10,A7///)
DECODE (77,17,CARD) INK
17 FORMAT (77I1)
ITOT = 0
DO 4 I = 1,77
4 ITOT = ITOT + INK(I)
ISAVE = ITOT
WRITE (6,200) ISAVE
200 FORMAT (19X,*TOTAL OF 77 SCORES ON CARD = *,I10)
80 STOP
END
SUBROUTINE ERROR1
WRITE (6,1)
1 FORMAT (5X,*NUMBER IS NOT 2*)
RETURN
END

```

(An explanation of this program appears in part II.)

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

Starting at the named location, *v*, data is transmitted according to the specified format and stored in the list variables. If the number of characters per record is not a multiple of 10 (a display code word contains 10 display code characters) the balance of the word is ignored. However, if the number of characters specified by the list and the format specification exceeds the number of characters per record, an execution error message is printed. DECODE processing an illegal BCD character for a given conversion specification produces a FATAL error. If DECODE is processing an A or R FORMAT specification and encounters a zero character (6 bits of binary zero), the character is treated as a colon under 64-character set or as a blank under 63-character set.

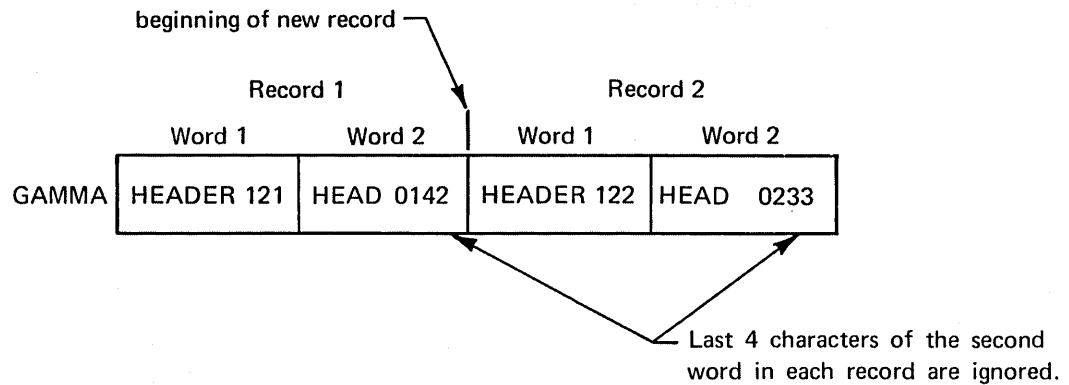
Example:

$c \neq$  multiple of 10

```

DECODE (16,1,GAMMA) X,B,C,D
1 FORMAT (2A8)

```



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

```
X=HEADER 1
B=21HEAD
C=HEADER 1
D=22HEAD
```

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

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

To form SSSSDDDDD in storage location NAME:

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

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

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

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



## INPUT/OUTPUT LISTS AND FORMAT STATEMENTS I-10

---

This chapter covers input/output lists and FORMAT statements. Input/output statements, which include READ and WRITE, are covered in section I-9.

### INPUT/OUTPUT LISTS

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

If no list appears on input, a record is skipped. Only Hollerith information from the FORMAT statement can be output with a null (empty) output list.

A list consists of a variable name, an array name, an array element name, or an implied DO list. On output the data list can include Hollerith constants and arithmetic expressions. Such expressions must not reference a function if such reference would cause any input/output operations (including DEBUG output) to be executed or would cause the value of any element of the output statement to be changed.

Multiple lists may appear, separated by commas, each of which may be enclosed in parentheses, such as: (...),(...).

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

Subscripts in an input/output list may be any valid subscript (section I-2).

Examples:

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

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

A DO-implied specification takes one of the following forms:

$i = m_1, m_2, m_3$

$i = m_1, m_2$

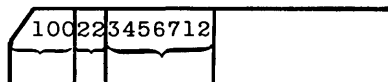
The elements  $i$ ,  $m_1$ ,  $m_2$ , and  $m_3$  have the same meaning as in the DO statement. The range of a DO-implied specification is that of the DO-implied list. The values of  $i$ ,  $m_1$ ,  $m_2$ , and  $m_3$  must not be changed within the range of the DO implied list by a READ statement.

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

Example:

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

Input record



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

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

## IMPLIED DO IN I/O LIST

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

Example:

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

has the same effect as the statement

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

The general form for an implied DO loop is:

$$(\dots((list, i_1=m_1, m_2, m_3), i_2=j_1, j_2, j_3), \dots, i_n=k_1, k_2, k_3)$$

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

$i_1 \dots i_n$  are integer control variables. A control variable should not be used twice in the same implied DO nest, but array names, array elements, and variables may appear more than once.



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

The general form for an array is:

```
(( (A(I, J, K), i1=m1, m2, m3), i2=n1, n2, n3), i3=k1, k2, k3)
```

Example:

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

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

Examples of simple implied DO loop list items:

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

The following DO loop would have the same effect:

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

Example:

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

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

Implied DO loops may be nested.

Example:

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

Equivalent to the following:

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

The list is similar to the nest of DO loops:

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

Example:

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

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

Example:

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

Data is transmitted in the following sequence:

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

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

Examples:

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

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

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

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

```

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

```

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

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

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

Example:

```

      DIMENSION B (10,15)
      READ 13,B

```

is equivalent to

```

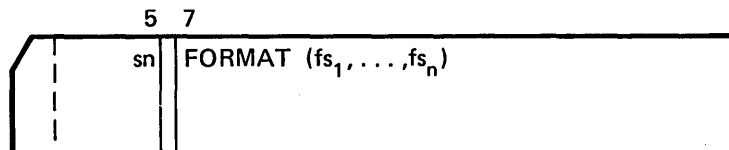
      READ 13, ((B(I,J),I=1,10),J=1,15)

```

The entire array B will be transmitted in both cases.

## FORMAT STATEMENT

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



sn                      Statement label which must appear

fs<sub>1</sub>,...,fs<sub>n</sub>            Set of one or more field specifications separated by commas and/or slashes and optionally grouped by parentheses

Example:

```

      READ (5,100) INK,NAME,AREA
100 FORMAT (10X,I4,I2,F7.2)

```

FORMAT is a non-executable statement which specifies the format of data to be moved between input/output device and main memory. It is used in conjunction with read and write statements, and it may appear anywhere in the program.

The FORMAT specification is enclosed in parentheses. Blanks are not significant except in Hollerith field specifications.

Generally, each item in an input/output list is associated with a corresponding field specification in a FORMAT statement. The FORMAT statement specifies the external format of the data, and the type of conversion to be used, and defines the length of the FORTRAN record or records. COMPLEX variables always correspond to two field specifications. DOUBLE variables correspond to one floating point field specification (D, E, F, G) or two of any other kind. The D field specification will correspond to exactly one list item or half of a COMPLEX item.

The type of conversion should correspond to the type of the variable in the input/output list. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete.

For example:

```
INTEGER N
READ (5,100) N
100 FORMAT (F10.2)
```

A floating point number is assigned to the variable N which could cause unpredictable results if N is referenced later as an integer.

## DATA CONVERSION

The following types of data conversions are available:

srEw.d	Single precision floating point with exponent
srEw.dEe	With explicitly specified exponent length
srEw.dDe	With explicitly specified exponent length
srFw.d	Single precision floating point without exponent
srGw.d	Single precision floating point with or without exponent
srDw.d	Double precision floating point with exponent
rIw	Decimal integer conversion
rIw.z	With minimum number of digits specified
rLw	Logical conversion
rAw	Character conversion
rRw	Character conversion
rOw	Octal integer conversion
rOw.z	With minimum number of digits specified
rZw	Hexadecimal conversion
srVw.d	Variable type conversion

E, F, G, D, I, L, A, R, O, and Z are the codes which indicate the type of conversion.

- w Non-zero, unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading blanks, + or - signs, decimal point, and exponent.
- d Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded.
- e Non-zero, unsigned integer constant specifying the number of digits in the exponent.
- r Non-zero, unsigned integer constant less than  $2^{17}-1$  specifying the number of times the conversion code is to be repeated.
- s Optional scale factor.
- z Unsigned integer constant specifying the minimum number of digits to be output.

The field width  $w$  must be specified for all conversion codes. If  $d$  is not specified for  $w.d$ , it is assumed to be zero.  $w$  must be  $\geq d$ .

## FIELD SEPARATORS

Field separators are used to separate specifications and groups of specifications. The format field separators are the slash (/) and the comma. The slash is also used to specify demarcation of formatted records.

## CONVERSION SPECIFICATION

Leading blanks are not significant in numeric input conversions; other blanks are treated as zeros. Plus signs can be omitted. An all-blank field is considered to be minus zero, except for logical input, where an all-blank field is considered to be FALSE. When an all-blank field is read with a Hollerith input specification, each blank character is translated into a display code 55 octal.

For the E, F, G, and D input conversions, a decimal point in the input field overrides the decimal point specification of the field descriptor.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field. The number of characters produced by an output conversion must not be greater than the field width. If the field width is exceeded, asterisks are inserted throughout the field.

Complex data items are converted on input/output as two independent floating point quantities. The format specification uses two conversion elements.

Example:

```
COMPLEX A,B,C,D
PRINT 10,A
10 FORMAT (F7.2,E8.2)
READ 11,B,C,D
11 FORMAT (2E10.3,2(F8.3,F4.1))
```

Data of differing types may be read by the same FORMAT statement. For example:

```
10 FORMAT (I5,F15.2)
```

specifies two numbers, the first of type integer, the second of type real.

```
READ (5,15) NO,NONE,INK,A,B,R
15 FORMAT (3I5,2F7.2,A4)
```

reads 3 integer variables

reads 2 real variables

reads 1 character variable

### lw and lw.z INPUT

The I conversion is used to input decimal integer constants.

lw lw.z

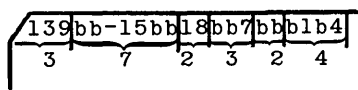
w is a decimal integer constant designating the total number of characters in the field including signs and blanks. z is ignored on input.

The plus sign may be omitted for positive integers. When a sign appears, it must precede the first digit in the field. Blanks are interpreted as zeros. An all blank field is considered to be minus zero. Decimal points are not permitted. The value is stored in the specified variable. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will terminate execution.

Example:

```
READ 10,I,J,K,L,M,N
10 FORMAT (I3,I7,I2,I3,I2,I4)
```

Input Card:



In storage:

I contains 139                      L contains 7  
 J contains -1500                  M contains -0  
 K contains 18                      N contains 104

### lw and lw.z OUTPUT

The I specification is used to output decimal integer values.

lw lw.z

w is a decimal integer constant designating the total number of characters in the field including signs and blanks. If the integer is positive the plus sign is suppressed. Numbers in the range of  $-2^{59} + 1$  to  $2^{59} - 1$  ( $2^{59} - 1 = 576\ 460\ 752\ 303\ 423\ 487$ ) are output correctly.

z is a decimal integer constant designating the minimum number of digits output. Leading zeros are generated when the output value requires less than z digits. If z=0, a zero value will produce all blanks. If z=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. Not specifying z produces the same results as z=1.

The specification Iw or Iw.z outputs a number in the following format:

ba...a  
 b            Minus sign if the number is negative, or blank if the number is positive  
 a...a        May be a maximum of 18 digits

The output quantity is right justified with blanks on the left.

If the field is too short, all asterisks occupy the field.

Example:

```

PRINT 10,I,J,K           I contains -3762
                          J contains +4762937
10 FORMAT (I9,I10,I5.3)  K contains +13

Result:
                          bbb-3762|bbb4762937|bb013|
                          8      10      5
                          ↑
                          1st blank taken as
                          printer control character
  
```

Example:

```

WRITE (6,100)N,M,I      N contains +20
                          M contains -731450
100 FORMAT (I5,I6,I9)   I contains +205

Result:
                          bb20|*****|bbbbbb205|
                          4      6      9
                          ↑
                          specification too
                          small;* indicates field
                          is too short

1st blank taken
as printer control
character
  
```

### Ew.d, Ew.dEe and Ew.dDe OUTPUT

E specifies conversion between an internal real value and an external number written with exponent.

Ew.d    Ew.dEe    Ew.dDe

w is an unsigned integer designating the total number of characters in the field. w must be wide enough to contain digits, plus or minus signs, decimal point, E, the exponent, and blanks. Generally,  $w \geq d + 6$  or  $w \geq d + e + 4$  for negative numbers and  $w \geq d + 5$  or  $w \geq d + e + 3$  for positive numbers. Positive numbers need not reserve a space for the sign of the number. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

d specifies the number of digits to the right of the decimal within the field.

e specifies the number of digits in the exponent.

The Ew.d specification produces output in the following formats:

b.a...aE ± ee                      For values where the magnitude of the exponent is less than one hundred

b.a...a ± ee                      For values where the magnitude of the exponent exceeds one hundred

b is a minus sign if the number is negative, and a blank if the number is positive

a...a is the most significant digits of the value correctly rounded

When the specification Ew.dEe or Ew.dDe is used, the exponent is denoted by E or D and the number of digits used for the exponent field not counting the letter and sign is determined by e. If e is specified too small for the value being output, the entire field width as specified by w will be filled with asterisks.

Examples:

```
PRINT 10,A                                      A contains -67.32 or +67.32
10 FORMAT (E10.3)
```

Result:                                      -.673E+02 or b.673E+02

```
PRINT 10,A
10 FORMAT (E13.3)
```

Result:                                      bbb-.673E+02 or bbbb.673E+02

If an integer variable is output under the Ew.d specification, results are unpredictable since the internal format of real and integer values differ. An integer value does not have an exponent and will be printed, therefore, as a very small value or 0.0.



**Ew.d, Ew.dEe and Ew.dDe INPUT**

E specifies conversion between an external number written with an exponent and an internal real value.

Ew.d    Ew.dEe    Ew.dDe

w is an unsigned integer designating the total number of characters in the field, including plus or minus signs, digits, decimal point, E and exponent. If an external decimal point is not provided, d acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is:

$$(\text{integer subfield}) \times 10^{-d} \times 10^{(\text{exponent subfield})}$$

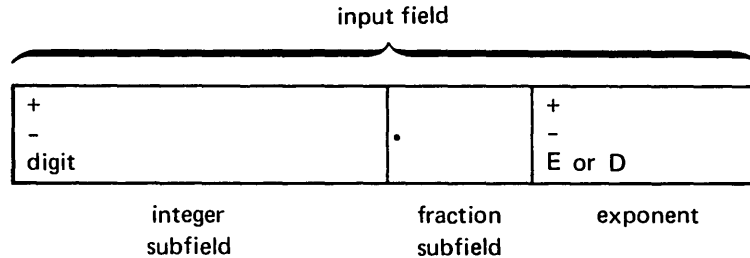
For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as:  $3267 \times 10^{-8} \times 10^5 = 3.267$ .

If an external decimal point is provided, it overrides d. If d does not appear it is assumed to be zero. e, if specified, has no effect on input.

In the input data, leading blanks are not significant; other blanks are interpreted as zeros.

An input field consisting entirely of blanks is interpreted as minus zero.

The following diagram illustrates the structure of the input field:

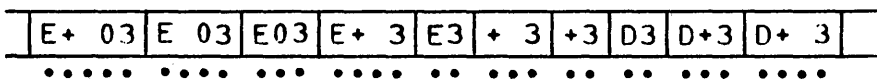


The integer subfield begins with a + or - sign, a digit, or a blank; and it may contain a string of digits. The integer field is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, - or the end of the input field. It may contain a string of digits.

The exponent subfield may begin with E, + or -. When it begins with E, the + is optional between E and the string of digits in the subfield.

For example, the following are valid equivalent forms for the exponent 3:



The range, in absolute value, of permissible values is 3.13152E-294 to 1.26501E322 approximately. Smaller numbers will be treated as zero; larger numbers will cause a fatal error message.

Valid subfield combinations:

+1.6327E-04	Integer-fraction-exponent
-32.7216	integer-fraction
+328+5	integer-exponent
.629E-1	fraction-exponent
+136	integer only
136	integer only
.07628431	fraction only
E-06 (interpreted as zero)	exponent only

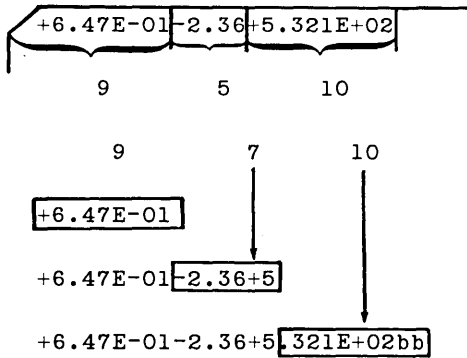
If the field length specified by w in Ew.d is not the same as the length of the field containing the input number, incorrect numbers may be read, converted, and stored. The following example illustrates a situation where numbers are read incorrectly, converted and stored; yet there is no immediate indication that an error has occurred:

```

READ 20,A,B,C
20 FORMAT (E9.3,E7.2,E10.3)

```

On the card, input quantities are in three adjacent fields, columns 1-24:



First, +647E-01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number -2.36+5 is read instead of -2.36. The specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input number. Since the second specification incorrectly took two digits from the third number, the specification for the third number is now incorrect. The number .321E+02bb is read. Trailing blanks are treated as zeros; therefore the number .321E+0200 is read converted and placed in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Examples of Ew.d input specifications:

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	.14326	All subfields present
-12.437629E+1	E13.6	-124.37629	All subfields present
327.625	E7.3	327.625	No exponent subfield
4.376	E5	4.376	No d in specification
-.0003627+5	E11.7	-36.27	Integer subfield left of decimal contains only a minus sign and a plus sign appears instead of E in input field
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only
blanks	Ew.d	-0.	All subfields empty
1E1	E3.0	10.	No fraction subfield; input number converted as 1.x10
E+06	E10.6	0.	No integer or fraction subfield; zero stored regardless of exponent field contents
1.bEb1	E6.3	10.	Blanks are interpreted as zeros
1.0E13	E6.3	10000000000000.	

#### Fw.d OUTPUT

The F specification outputs a real number without a decimal exponent.

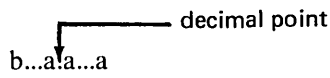
#### Fw.d

w is an unsigned integer which designates the total number of characters in the field including the sign (if negative) and decimal point. w must be  $\geq d + 2$ .

d specifies the number of places to the right of the decimal point. When d is zero, only the digits to the left of the decimal and the decimal point are printed.

The plus sign is suppressed for positive numbers. If the field is too short, all asterisks appear in the output field. If the field is longer than required, the number is right justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

The specification Fw.d outputs a number in the following format:



b Minus sign if the number is negative, or blank if the number is positive.

Examples:

Value of A	FORMAT Statement	PRINT Statement	Printed Result
+32.694	10 FORMAT (1H ,F6.3)	PRINT 10,A	32.694
+32.694	11 FORMAT (1H ,F10.3)	PRINT 11,A	bbbb32.694
-32.694	12 FORMAT. (1H ,F6.3)	PRINT 12,A	*****
.32694	13 FORMAT (1H ,F4.3,F6.3)	PRINT 13,A,A	.327bb.327

The specification 1H is the carriage control character.

#### Fw.d INPUT

On input F specification is treated identically to the E specification.

Examples of the F format specification:

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field
-4.7366	F7	-4.7366	No d in specification
.62543	F6.5	.62543	No integer subfield
.62543	F6.2	.62543	Decimal point overrides d of specification
+144.15E-03	F11.2	.14415	Exponents are allowed in F input, and may have P scaling
5bbbb	F5.2	500.00	No fraction subfield; input number converted as $50000 \times 10^{-2}$
bbbbbb	F5.2	-0.00	Blanks in input field interpreted as -0

### Gw.d INPUT

Input under control of G specification is the same as for the E specification. The rules which apply to the E specification apply to the G specification.

Gw.d

w Unsigned integer which designates the total number of characters in the field including E, digits, sign, and decimal point

d Number of places to the right of the decimal point

Example:

```
READ (5,11) A,B,C
11 FORMAT (G13.6,2G12.4)
```

### Gw.d OUTPUT

Output under control of the G specification is dependent on the size of the floating point number being converted. The number is output under the F conversion unless the magnitude of the data exceeds the range which permits effective use of the F. In this case, it is output under E conversion with an exponent.

Gw.d

w Unsigned integer which designates the total number of characters in the field including digits, signs and decimal point, the exponent E, and any leading blanks.

d Number of significant digits output.

If a number is output under the G specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field  $E \pm 00$ ). Therefore, for output under G conversion w must be greater than or equal to  $d + 6$ . The six extra spaces are required for sign and decimal point plus four spaces for the exponent field.

Example:

```
PRINT 200,YES          YES contains 77.132
200 FORMAT (G10.3)
```

Output: b77.1bbbb b denotes a blank

If the decimal point is not within the first d significant digits of the number, the exponential form is used (G is treated as if it were E).

Example:

```
PRINT 100, EXIT  
100 FORMAT (G10.3)
```

EXIT contains 1214635.1

Output: .121E+07

Example:

```
READ (5,50) SAMPLE  
.  
.  
.  
WRITE (6,20) SAMPLE  
20 FORMAT (1X,G17.8)
```

Data read by READ statement	Data Output	Format Option
.1415926535bE-10	.14159265E-10	E conversion
.8979323846	.89793238	F conversion
2643383279.	.26433833E+10	E conversion
-693.9937510	-693.99375	F conversion

#### Dw.d OUTPUT

Dw.d

Type D conversion is used to output double precision variables. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed.

Examples of type D output:

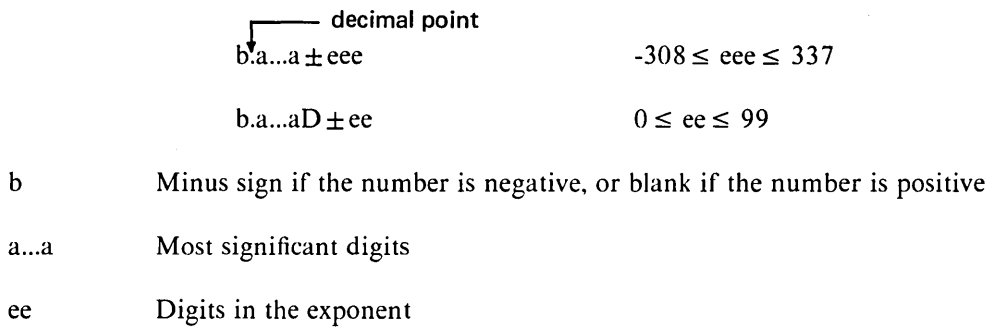
```

DOUBLE A,B,C
A = 111111.11111
B = 222222.22222
C = A + B
PRINT 10,A,B,C
10 FORMAT (3D23.11)

.111111111111D+06   .22222222222D+06   .33333333333D+06

```

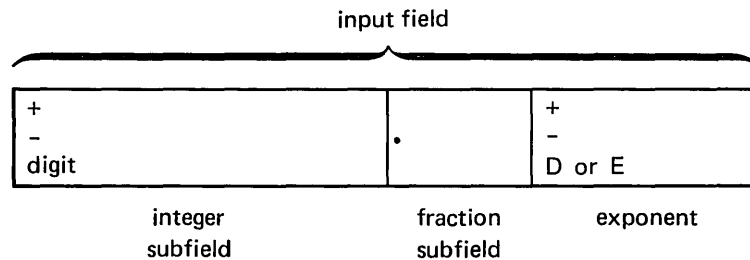
The specification Dw.d produces output in the following format:



**Dw.d INPUT**

D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield.

The following diagram illustrates the structure of the input field:



**Ow INPUT**

Octal values are converted under the O specification.

Ow

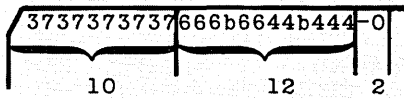
w is an unsigned integer designating the total number of characters in the field. The input field may contain a maximum of 20 octal digits. Blanks are allowed and a plus or minus sign may precede the first octal digit. Blanks are interpreted as zeros and an all blank field is interpreted as minus zero. A decimal point is not allowed.

The list item corresponding to the Ow specification should be integer.

Example:

```
INTEGER P,Q,R
READ 10,P,Q,R
10 FORMAT (010,012,02)
```

Input Card:



Input storage (octal representation):

P	00000000003737373737
Q	00000000666066440444
R	77777777777777777777

#### Ow OUTPUT

The O specification is used to output the internal representation in octal.

Ow    Ow.d

w is an unsigned integer designating the total number of characters in the field. If w is less than 20, the rightmost digits are output. For example, if the contents of location P were output with the following statement the digit 3737 would be output.

```
WRITE (6,1) P                    location P 0000000003737373737
100 FORMAT (1X,04)
```

If w is greater than 20, the 20 octal digits (20 octal digits = a 60-bit word) are right justified with blanks on the left.

For example, if the contents of location P are output with the following statement

```
WRITE (6,200) P
200 FORMAT (1X,022)
```

Output would appear as follows:

bb0000000003737373737                    b = blank

A negative number is output in one's complement internal form.

If d is specified, the number is printed with leading zero suppression and with a minus sign for negative numbers. At least d digits will be printed. If the number cannot be output in w octal digits, all asterisks will fill the field.



Example:

```
I = -11  
WRITE (6,200) I
```

Output would appear as follows:

```
bb77777777777777777764
```

The specification *Ow* produces a string of up to 20 octal digits. Two octal specifications must be used for variables whose type is complex or double precision.

### Zw INPUT and OUTPUT

Hexadecimal values are converted under the *Z* specification.

*Zw*

*w* is an unsigned integer designating the total number of characters in the field. The input field may contain digits and the letters A through F. A maximum of 15 hexadecimal digits is allowed, blanks and a plus or minus sign may precede the first hexadecimal digit. On output if *w* is greater than 15, leading blanks will occur.

### Aw INPUT

The *A* specification is used to input character data

*Aw*

*w* is an unsigned integer designating the total number of characters in the field.

Character information is stored as 6-bit display code characters, 10 characters per 60-bit word. For example, the digit 4 when read under *A* specification is stored as a display code 37. If *w* is less than 10, the input quantity is stored left justified in the word; the remainder of the word is filled with blanks.

Example:

```
READ (5,100) A  
100 FORMAT (A7)
```

Input record:

```
┌EXAMPLE
```

When EXAMPLE is read it is stored left justified in the 10 character word

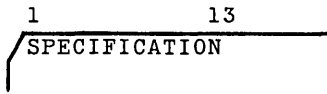
```
1234567890  
EXAMPLE
```

If *w* is greater than 10, the rightmost 10 characters are stored and remaining characters are ignored.

Example:

```
READ (5,200) B  
200 FORMAT (A13)
```

Input record:

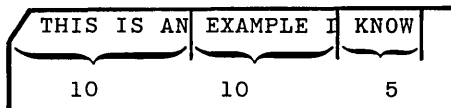


In storage:

```
12345678910
[C I F I C A T I O N ]

READ (5,10) Q,P,R
10 FORMAT (A10,A10,A5)
```

Input record:



In storage:

```
12345678910

Q [ T H I S   I S   A N ]
P [ E X A M P L E   I ]
R [ K N O W   _ _ _ ]
```

### Aw OUTPUT

The A specification is used to output alphanumeric characters.

Aw

w is an unsigned integer designating the total number of characters in the field. If w is less than 10, the leftmost characters in the word are printed. For example, if the contents of location A in the Aw input example are output with the following statements:

```
WRITE (6,300)A
300 FORMAT (1X,A4)
```

In storage:

```
A [ E X A M P L E _ _ ]
```

Characters EXAM are output

If w is greater than 10, the characters are output right-justified in the field, with blanks on the left. For example, if A in the previous example is output with the following statements:

```
WRITE (6,400)A
400 FORMAT (1X,A12)
```

Output is as follows:

```
bbEXAMPLEbbb          b = blank
```

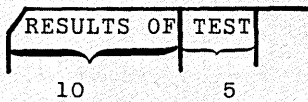
**Rw INPUT**

w is an unsigned integer designating the total number of characters in the field. The R specification is the same as the A specification unless w is less than 10. If w is less than 10, the input characters are stored right-justified, with binary zero fill on the left.

Example:

```
READ (5,600) H00,RAY
600 FORMAT (R10,R5)
```

Input card:



In storage:

```
H00 [RESULTSbOF]
RAY [0...00bTEST]          b = blank
```

**Rw OUTPUT**

Rw

w is an unsigned integer designating the total number of characters in the field.

This specification is the same as the A specification unless w is less than 10. If w is less than 10, the right-most characters are output. For example, if RAY from the previous example is output with the following statements:

```
WRITE (6,700) RAY
700 FORMAT (1X,R3)          Characters EST are output.
```

## Lw INPUT

The L specification is used to input logical variables.

Lw

w is an unsigned integer designating the total number of characters in the field.

If the first non-blank character in the field is T, the logical value `.TRUE.` is stored in the corresponding list item, which should be of type logical. If the first non-blank character is F, the value `.FALSE.` is stored. If the first non-blank character is not T or F, a diagnostic is printed. An all blank field has the value `.FALSE.`

## Lw OUTPUT

Lw

w is an unsigned integer designating the total number of characters in the field.

Variables output under the L specification should be of type logical. A value of `.TRUE.` or `.FALSE.` in storage is output as a right justified T or F with blanks on the left.

Example:

```
LOGICAL I,J,K      I contains -0
PRINT 5,I,J,K      J contains 0
5 FORMAT (3L3)     K contains -0
```

Output:

```
bTbbFbbT
```

## SCALE FACTORS

The scale factor P is used to change the position of a decimal point of a real number when it is input or output. Scale factors may precede D, E, F and G format specifications.

```
nPDw.d      nPDw.dEe      nPDw.dDe
nPEw.d      nPEw.dEe      nPEw.dDe
nPFw.d
nPGw.d.
nP
```

n is the scale factor which can be any integer constant. w is an unsigned integer constant designating the total width of the field. d determines the number of digits to the right of the decimal point.

A scale factor of zero is established when each format control statement is first referenced; it holds for all F, E, G, and D field descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G specifications in that FORMAT statement until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G specifications, a zero scale factor, 0P must precede a specification.

Example:

```
15 FORMAT(2PE14.3,F10.2,G16.2,0P4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specification. The 0P scale factor restores normal scaling ( $10^0 = 1$ ) for the subsequent specification 4F13.2.

A scaling factor may appear independently of a D, E, F or G specification. It holds for all subsequent D, E, F or G specifications within the same FORMAT statement, until changed by another scaling factor.

Example:

```
FORMAT(3P,5X,E12.6,F10.3,0PD18.7,-1P,F5.2)
```

E12.6 and F10.3 specifications are scaled by  $10^3$ , the D18.7 specification is not scaled, and the F5.2 specification is scaled by  $10^{-1}$ .

The specification (3P,3I9,F10.2) is the same as the specification (3I9,3PF10.2).

## Fw.d SCALING

### INPUT

The number in the input field is divided by  $10^n$  and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is  $314.1592 \times 10^{-2} = 3.141592$ . However, if an exponent is read the scale factor is ignored.

### OUTPUT

The number in the output field is the internal number multiplied by  $10^n$ . In the output representation, the decimal point is fixed; the number moves to the left or right, depending on whether the scale factor is plus or minus. For example, the internal number -3.1415926536 may be represented on output under scaled F specifications as follows:

```
.....
(-1PF13. 6)    -.314159
(   F13. 6)    -3.141593
(  1PF13. 6)   -31.415927
(  3PF13. 6)  -3141.592654
.....
```

## Ew.d AND Dw.d SCALING

### INPUT

Ew.d scaling on input is the same as Fw.d scaling on input.

### OUTPUT

The effect of the scale factor nP is to shift the output coefficient left n places and reduce the exponent by n. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent such that: if  $n \leq 0$ , there will be exactly -n leading zeros and d + n significant digits after the decimal point; if  $n > 0$ , there will be exactly n significant digits to the left of the decimal point and d - n + 1 significant digits to the right of the decimal point. For example, the number -3.1415926536 is represented on output under the indicated Ew.d scaling as follows:

```
.....  
(-3PE20. 4)          -.0003E+04  
(-1PE20. 4)          -.0314E+02  
(  E20. 4)           -.3142E+01  
( 1PE20. 4)          -3.1416E+00  
( 3PE20. 4)          -314.16E-02  
.....
```

## Gw.d SCALING

### INPUT

Gw.d scaling on input is the same as Fw.d scaling on input.

### OUTPUT

The effect of the scale factor is nullified unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number  $N < 10^{d-1}$  or  $N \geq 10^d$ ). In these cases, the scale factor has the same effect as described above for Ew.d and Dw.d scaling. For example, the numbers -3.1415926536 and -.00031415926536 are represented on output under the indicated Gw.d scaling as follows:

```
.....  
(-3PG20. 6)          -3.14159          (-3PG20. 6)          -.000314E+00  
(-1PG20. 6)          -3.14159          (-1PG20. 6)          -.031416E-02  
(  G20. 6)           -3.14159          (  G20. 6)           -.314159E-03  
( 1PG20. 6)          -3.14159          ( 1PG20. 6)          -3.141593E-04  
( 3PG20. 6)          -3.14159          ( 3PG20. 6)          -314.1593E-06  
( 5PG20. 6)          -3.14159          ( 5PG20. 6)          -31415.93E-08  
.....  
.....  
( 7PG20. 6)          -3.14159  
.....
```

## X SPECIFICATION

The X specification is used to skip characters in an input line or output line. On output, any character positions not previously filled during this record generation will be set to blank. It is not associated with a variable in the input/output list.

- nX Number of characters, n, to be skipped. An optional plus sign may precede n.
  - 0X is ignored, X is interpreted as 1X. The comma following X in the specification list is optional.
- nX Back up n characters, will not back up beyond the first column.

Example:

```

WRITE (6,100) A,B,C
100 FORMAT (F9.4,4X,F7.5,4X,I3)
A = -342.743
B = 1.53190
C = 22

```

Output record:

```

-342.743bbbb1.53190bbbb22      b is a blank

```

on input n columns are skipped.

Example:

```

READ 11,R,S,T
11 FORMAT (F5.2, 3X, F5.2, 6X, F5.2)

```

or

```

11 FORMAT (F5.2, 3XF5.2, 6XF5.2)

```

Input card:

```

14.62bb$13.78bCOSTb15.97

```

In storage:

```

R 14.62
S 13.78
T 15.97

```

Example:

```

INTEGER A
PRINT 10,A,B,C
10 FORMAT (I2,6X,F6.2,6X,E12.5)
A contains 7
B contains 13.6
C contains 1462.37

```

```

Result:          7bbbbbbb13.60bbbbbbb.146237E+04

```

## nH OUTPUT

The H specification is used to output strings of alphanumeric characters and like X, H is not associated with a variable in the input/output list.

nH

n            Number of characters in the string including blanks.

H            Denotes a Hollerith field. **The comma following the H specification is optional.**

For example, the statement:

```
WRITE (6,1)
1 FORMAT (15HbENDbOFbPROGRAM)
```

can be used to output the following on the output listing.

```
END OF PROGRAM
```

Examples:

Source program:

```
PRINT 20
20 FORMAT (28HbBLANKSbCOUNTbINbANbHbFIELD.)
```

produces output record:

```
BLANKSbCOUNTbINbANbHbFIELD.
```

Source program:

```
PRINT 30,A                            A contains 1.5
30 FORMAT (6HbLMAX=,F5.2)
```

produces output record:

```
LMAX=b1.50
```

## nH INPUT

The H specification can be used to read Hollerith characters into an existing H field within the FORMAT statement.

Example:

Source program:

```
READ 10
10 FORMAT (27Hbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb)
```



Input card:

```
bTHIS IS A VARIABLE HEADING
```

After READ, the FORMAT statement labeled 10 contains the alphanumeric information read from the input card; a subsequent reference to statement 10 in an output statement acts as follows:

```
PRINT 10
```

produces the print line:

```
THIS IS A VARIABLE HEADING
```

```
*... * ≠...≠
```

Character strings delimited by a pair of \* or ≠ symbols can be used as alternate forms of the H specification for output. The paired symbols delineate the Hollerith field. This specification need not be separated from other specifications by commas. If the Hollerith field is empty, or invalidly delimited a fatal execution error occurs, and an error message is printed.

An asterisk cannot be output using the specification \* \*. For example,

```
PRINT 1  
1 FORMAT (*ABC*DE*)
```

The second \* in the FORMAT statement causes the specification to be interpreted as \*ABC\* and DE\*, which is not valid.

The H specification or ≠...≠ could be used to output this correctly:

```
PRINT 1  
1 FORMAT (7H ABC*DE)
```

Output appears as follows: ABC\*DE

```
PRINT 2  
2 FORMAT (≠ ABC*DE≠)
```

Output appears as follows: ABC\*DE

≠ can be represented within ≠...≠ by two consecutive ≠ symbols.

Example:

```
PRINT 3  
3 FORMAT (≠ DON≠≠T≠)
```

Output examples:

```
PRINT 10
10 FORMAT (* SUBTOTALS*)
```

produces the following output:

```
SUBTOTALS
```

```
WRITE (6,20)
20 FORMAT (≠bRESULT OF CALCULATIONS IS AS FOLLOWS≠)
```

produces the following output:

```
RESULT OF CALCULATIONS IS AS FOLLOWS
```

```
PRINT 1, ≠SQRT≠, SQRT(4.)
1 FORMAT (A10,E10.2)
```

produces the following output:

```
SQRT      2.0
```

Note: ≠ is output as ' on some printers.

The \*...\* or ≠...≠ specification can be used to read alphanumeric data; however, the effect differs depending on whether \*...\* or ≠...≠ occurs in an actual FORMAT statement or in a format specification contained in a variable or array. When the READ statement contains a constant specifying a FORMAT statement, alphanumeric characters are read into the \*...\* or ≠...≠ specification. When a name occurs in the READ statement to specify the format information (variable format), characters in the input stream are skipped and no change is made in the \*...\* or ≠...≠ specification.

In FORMAT statements, the \*...\* or ≠...≠ specification is changed to nH... at compile time. This conversion does not occur with variable format specifications.

## FORTRAN RECORD SLASH

The slash indicates the end of a FORTRAN record anywhere in the FORMAT specification. Where a slash is used to separate field specification elements, a comma is allowed but not required. Consecutive slashes can be used and need not be separated from other elements by commas. When a slash is the last format specification to be processed, it causes a blank record to be written on output or an input record to be skipped. Normally, the slash indicates the end of a record during output and specifies that further data comes from the next record during input.

Example:

```
PRINT 10
10 FORMAT (6X, 7HHEADING///3X, 5HINPUT, 8H OUTPUT)
```

Printout:

```
HEADING _____ line 1
          _____ (blank) _____ line 2
          _____ (blank) _____ line 3
INPUT OUTPUT _____ line 4
```

Each line corresponds to a formatted record. The second and third records are blank and produce the line spacing illustrated.

Example:

```
I=5
J=6
K=7
PRINT 1,I,J,K
1  FORMAT (3I5/F10.4)
PRINT 2
2  FORMAT(* A BLANK LINE SHOULD PRECEDE THIS LINE*)
```

Printout:

```
5    6    7

A BLANK LINE SHOULD PRECEDE THIS LINE
```

The variable list (I, J, K) is exhausted and processing continues until a variable conversion is encountered (F10.4). Since the slash has been processed, it causes a blank line to be printed, and F10.4 is ignored because there is nothing to be converted.

Example:

```
DIMENSION B(3)
READ (5,100)IA,B
100 FORMAT (I5/3E7.2)
```

These statements read two records; the first contains an integer number, and the second contains three real numbers.

```
PRINT 11,A,B,C,D
11 FORMAT (2E10.2/2F7.3)
```

In storage:

```
A -11.6
B .325
C 46.327
D -14.261
```

Output:

```
b-.12E+02bbb.33E+00
46.327-14.261
```

```
PRINT 11,A,B,C,D
11 FORMAT (2E10.2//2F7.3)
```

Output:

```
b-.12E+02bbb.33E+00 _____ line 1
_____ (blank) _____ line 2
46.327-14.261 _____ line 3
```

The second slash causes the blank line.

## REPEATED FORMAT SPECIFICATION

FORMAT specifications can be repeated by prefixing the control characters D, E, F, G, I, A, R, L, Z, and O with a non-zero, unsigned integer constant specifying the number of repetitions required.

100 FORMAT (3I4,2E7.3) is equivalent to: 100 FORMAT (I4,I4,I4,E7.3,E7.3)

50 FORMAT (4G12.6) is equivalent to: 50 FORMAT (G12.6,G12.6,G12.6,G12.6)

A group of specifications can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor.

1 FORMAT (I3,2(E15.3,F6.1,2I4))

is equivalent to the following specification if the number of items in the input/output list does not exceed the format conversion codes:

1 FORMAT (I3,E15.3,F6.1,I4,I4,E15.3,F6.1,I4,I4)

A maximum of nine levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If the number of items in the input/output list is fewer than the number of format codes in the FORMAT statement, excess FORMAT codes are ignored.

If the number of items in the input/output list exceeds the number of format conversion codes when the final right parenthesis in the FORMAT statement is reached, the line formed internally is output. The FORMAT control then scans to the left looking for a right parenthesis within the FORMAT statement. If none is found, the scan stops when it reaches the beginning of the FORMAT specification. If a right parenthesis is found, however, the scan continues to the left until it reaches the field separator which precedes the left parenthesis pairing the right parenthesis. Output resumes with the FORMAT control moving right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered.

A repetition factor can be used to indicate multiple slashes, n(/), where n is an unsigned integer constant indicating the number of slashes required and n-1 is the number of lines skipped on output.

Example:

```
PRINT 15, (A(I),I=1,9)
15 FORMAT (8HbRESULTS4(/),(3F8.2))
```

Format statement 15 is equivalent to:

```
15 FORMAT (8HbRESULTS//// (3F8.2))
```

Printout:

```

RESULTS _____ line 1
                _____ (blank) _____ line 2
                _____ (blank) _____ line 3
                _____ (blank) _____ line 4
    3.62  -4.03  -9.78 _____ line 5
   -6.33   7.12   3.49 _____ line 6
    6.21  -6.74  -1.18 _____ line 7
  
```

Example:

```

READ (5,300)I,J,E,K,F,L,M,G,N,R
300 FORMAT (I3,2(I4,F7.3),I7)
  
```

is equivalent to storing data in I with format I3, J with I4, E with F7.3, K with I4, F with F7.3, and L with I7. A new record is then read; data is stored in M with the format I4, G with F7.3, N with I4, and R with F7.3.

```

READ (5,100) NEXT, DAY, KAT, WAY, NAT, RAY, MAT
100 FORMAT (I7,(F12.7,I3))
  
```

NEXT is input with format I7, DAY is input with F12.7, KAT is input with I3. The FORMAT statement is exhausted (the right parenthesis has been reached), a new card is read, and the statement is rescanned from the group (F12.7,I3). WAY is input with the format F12.7, NAT with I3, and from a third card, RAY with F12.7, and MAT with I3.

## PRINTER CONTROL CHARACTER

The first character of a printer output record is used for carriage control and is not printed. It appears in all other forms of output as data.

The printer control characters are as follows:

Character	Action
Blank	Space vertically one line then print
0	Space vertically two lines then print
1	Eject to the first line of the next page before printing
+	No advance before printing; allows overprinting
Any other character	Refer to the operating system reference manual

For output directed to the card punch or any device other than the line printer, control characters are not required. If carriage control characters are transmitted to the card punch, they are punched in column one.

Carriage control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash. Carriage control characters can be generated by any means.

Examples:

```
FORMAT (1H0,F7.3,I2,G12.6)
```

```
FORMAT (1H1,I5,*RESULT = *,F8.4)
```

```
FORMAT (*1*,I4,2(F7.3))
```

```
FORMAT (1X,I4,G16.8)
```

Example:

```
PROGRAM CHARCON (OUTPUT)
PRINT 10
10 FORMAT (1H1, 5X, *HERE WE ARE AT THE TOP OF A NEW PAGE*)
PRINT 20
20 FORMAT (3(/))
DO 30 I = 2,8
IF (I .EQ. 4 .OR. I .EQ. 6) 40,50
50 PRINT 60
60 FORMAT (21X, # X X # / 1H+, 20X, # == #)
GO TO 30
40 PRINT 70
70 FORMAT (20X, # XXXXXXXXXXXX # / 1H+, 19X, # ===== #)
30 CONTINUE
PRINT 80
80 FORMAT (1H0, 5X, #BEGIN TIC TAC TOE #)
STOP
END
```

Output

HERE WE ARE AT THE TOP OF A NEW PAGE

```
  X  X
  X  X
XXXXXXXXXXXXXXXXX
  X  X
XXXXXXXXXXXXXXXXX
  X  X
  X  X
```

BEGIN TIC TAC TOE

### Tn SPECIFICATION

This specification is a column selection control.

Tn

n            Unsigned integer. If n = zero, column 1 is assumed.

When Tn is used, control skips columns right or left until column n is reached; then the next format specification is processed. Using card input, if n > 80 the column pointer is moved to column n, but a succeeding specification would read only blanks.

```
      READ 40, A, B, C
40    FORMAT (T1, F5.2, T11, F6.1, T21, F5.2)
```

Input:

```
84.73bbbb2436.2bbbb89.14.
```

A is set to 84.73, B to 2436.2, and C to 89.14.

```
      WRITE (31, 10)
10    FORMAT (T20,*LABELS*)
```

The first 19 characters of the output record are skipped and the next six characters, LABELS, are written on output unit number 31 beginning in character position 20.

With T specification, the order of a list need not be the same as the printed page or card input, and the same information can be read more than once.

When a T specification causes control to pass over character positions on output, positions not previously filled during this record generation are set to blanks; those already filled are left unchanged.



Example:

```
5 7
PROGRAM TEST (OUTPUT)
1 FORMAT (12(10H0123456789))
  PRINT 1
  PRINT 60
60 X FORMAT (T80,*COMMENTS*,T60,*HEADING4*,T40,
  *HEADING3*,T20,*HEADING2*,T2,*HEADING1*)
  PRINT 10
10 FORMAT (20X*THIS IS THE END OF THIS RUN*T52*...HONEST*)
  PRINT 1
  STOP
  END
```

```
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
HEADING1          HEADING2          HEADING3          HEADING4          COMMENTS
THIS IS THE END OF THIS RUN  ...HONEST
123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
```

Since the first character in a line output to the printer is used for printer control, T2 is output in the first print position.

The following example shows that it is possible to destroy a previously formed field inadvertently. The specification T5 destroys part of the Hollerith specification 10H DISASTERS.

```
1 FORMAT (10H DISASTERS, T5, 3H123)
  PRINT 1
```

produces the following output:

```
DIS123ERS
```

### V SPECIFICATION

When V is encountered in a FORMAT statement, the rightmost 6 bits from the current variable in the input/output list are interpreted as display code for a character to be used in place of the V as the conversion specification for the next variable in the input/output list. V can be used as a dummy specification for the following conversions: A, D, E, F, G, I, L, O, P, R, T, X and Z. It cannot be used as the E or D explicitly specifying exponent length, as in Ew.dVe.

Example:

```
PROGRAM V (OUTPUT)
INTEGER AFORMAT, RFORMAT
AFORMAT = 1RA
RFORMAT = 1RR
NUM = 10H0123456789
PRINT 10, AFORMAT, NUM
10 FORMAT (T8, *FORMAT SPECIFICATION TAKEN FROM VARIABLE AFORMAT; NUM
- OUTPUTS AS *, V5 /)
PRINT 20, RFORMAT, NUM
20 FORMAT (T8, *FORMAT SPECIFICATION TAKEN FROM VARIABLE RFORMAT; NUM
- OUTPUTS AS *, V5)
STOP
END
```

Output:

```
FORMAT SPECIFICATION TAKEN FROM VARIABLE AFORMAT; NUM OUTPUTS AS 01234
FORMAT SPECIFICATION TAKEN FROM VARIABLE RFORMAT; NUM OUTPUTS AS 56789
```

## EQUALS SIGN

When = is encountered in a FORMAT statement, the current variable in the input/output list supplies a positive integer value to be used in place of the = in the conversion specification for the next variable in the input/output list. The = can be used in place of a number anywhere within a FORMAT statement. Such use of = precludes compilation syntax checking of the FORMAT statement. V and = can be combined in one conversion specification.

Example:

```
PROGRAM EQUALS (OUTPUT)
INTEGER W(10)
DATA W/1,2,3,4,5,6,7,8,9,10/
NUM = 10H0123456789
DO 10 I = 1,10
10 PRINT 20, W(I), NUM
20 FORMAT (T30, A=)
STOP
END
```

Output:

```
0
01
012
0123
01234
012345
0123456
01234567
012345678
0123456789
```

A variable must exist in the I/O list for each time an = or V is processed in the format statement.

Example:

```
DIMENSION A(5),B(5)
I3 = 3
PRINT 1,I3,A,I3,B
1 FORMAT(1X,5F10.=)
```

Two lines of five values each are printed; however, I3 must be repeated in the I/O list or the first value of B is used to replace the =.

Example:

```
PROGRAM VEQUALS (OUTPUT)
INTEGER FORMAT(2), W(10)
DATA FORMAT/1RA, 1RR/, W/1,2,3,4,5,6,7,8,9,10/
NUM = 10H0123456789
DO 10 I = 1,2
DO 10 J = 1,10
K = J
IF (I .EQ. 2) K = 11-J
10 PRINT 20, FORMAT(I), W(K), NUM
20 FORMAT (T20, V=)
STOP
END
```

```

Output:  0
         01
         012
         0123
         01234
         012345
         0123456
         01234567
         012345678
         0123456789
         0123456789
         123456789
         23456789
         3456789
         456789
         56789
         6789
         789
         89
         9

```

## EXECUTION TIME FORMAT STATEMENTS

Variable FORMAT statements can be read in as part of the data at execution time and used by READ, WRITE, PRINT, PUNCH, ENCODE, or DECODE statements later in the program. The format is read in as alphanumeric text under the A specification and stored in an array or a simple variable, or it may be included in a DATA statement. The format must consist of a list of format specifications enclosed in parentheses, but without the word FORMAT or the statement label.

For example, a data card could consist of the characters:

```

      (E7.2,G20.5,F7.4,I3)

```

The name of the array containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The array name, which appears with or without subscripts, specifies the location or the first word of the FORMAT information.

For example, assume the following FORMAT specifications:

```

      (E12.2,F8.2,I7,2E20.3,F9.3,I4)

```

This information on an input card can be read by the statements of the program such as:

```

      DIMENSION IVAR(3)
      READ 1, IVAR
      1 FORMAT (3A10)

```

The elements of the input card are placed in storage as follows:

```
IVAR(1)          (E12.2,F8.  
IVAR(2)          2,I7,2E20.  
IVAR(3)          3,F9.3,I4)
```

A subsequent output statement in the same program can refer to these FORMAT specifications as:

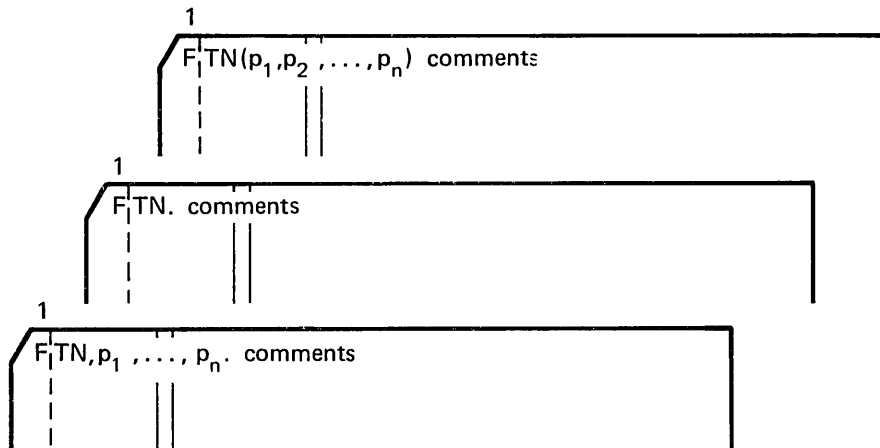
```
PRINT IVAR,A, B, I, C, D, E, J
```

Produces exactly the same result as the program.

```
PRINT 10, A, B, I, C, D, E, J  
10 FORMAT (E12.2,F8.2,I7,2E20.3,F9.3,I4)
```



The FORTRAN Extended compiler is called from the library and executed by an FTN control card. The FTN control card calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This control card may be in any of the following forms:



Example:

```
FTN (A,L,R,GO,S=0)
```

## PARAMETERS

The optional parameters,  $p_1, \dots, p_n$  must be separated by commas and may be in any order within the parentheses. If no parameters are specified, FTN is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the syntax for job control statements as defined in the operating system reference manual, with the added restriction that a comma is the only valid parameter delimiter. Card columns following the right parenthesis or period can be used for comments; they are ignored by the compiler, but are printed on the dayfile.

Default values are used for omitted parameters. These defaults are set when the system is installed; since installations can change default values, the user should determine what default values are in effect at the user's particular installation.

Unrecognizable parameters are ignored. Conflicting options either are resolved or cause compilation to terminate, depending on the severity of the conflict; this resolution is indicated in a dayfile entry.

The values of the A, B, D, G, I, L, ML, P, S, and X parameters are passed to COMPASS when intermixed COMPASS subprograms are present.

In the following description of the FTN control card parameters, lfn indicates a file name consisting of 1 letter followed by 0-6 letters or digits.

**A EXIT PARAMETER** (Default: A = 0)

A System searches the control card record for an EXIT card at end of compilation if fatal errors have been found. If such a card is not present, the job terminates.

A = 0 System advances to the next control card at end of compilation if fatal errors have been found.

**B BINARY OBJECT FILE** (Default: B = LGO)

B Generated binary object code is output on file LGO.

B = lfn Generated binary object code is output on file lfn.

B = 0 No binary object file is produced. Cannot be specified with Go.

The B option conflicts with the Q and E options.

**BL BURSTABLE LISTING** (Default: BL = 0)

BL Generates output listing that is easily separable into components by issuing page ejects between source code, error summary (if present), cross reference map, and object code (if requested); and ensures that each program unit listing contains an even number of pages (page parity) issuing a blank page at the end if necessary.

BL = 0 Generates listings in compact format.

**C COMPASS ASSEMBLY** (Default: C = 0)

C Selects the COMPASS assembler to process the symbolic object code generated by FTN. When the C parameter is specified, FTNMAC is selected for the system texts for the COMPASS assembly; therefore, if the C option is selected, the maximum number of system texts that can be specified with the G and S parameters is six.

C = 0 Selects the FTN internal assembler (regardless of installation default), which is two to three times faster than the COMPASS assembler.

The C option conflicts with the TS, Q, and E options.



## D DEBUGGING MODE PARAMETER

(Default: D = 0)

D = lfn This option must be specified if the debug utility described in section I-13 is to be used. lfn is the name of the file where the user debug deck resides (see figure 13-4, section I-13). Binary object code is generated on the file indicated by the B parameter regardless of compilation errors or the exit parameter A. Interspersed COMPASS code, if present, is assembled under the COMPASS D option. Specifying D automatically activates OPT=0 and the T option; thus, FTN(D) is equivalent to FTN(D,OPT=0,T).

D Implies D = INPUT

D = 0 Debug statements are ignored.

OPT=1, or OPT=2, is ignored if D or D=lfn is specified. The D option conflicts with the TS option.

## E EDITING PARAMETER

(Default: E = 0)

E = lfn Generated object code is output as COMPASS line images on the file lfn, which is rewound at the end of compilation. Each program unit is prefaced with the line image, \*DECK,program, so that the file will be suitably formatted for input to UPDATE or MODIFY. Binary object code is not produced; and COMPASS is not called. When the file lfn is assembled subsequently, S=FTNMAC must be specified on the COMPASS control card.

E Implies E = COMPS

E = 0 Object file is generated in normal binary code rather than as COMPASS line images.

The E option conflicts with the B, C, GO, OL, TS, and Q options.

## EL ERROR LEVEL

(Default: EL = I)

EL = A Lists diagnostics indicating all non-ANSI usages, as well as fatal diagnostics; lists informative diagnostics if compiling under OPT = 0, 1, or 2; lists note and warning diagnostics if compiling in TS mode.

EL = I Lists informative and fatal diagnostics if compiling under OPT = 0, 1, or 2; lists note, warning, and fatal diagnostics if compiling in TS mode.

EL = N Lists note, warning, and fatal diagnostics if compiling in TS mode; lists fatal diagnostics if compiling under OPT = 0, 1, or 2.

EL = W Lists warning and fatal diagnostics if compiling in TS mode; lists fatal diagnostics if compiling under OPT = 0, 1, or 2.

EL = F Lists fatal diagnostics.

**G GET SYSTEM TEXT FILE** (Default: G = 0)

G = lfn Loads the first system text overlay from the sequential binary file, lfn.

G = lfn/ovl Searches the sequential binary file, lfn, for a system text overlay with the name ovl and loads the first such overlay encountered.

G Implies G = SYSTEXT

G = 0 Prevents system text loading from sequential binary file.

A maximum of seven system texts can be specified by any combination of the G, S, and C parameters.

This feature is for COMPASS subprograms only.

**GO AUTOMATIC EXECUTION (LOAD AND GO)** (Default: GO = 0)

GO Binary object file is loaded and executed at end of compilation.

GO = 0 Binary object file is not loaded and executed.

The GO option conflicts with the Q, E, and B = 0 options.

**I SOURCE INPUT FILE** (Default: I = INPUT)

I = lfn Source code to be compiled appears on file lfn. Compilation ends when an end-of-section, end-of-partition, or end-of-information is encountered.

I Implies I = COMPILE

**L LIST OUTPUT FILE** (Default: L = OUTPUT)

L = lfn Listable output (specified by list control options BL, EL, OL, R, and SL) is to be written onto file lfn. If list control options are not specified, the listing consists of the source program, informative and fatal diagnostics, and a short reference map.

L Implies L = OUTPUT

L = 0 Fatal diagnostics and the statement that caused them are listed on the file OUTPUT. All other compile-time output, including intermixed COMPASS, is suppressed. List control options are ignored.

**LCM LEVEL 2 AND LEVEL 3 STORAGE ACCESS†**

(Default: LCM = D)

- LCM = D            Direct mode: selects 17-bit address mode for level 2§ or 3 data. This method produces more efficient code for accessing data assigned to level 2 or 3. User LCM or ECS field length must not exceed 131,071 words.
- LCM = I            Indirect mode: selects 21-bit address mode for level 2§ or 3 data. This mode depends heavily upon indirect addressing. LCM = I must be specified if the execution LCM or ECS field length exceeds 131,071 words.
- LCM                Implies LCM = D

In TS mode, all LCM addressing is done in 21-bit mode, regardless of the LCM parameter.

**ML MODLEVEL**

(Default: ML)

- ML = nnn            Specifies nnn as the value of the MODLEVEL micro used by COMPASS. nnn consists of 1 to 7 letters or digits.
- ML                 Uses current date in the form yyddd (where yy is the year and ddd is the number of day within the year) for the MODLEVEL micro.

**OL OBJECT LIST**

(Default: OL = 0)

- OL                 Generated object code is listed on the list output file.
- OL = 0             Object code is not listed.

The OL option conflicts with the Q and E options.

**OPT OPTIMIZATION PARAMETER (see section III-14)**

(Default: OPT = 1)

- OPT = 0            Fast compilation (automatically activates T option).
- OPT = 1            Standard compilation and execution.
- OPT = 2            Fast execution.
- OPT                Implies OPT = 2

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

† See LEVEL statement, section I-6, for further information.

**P PAGINATION** (Default: P = 0)

P Page numbering of output listing is continuous from subprogram to subprogram, including intermixed COMPASS output.

P = 0 Page numbers begin at 1 for each subprogram.

**PL PRINT LIMIT** (Default: PL = 5000)

PL = n n is the maximum number of records that can be written at execution-time on the file OUTPUT. n must not exceed 999 999 999. If n is suffixed with the letter B, it is interpreted as an octal number and must not exceed 77 777 777.

The PL parameter is operative only when appearing on an FTN card used to compile a main program.

The print limit (specified at compilation-time either explicitly or by default) can be overridden at execution-time by a parameter of the same format appearing on the LGO or EXECUTE control card; see Execution Time Options, section III-3.

**Q PROGRAM VERIFICATION** (Default: Q = 0)

Q Quick mode: compiler performs full syntactic scan of the program, but no object code is produced. No code addresses are provided if a reference map is requested. This mode is substantially faster than a normal compilation; but it must not be selected if the program is to be executed.

Q = 0 Normal compilation.

The Q option conflicts with the B, C, GO, OL, TS, and E options.

**R SYMBOLIC REFERENCE MAP** (see section III-1) (Default: R = 1)

R = 0 No map

R = 1 Short map (symbols, addresses, properties, DO loop map)

R = 2 Long map (short map plus references by line number)

R = 3 Long map plus listing of common block members and equivalence classes

R Implies R = 2

In TS mode, R = 3 is identical to R = 2; common and equivalence classes are not listed (see section III-15).

## ROUND    ROUNDED ARITHMETIC COMPUTATIONS

(Default: ROUND = 0)

ROUND = op            op is any combination of the arithmetic operators + - \* / Single precision real and complex floating point arithmetic operations are performed using the hardware rounding feature, as described in the various Computer Systems Reference Manuals.

ROUND = 0            Computation for the indicated operators is not rounded.

ROUND                Implies ROUND = +-\* /

The ROUND option controls only the in-line object code compiled for arithmetic expressions; it does not affect computations of library subprograms or input/output routines.

## S    SYSTEM TEXT (LIBRARY) FILE

( Default: S = SYSTEXT  
          if G parameter = 0  
          S = 0  
          if G parameter is  
          other than G = 0 )

S = ovl                System text overlay, ovl, is loaded from the job's current library set.

S = lib/ovl            System text overlay, ovl, is loaded from the user library file or system library, lib. (Valid only if the operating system supports partitioned library sets.)

S = 0                 System text file is not loaded when COMPASS is called to assemble any intermixed COMPASS programs.

S                     Implies S = SYSTEXT

This feature is for COMPASS subprograms only.

## SEQ    SEQUENCED INPUT

(Default: SEQ = 0)

SEQ                  Source input file is in sequenced line format (see section III-15).

SEQ = 0              Source input file is in standard FORTRAN format.

Specifying the SEQ option automatically activates the TS option; sequenced line format is not recognized by the optimizing compiler or COMPASS. The SEQ option conflicts with the OPT = 0, 1, or 2 options.

## SL    SOURCE LIST

(Default: SL)

SL                    Source program is listed on the file specified by the L parameter.

SL = 0                Source program is not listed.

## **SYSEEDIT SYSTEM EDITING**

(Default: SYSEEDIT = 0)

- SYSEEDIT** All input/output references are accomplished indirectly through a table search at object time. File names are not entry points in the main program, and subprograms do not produce external references to the file name.
- SYSEEDIT = 0** Input/output references are accomplished directly; file names are used as entry points in the main program, and subprograms produce external references to the file name.

This feature is used primarily for system-resident programs.

## **T ERROR TRACEBACK**

(Default: T = 0)

- T** Full error traceback occurs when an error is detected. Calls to basic external functions are made with call-by-name sequence (see section III-10).
- T = 0** No traceback occurs when an error is detected. Calls to basic external functions are made with the more efficient call-by-value sequence. A saving in memory space and execution time is realized.

This option is provided to assist in debugging programs. Selecting the D parameter or OPT=0 automatically activates the T option.

## **TS TIMESHARING MODE**

(Default: OPT = 1)

- TS** In TS mode, compilation speed and field length are optimized at the expense of execution speed and field length. TS mode is preferable to the optimizing compilation modes (OPT = 0, 1, or 2) for the debugging stages of a program. (For more information about TS mode, see sections III-2, III-14, and III-15.) Specifying option TS together with option C, D, E, or Q constitutes a fatal control card error. If TS is specified, any OPT parameters are ignored.

## **UO UNSAFE OPTIMIZATION**

(Default UO = 0)

- UO** Allows the compiler to perform certain optimizations which are potentially unsafe. UO is ignored unless OPT = 2 is also specified. Section III-14 contains further details.
- UO = 0** Unsafe optimization is not performed.

The installation cannot select UO as a default option.

## X EXTERNAL TEXT NAME

(Default: X = OLDPL)

X = lfn                    File lfn is source of external text (XTEXT) when location field of XTEXT pseudo instruction is blank. Only one X parameter may be specified.

X                            Impies X = OPL.

This feature is for COMPASS subprograms only.

## Z ZERO PARAMETER

(Default: Z = 0)

Z                            All subroutine calls having no parameters are forced to pass a parameter list consisting of a zero word. This feature is useful to COMPASS-coded subroutines expecting a variable number of parameters. Z should not be specified unless necessary, since programs require less memory if Z is omitted.

Z = 0                        The zero word parameter list is not passed.

## FTN CONTROL CARD SAMPLES

Example:

```
FTN (A,EL=F,GO,L=SEE,R=2,S=0,SL=0)
```

Selects the following options:

A	Branch to an EXIT card if compilation errors occur.
EL=F	Fatal diagnostics only are listed.
GO	Generated binary object file is loaded and executed at end of successful compilation.
L=SEE	Listed output appears on file SEE.
R=2	Long reference map is listed.
S=0	When COMPASS is called to assemble an intermixed COMPASS subprogram, it does not read in a systems text file.
SL=0	Source program is not listed.

Example:

```
FTN (GO,T)
```

Source program on INPUT file; object code on LGO; source program, short map, informative and fatal diagnostics listed on file OUTPUT; call-by-name sequence generated for calls to basic external functions; no debug package; standard compile mode; and unrounded arithmetic. Program is executed if no fatal errors occur.

Example:

FTN.

Selects the following options (unless option default values are changed by the installation):

A=0	I=INPUT	R=1
B=LGO	L=OUTPUT	ROUND=0
BL=0	LCM=D	S=SYSTEXT
C=0	ML=yyddd	SEQ=0
D=0	OL=0	SL
E=0	OPT=1	SYSEDIT=0
EL=I	P=0	T=0
G=0	PL=5000	TS=0
GO=0	Q=0	UO=0
		X=OLDPL
		Z=0

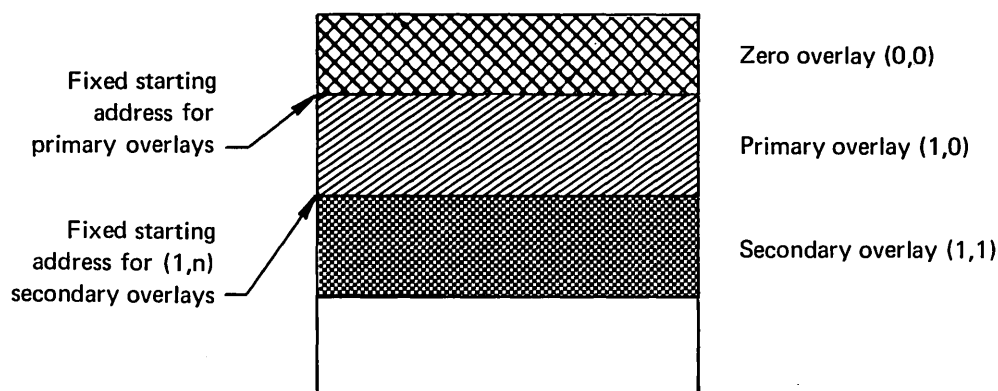


To reduce the amount of storage required, and to make more efficient use of his field length, a user can divide his program into overlays. Prior to execution, the object modules of an overlay program are linked by the loader and placed on a mass storage device or tape file in their absolute form; no time is required for linking at execution time. (See Loader Reference Manual for more details.)

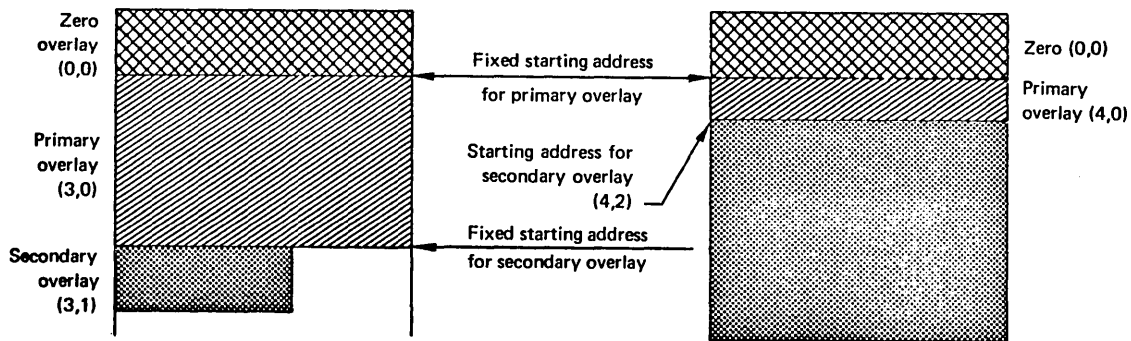
An overlay is a portion of a program written on a file in absolute form and loaded at execution time without relocation. As a result, the size of the resident loader for overlays can be reduced substantially. Overlays can be used when the organization of core can be defined prior to execution.

When each overlay is generated, the loading operation is completed by loading library and user subprograms and linking them together. The resultant overlay is in fixed format, in that internal references are fixed in their relationship to one another. The entire overlay has a fixed origin address within the field length and, therefore, is not relocatable. The overlay loader simply reads the required overlay from the overlay file and loads it starting at its pre-established origin in the user's field length.

Overlays are loaded into memory at three levels: zero, primary, and secondary.



The zero or main overlay is loaded first and remains in core at all times. A primary overlay may be loaded immediately following the zero overlay, and a secondary overlay immediately following the primary overlay. Overlays may be replaced by other overlays. For example, if a different secondary overlay is required, the overlay loader simply reads it from the overlay file and places it in memory at the same starting address as the previously loaded overlay.



When a primary overlay is loaded, the previously loaded primary overlay and any of its associated secondary overlays are destroyed. Loading a secondary overlay destroys a previously loaded secondary overlay. Loading any primary overlay destroys any other primary overlay. For this reason, no primary overlay may load other primary overlays.

Overlays are identified by a pair of integers:

zero or main overlay (0,0)

primary overlay (n,0)

secondary overlay (n,k)

n and k are positive integers in the range 0-77 octal. For any given program execution, all overlay identifiers must be unique.

For example, (1,0) (2,0) (3,0) (4,0) are primary overlays. (3,1) (3,2) (3,5) (3,7) are secondary overlays associated with primary overlay (3,0). Secondary overlays are denoted by the primary number and a non-zero secondary number. For example, (1,3) denotes that secondary overlay number 3 is related to primary overlay (1,0). (2,5) denotes secondary overlay 5 is related to primary overlay (2,0).

A secondary overlay can be called into core by its primary overlay or by the main overlay. Thus overlay (0,0) and overlay (1,0) may call (1,2); but overlay (2,0) may not call (1,2).

Overlay numbers (0,n) are not valid. For example, (0,3) is an illegal overlay number.

Execution is faster if the more commonly used subprograms are placed in the zero overlay, which remains in main memory at all times, and the less commonly used subprograms are placed in primary and secondary overlays which are called into memory as required.

An overlay can consist of one or more FORTRAN or COMPASS program units. Each overlay must contain one FORTRAN main program; it need not be the first program unit in the overlay. The program name in the PROGRAM statement becomes the primary entry point for the overlay when the overlay is called.

## OVERLAY COMMUNICATION

Data is passed between overlays through labeled or blank common. Any element of a labeled or blank common block in the main overlay (0,0) may be referenced by any higher level overlay. Any labeled or blank common declared in a primary overlay may be referenced only by the primary overlay and its associated secondary overlays — not by the zero overlay. If blank common is used for communicating between overlays, the user must ensure that sufficient field length is reserved to accommodate the largest loaded overlay in addition to blank common. Data stored in blank common must be used by each level of the overlay in exactly the same format, since no linkage is provided between the different levels of overlay and blank common at execution or load time.

Blank common is located at the top (highest address) of the first overlay in which blank common is declared. For example, if blank common is declared in the (0,0) overlay, it is located at the top of the (0,0) overlay and is accessible to all higher level overlays. If blank common is declared in the (1,0) overlay, it is allocated at the top of the (1,0) overlay and is accessible only to the associated (1,k) overlays. Labeled common blocks are generated in the overlay in which they are first encountered; data may only be preset in labeled common blocks in this overlay.

On the CDC CYBER 70/Model 76 and 7600 computers, LCM common blocks must be defined and preset in the main (0,0) overlay. The entire overlay structure can reference an LCM common block.

## CREATING AN OVERLAY

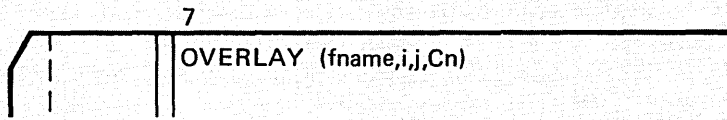
An overlay is established by an OVERLAY directive preceding the program units for that overlay. An overlay consists of all program units appearing between its OVERLAY directive and the next OVERLAY directive or an end-of-file (6/7/8/9) card. The directive must be punched starting in column 7 or later and must be contained wholly on one card.

The PROGRAM statement for the zero or main overlay (0,0) must specify all file names such as INPUT, OUTPUT, TAPE1, etc., required for all overlay levels. File names should not appear in PROGRAM statements for other than the (0,0) OVERLAY.

Loading overlays from a file requires an end-around search of the file for the specified overlay; this can be time consuming in large files. When speed is essential, each overlay should be written on a separate file, or it should be called in the same order in which it was generated.

The group of relocatable decks to be processed by the loader to create an overlay must be presented to the loader in the following order. The main overlay must be loaded first. Any primary group followed by its associated secondary group can follow, then any other primary group followed by its associated secondary group, and so forth.

The OVERLAY directive format is:



fname	Name of the file on which the generated overlay is to be written.
i,j	Overlay level numbers in understood octal.
Cn	Optional parameter consisting of the letter C and a 6-digit octal number, which indicates the overlay is to be loaded n words from the start of blank common.

The first overlay directive must have a file name and i,j must be 0,0. Subsequent directives can omit file name indicating that the overlays are to be written on the same file. All overlays need not reside on the same file. The second overlay directive must be the zero level of a primary overlay such as 3,0.

If the Cn parameter is omitted, the overlay is loaded in the normal way directly after the zero overlay. The Cn parameter cannot be included on the zero overlay directive. It is used on primary and secondary overlay directives to allow the programmer to change the size of blank common at execution time.

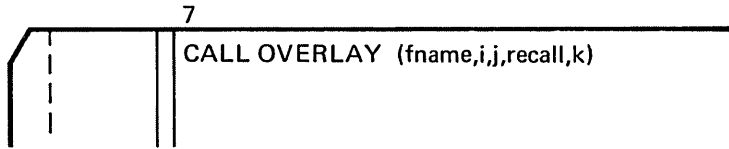
Example:

```
OVERLAY(FNAME,0,0)
PROGRAM CAT(INPUT,OUTPUT,TAPE5=INPUT)
.
.
.
OVERLAY(1,0)
PROGRAM A
.
.
.
OVERLAY(1,1)
PROGRAM B
.
.
.
OVERLAY(1,2)
PROGRAM C
.
.
.
OVERLAY(1,3)
PROGRAM D
.
.
.
```

All the above overlays are written on the file FNAME.

## CALLING AN OVERLAY

Primary and secondary overlays are called with a CALL OVERLAY statement while the zero overlay (0,0) is loaded when a program call control card is encountered. The format of the CALL OVERLAY statement is:



fname	The variable name of the location containing the name of the file in H format.
i,j	Overlay level numbers in understood octal.
recall	Optional recall parameter.
k	Optional parameter specifying where the overlay is located; can be zero, non-zero, or 7-character L format Hollerith constant.

If the k parameter is zero or not specified, the overlay is included in the file referenced by fname. If a non-zero k parameter is specified, fname is the variable name of the location containing the overlay to be loaded. If k is a 7-character Hollerith constant, the overlay is loaded from the library named in the constant. If k is any other non-zero value, the overlay is loaded from the global library set (refer to the appropriate operating system reference manual or the Loader Reference Manual).

The three parameters, fname, i, and j must be specified or the results are unpredictable.

When a RETURN or END statement is encountered in the main program of a zero overlay, execution of the program terminates and control returns to the operating system. When either of the statements is encountered in a primary or secondary overlay, control returns to the next executable statement after the CALL OVERLAY statement that invoked the current overlay.

Example 1:

```
CALL OVERLAY(1HA,1,0)
```

This statement causes a primary overlay to be loaded from the file named A.

Example 2:

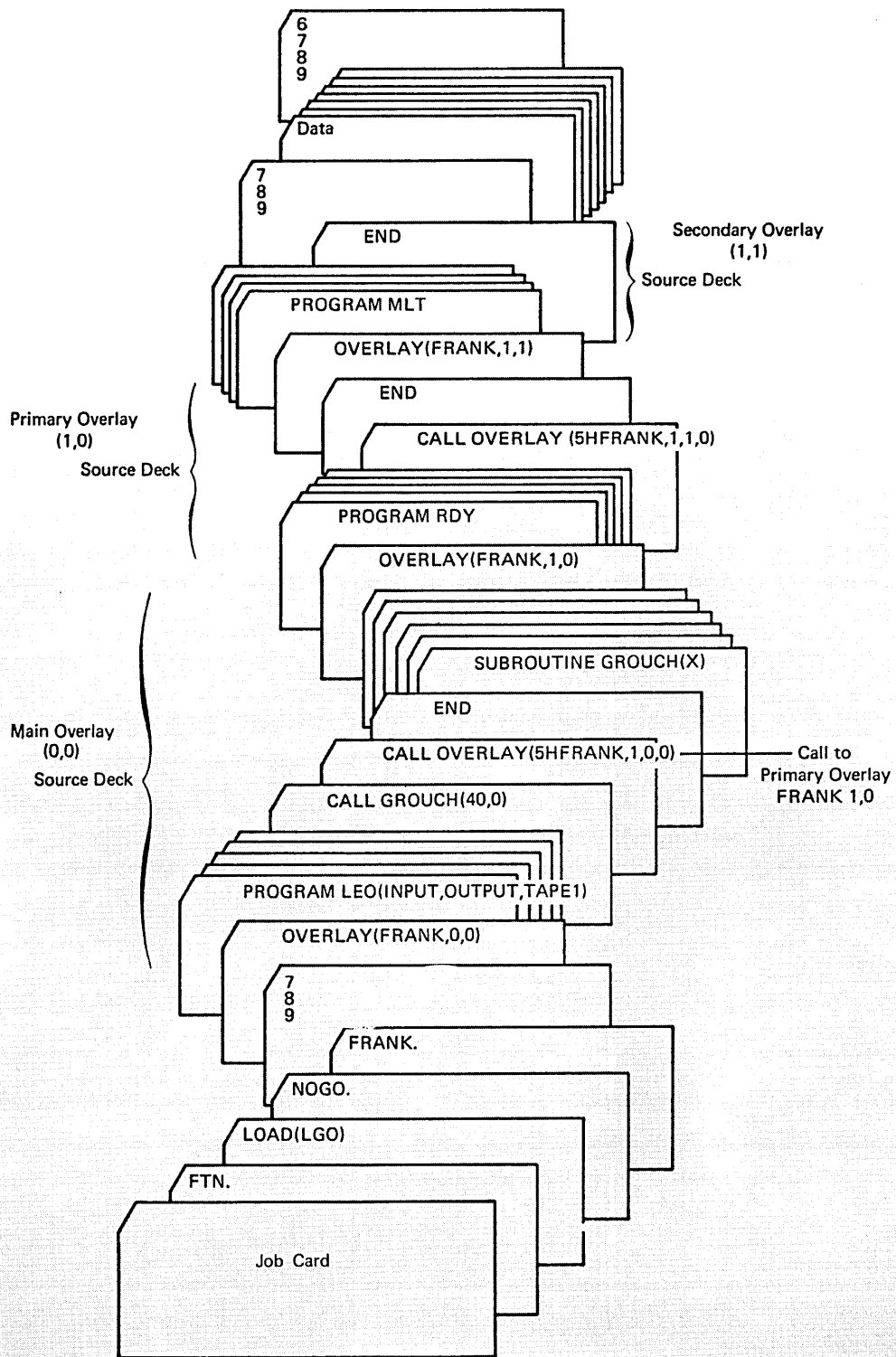
```
CALL OVERLAY(3HBJR,0,0,1)
```

This statement, which specifies the k parameter as a non-zero value, causes a main overlay with the name BJR to be loaded from the global library set.

Example 3:

```
OVERLAY(XFILE,0,0)
PROGRAM ONE(INPUT,OUTPUT,PUNCH)
.
.
.
CALL OVERLAY(5HXFILE,1,0,0)
.
.
.
STOP
END
OVERLAY(XFILE,1,0)
PROGRAM ONE ZERO
CALL OVERLAY(5HXFILE,1,1,0)
.
.
.
RETURN
END
OVERLAY(XFILE,1,1)
PROGRAM ONE ONE
.
.
.
RETURN
END
```

Execution of RETURN in the 1,1 overlay returns control to the statement in the 1,0 overlay following the 1,1 call. Execution of RETURN in the 1,0 overlay returns control to the statement in the main overlay following the 1,0 call.



Preparation of Overlay 0,0; 1,0; and 1,1

The above example illustrates the preparation of zero, primary and secondary overlays. The zero overlay, FRANK,0,0, consists of a main program LEO and a subroutine GROUCH. The primary overlay FRANK 1,0 consists of a main program MLT and a data deck. All three overlays reside on the file FRANK.

The LOAD(LGO) card requests the loader to load the program from the file LGO. As the loader reads file LGO, it encounters the overlay directive OVERLAY (FRANK,0,0) which instructs it to create a main overlay from the program and write it on file FRANK. When the absolute form of all the overlays has been generated, execution begins when the control card FRANK. is encountered. FRANK. causes the main overlay to be loaded from file FRANK and executed.

During execution of the main overlay, the CALL OVERLAY (5HFRANK,1,0,0) statement is encountered and the primary overlay 1,0 is loaded into central memory. The CALL OVERLAY (5HFRANK,1,1) statement in the primary overlay causes the secondary overlay to be loaded into memory.

The primary and secondary overlays can reside on files other than FRANK. For example, the primary overlay could be on file JIM and the secondary overlay on file JOHN.

```
FTN.  
LGO.  
FRANK.  
7/8/9  
OVERLAY (FRANK,0,0)  
PROGRAM LEO (INPUT,OUTPUT,TAPE1)  
.br/>.br/>.br/>CALL OVERLAY (3HJIM,1,0,0)  
.br/>.br/>OVERLAY (JIM,1,0)  
PROGRAM RDY  
.br/>.br/>CALL OVERLAY (4HJOHN,1,1,0)  
END  
OVERLAY (JOHN,1,1)  
PROGRAM MLT  
.br/>.br/>END
```

Example:

The following program, which contains several subroutines and functions, is to be used repeatedly. The entire program can be generated, therefore, as a main overlay and placed on the file in the absolute form. The control card CATALOG creates a permanent file OVRLY where the absolute form of the



program will be kept. When the program is required again, the permanent file OVRLY is called by an ATTACH control card.

The first program must be a main program; in this case program A.

Control Cards	{	FTN. LOAD(LGO) NOGO. CATALOG( REPEAT,OVRLY, ID=IBB) 7/8/9
Main Overlay	{	OVERLAY (REPEAT,0,0) PROGRAM A ( INPUT,OUTPUT,TAPE1) . . . END SUBROUTINE B . . . END FUNCTION C . . . END SUBROUTINE D . . . END REAL FUNCTION E . . . END 7/8/9 data 6/7/8/9

Main program A and the subroutines and functions B-E reside on the file REPEAT in absolute form. They can be called and executed without recompilation by the control cards:

```
job card  
ATTACH(REPEAT,OVRLY, ID=IBB)  
REPEAT.  
6/7/8/9
```

The operating system or Loader Reference Manual gives full details of the control cards which appear in the above program.



---

The debugging facility allows the programmer to debug programs within the context of the FORTRAN language. Using the statements described in this section, the programmer can check the following:

- Array bounds
- Assigned GO TO
- Subroutine calls and returns
- Function references and the values returned
- Values stored into variables and arrays
- Program flow

The debugging facility, together with the source cross reference map, is provided specifically to assist the programmer develop or convert programs.

The debugging mode is selected by specifying D or D=lfm on the FTN control card (section I-11). This control card parameter automatically selects fast compilation (OPT=0) and full error traceback (T option). If any other optimization level is specified, it will be ignored. The following examples are equivalent:

```
FTN (D)
FTN (D=INPUT,OPT=0,T)
FTN (D,OPT=2)      OPT=2 is ignored, OPT=0 and T are automatically selected.
```

Debug output is written on the file DEBUG. When the job terminates, the DEBUG file is given a print disposition and it is printed separately from the output file. To obtain debugging information on the same file as the source program, or any other file, DEBUG must be equivalenced to that file in the PROGRAM statement.

Examples:

```
PROGRAM EX (INPUT,OUTPUT,DEBUG=OUTPUT)
```

Debug output is interspersed with program output on the file OUTPUT.

```
PROGRAM EX(INPUT,OUTPUT,TAPEX,DEBUG=TAPEX)
```

Debug output is written on the file TAPEX.

The following control card sequence causes the debug output to be printed on the output file at termination of the job. It is not interspersed with the results of program execution.

```
FTN(D)
LGO.
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
EXIT(S)           Abnormal termination
REWIND(DEBUG)
COPYCF(DEBUG,OUTPUT)
```

When the debug mode is selected, programs execute regardless of most compilation errors. Execution, however, terminates at that point in the program where a fatal error is detected, and the following message is printed:

```
FATAL ERROR ENCOUNTERED DURING PROGRAM EXECUTION
DUE TO COMPILATION ERROR
```

Partial execution is prohibited for only four classes of errors:

- Any declarative error (any error encountered before at least one valid executable statement is found).
- Any fatal compilation error (defined in section III-2).
- Any missing (undefined) DO termination.
- Any illegal transfer into an innermost DO loop that is not an extended range loop.

Partial execution of programs containing fatal errors allows the programmer to insert debugging statements in the program to assist in locating fatal and non-fatal errors.

When a program is compiled in debug mode, at least 12000 (octal) words are required beyond the minimum field length for normal compilation. To execute, at least 2500 (octal) words beyond the minimum are required. The CPU time required for compilation is also greater than for normal OPT=0 compilation.

If the D option is not specified on the FTN control card, all debugging statements are treated as comments; therefore, it is not necessary to remove the debugging statements after the program is sufficiently debugged.

All debugging options are activated and deactivated at compile time only. This compile time processing is not to be confused with program flow at execution time.

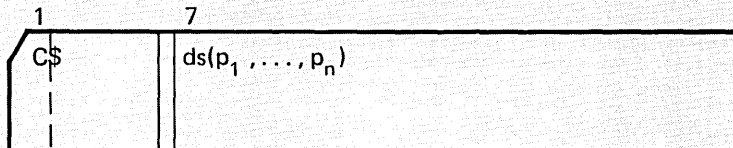
```

PROGRAM TEST (OUTPUT,DEBUG=OUTPUT)
.
.
.
GO TO 4
.
.
.
C$ (DEBUGGING OPTION)
C$ (DEBUGGING OPTION)
.
.
.
4 CONTINUE
.
.
.
END

```

Even though a section of code may never be executed, the debugging options are processed at compile time and are effective for the remainder of the program. In the above example, the code between the GO TO statement and the CONTINUE statement may never be executed. However, debugging statements between these statements are processed at compile time and are effective for the remainder of the program, or until deactivated by a C\$ OFF statement.

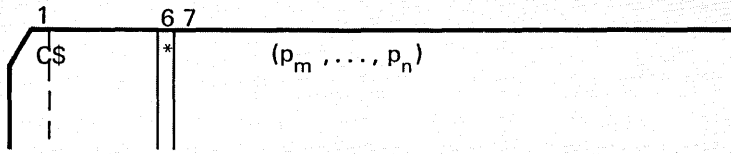
## DEBUGGING STATEMENTS



ds      Type of option, beginning after column 6: DEBUG, AREA, ARRAYS, CALLS, FUNCS, GOTOS, NOGO, OFF, STORES, TRACE

p<sub>i</sub>      Argument list; details extent of the option, ds (not used with NOGO, GOTOS; required for AREA, STORES; optional for other options)

## CONTINUATION CARD



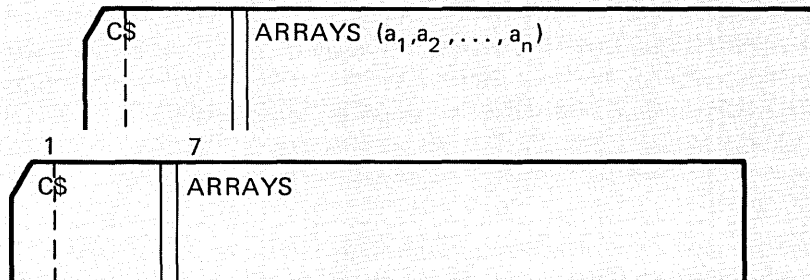
Debugging statements are written in columns 7-72, as in a normal FORTRAN statement, but columns 1 and 2 of each statement must contain the characters C\$. Any character, other than a blank or zero, in column 6 denotes a continuation line. Columns 3, 4, and 5 of any debugging statement must be blank. The restriction on the number of continuation lines is the same as for FORTRAN continuation lines.

Comment cards may be interspersed with debugging statements. The statement separator (\$) cannot be used with debugging statements. When the debug mode is not selected, all debugging statements are treated as comments.

Example:

```
C$  ARRAYS (A, BNUMB,Z10, C, DLIST, MATRIX,  
C$  *NSUM, GTEXT,  
C$  *TOTAL)
```

## ARRAYS STATEMENT



a<sub>1</sub>, ..., a<sub>n</sub> array names

The ARRAYS statement initiates subscript checking on specified arrays. If no argument list is specified, all arrays in the program unit are checked. Each time a specified or implied element of an array is referenced, the calculated subscript is checked against the dimensioned bounds. The address is calculated according to the method described in figure 2-1, section 2. Subscripts are not checked individually. If the address is found to be greater than the storage allocated for the array or less than one, a diagnostic is issued. The reference then is allowed to occur. Bounds checking is not performed for array references in input/output statements, or in ENCODE/DECODE statements.

```

PROGRAM ARRAYS (OUTPUT,DEBUG=OUTPUT)
INTEGER A(2), B(4), C(6), D(2,3,4)
PRINT 1
1 FORMAT(*0    ARRAYS  EXAMPLE*///)
*
*   TURN ON ARRAYS FOR ARRAYS  A  AND  D
*
C$  ARRAYS (A, D)
*
*   A(3) IS OUT OF BOUNDS  AND  ARRAYS IS ON FOR  A, SO A DIAGNOSTIC
*   IS PRINTED.
*
A(3) = 1
*
*   B(5) IS OUT OF BOUNDS  BUT  ARRAYS IS NOT ON FOR  B, SO NO
*   DIAGNOSTIC IS PRINTED.
*
B(5) = 1
*
C(2) = A(A(3))
*
*   EVEN THOUGH A(3) WAS OUT OF BOUNDS, THE ASSIGNMENT TOOK PLACE.
*   A(A(3)) IS EQUIVALENT TO  A(1). THIS SUBSCRIPT IS IN BOUNDS,
*   HOWEVER THE REFERENCE TO A(3) WILL CAUSE A DIAGNOSTIC.
*
D(-5,0,6) = 99
*
*   FOR THE ARRAY D(L,M,N) THE STORAGE ALLOCATED IS L * M * N.
*   THE SUBSCRIPT FOR THE ELEMENT D(I,J,K) IS COMPUTED AS FOLLOWS
*   (1 + L*(J-1) + M*(K-1))
*   FOR THE ELEMENT D(-5,0,6) THE SUBSCRIPT APPEARS TO
*   BE OUT OF BOUNDS BECAUSE THE INDIVIDUAL SUBSCRIPTS ARE OUT
*   OF BOUNDS.  HOWEVER, 22, THE COMPUTED ADDRESS, IS LESS THAN
*   24, THE STORAGE ALLOCATED, AND NO DIAGNOSTIC IS ISSUED.
*
*   TURN ON ARRAYS FOR ALL ARRAYS
*
C$  ARRAYS
*
*   WITH THIS FORM ALL ARRAY REFERENCES WILL BE CHECKED. THERE WILL
*   BE DIAGNOSTICS FOR B(5), C(-1), AND D(0,0,0).  BECAUSE A(2)
*   IS IN BOUNDS AND A(4) IS IN AN I/O STATEMENT, THERE WILL BE
*   NO DIAGNOSTICS FOR EITHER OF THESE REFERENCES.
*
A(2) = 1
B(5) = 2 + C(-1)
D(0,0,0) = 1
PRINT 2, A(4)
2 FORMAT(1X, A10)
END

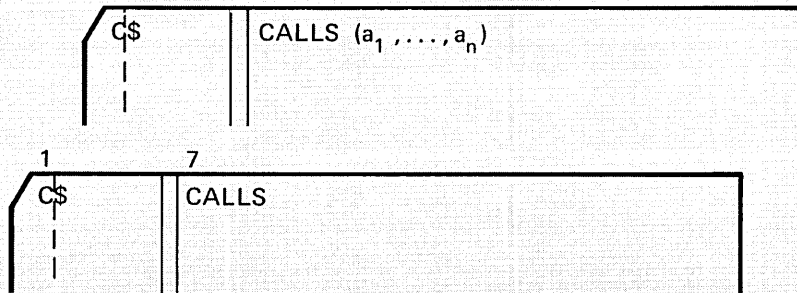
```

ARRAYS EXAMPLE

```

/DEBUG/ ARRAYS AT LINE 13- THE SUBSCRIPT VALUE OF      3 IN ARPAY A      EXCEEDS DIMENSIONED BOUND OF      2
/DEBUG/          AT LINE 20- THE SUBSCRIPT VALUE OF     3 IN ARRAY A      EXCEEDS DIMENSIONED BOUND OF      2
/DEBUG/          AT LINE 47- THE SUBSCRIPT VALUE OF     5 IN ARRAY B      EXCEEDS DIMENSIONED BOUND OF      4
/DEBUG/          AT LINE 47- THE SUBSCRIPT VALUE OF    -1 IN ARRAY C      EXCEEDS DIMENSIONED BOUND OF      6
/DEBUG/          AT LINE 48- THE SUBSCRIPT VALUE OF     -8 IN ARRAY D      EXCEEDS DIMENSIONED BOUND OF     24
    
```

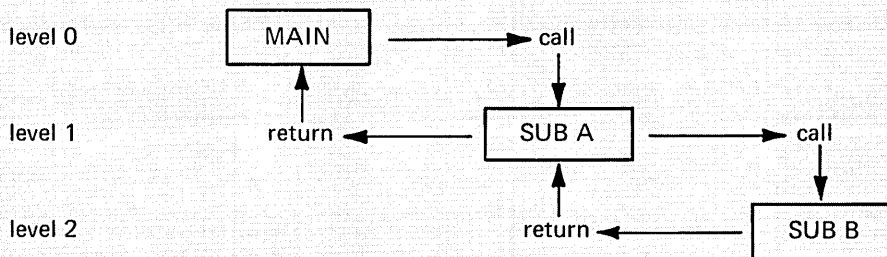
**CALLS STATEMENT**



$a_1, \dots, a_n$  subroutine names

The CALLS statement initiates tracing of calls to and returns from specified subroutines. If there is no argument list all subroutines will be traced. Non-standard returns, specified in a RETURNS list, are included. To trace alternate entry points to a subroutine, either the entry points must be explicitly named in the argument list, or the form with no argument list must be used (all external calls traced). The message printed contains the names of the calling and called routines, as well as the line and level number of the call and return.

A main program is at level zero; a subroutine or a function called by the main program is at level 1, another subprogram called by the subprogram at level 1, is at level 2, and so forth. Calls are shown in order of ascending level number, returns in order of descending level number.



For example, subroutine SUB A is called at level 1 and a return is made to level 0. SUB B is called at level 2 and a return is made to level 1.



Example:

```
PROGRAM CALLS(OUTPUT,DEBUG=OUTPUT)
PRINT 1
1 FORMAT(*0      CALLS TRACING*)
*
*   TURN ON CALLS FOR SUBROUTINES CALLS1 AND CALLS2
*
C$ CALLS(CALLS1, CALLS2)
   X = 1.
   CALL CALLS1 (X,Y), RETURNS (10)
10 IF (X .EQ. 1.) CALL CALLS2(X)
   CALL SUBNOT
   CALL CALLS1E (X,Y)
*
*   DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*   CALLS1 AND CALLS2. SINCE THE CALLS ARE FROM THE MAIN PROGRAM,
*   THEY ARE AT LEVEL 0. THE CALLS TO SUBNOT AND THE ALTERNATE
*   ENTRY POINT CALLS1E ARE NOT TRACED BECAUSE THEY DO NOT APPEAR
*   IN THE ARGUMENT LIST OF THE C$ CALLS STATEMENT.
*
*   TURN ON CALLS FOR ALL SUBROUTINES
*
C$ CALLS
   CALL SUBNOT
   CALL CALLS2(X)
   CALL CALLS1E (X,Y)
*   DEBUG MESSAGES WILL BE PRINTED FOR CALLS TO AND RETURNS FROM
*   SUBNOT, CALLS2, AND CALLS1E, SINCE ALL CALLS ARE TO BE
*   TRACED.
END

SUBROUTINE CALLS1(X,Y), RETURNS(A)
Y = -X
IF (Y .NE. X) RETURN A
RETURN
ENTRY CALLS1E
RETURN
END

SUBROUTINE CALLS2(X)
CALL CALLS1(X,Y), RETURNS(5)
5 RETURN
END

SUBROUTINE SUBNOT
X = -1.
CALL CALLS1(X,Y), RETURNS(5)
5 RETURN
END
```

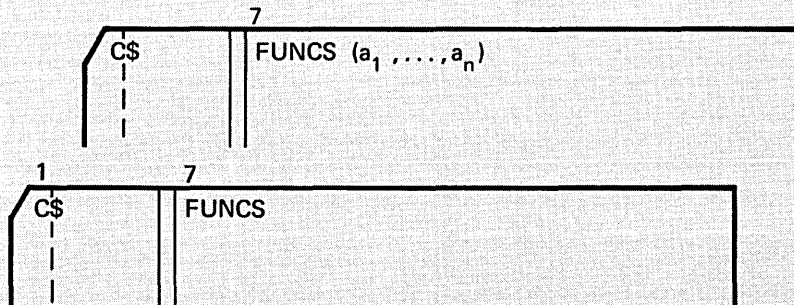
```

CALLS TRACING
/DEBUG/ CALLS AT LINE 9- ROUTINE CALLS1 CALLED AT LEVEL 0
/DEBUG/ AT LINE 10- ROUTINE CALLS1 RETURNS TO LEVEL 0 AT STATEMENT 10
/DEBUG/ AT LINE 10- ROUTINE CALLS2 CALLED AT LEVEL 0
/DEBUG/ AT LINE 11- ROUTINE CALLS2 RETURNS TO LEVEL 0
/DEBUG/ AT LINE 24- ROUTINE SUBNOT CALLED AT LEVEL 0
/DEBUG/ AT LINE 25- ROUTINE SUBNOT RETURNS TO LEVEL 0
/DEBUG/ AT LINE 25- ROUTINE CALLS2 CALLED AT LEVEL 0
/DEBUG/ AT LINE 26- ROUTINE CALLS2 RETURNS TO LEVEL 0
/DEBUG/ AT LINE 26- ROUTINE CALLS1E CALLED AT LEVEL 0
/DEBUG/ AT LINE 27- ROUTINE CALLS1E RETURNS TO LEVEL 0

```

In this example, only calls from the main program are traced. To trace calls from subprograms, a C\$ CALLS statement must appear in the subprograms.

### FUNCS STATEMENT



If no function names ( $a_1, \dots, a_n$ ) are listed, all external functions referenced in the program unit are traced. Alternate entry points must be named explicitly in the argument list, or implicitly in the C\$ FUNCS statement with no parameters.

Function tracing is similar to call tracing, but the value returned by the function is included in the debug message. Each time a specified external function is referenced, a message is printed which contains the routine name and line number containing the reference, function name and type, value returned, and level number. The level concept is the same as for the CALLS statement.

Statement function references are not traced, nor are function references, in input/output statements.

Example:

The following program, VARDIM2, illustrates both the C\$ FUNCS and C\$ CALLS statements. All function references in the main program are traced because C\$ FUNCS appears without an argument list; references to functions PVAL, AVG and MULT and the values returned to the main program (level 0) are traced. All subroutines in the main program are traced also because a C\$ CALLS statement without an argument list appears.

Function references within the FUNCTION subprograms PVAL, AVG and MULT are traced since C\$ FUNCS statements appear within these subprograms. If no C\$ FUNCS statements appear in the subprograms, only main program function references will be traced.

```

C      PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
C      THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBPROGRAM CONCEPTS
COMMON X(4,3)
REAL Y(6)
5      EXTERNAL MULT, AVG
      PVALSF(X,Y) = PVAL(X,Y)
C$     CALLS
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
10     CALL INC(X,12,-5.)
C
C      ALL EXTERNAL CALLS ARE DIAGNOSED.
C
C$     FUNCS
15     AA = PVALSF(12,AVG)
      AM = PVALSF(12,MULT)
C
C      PVALSF IS A STATEMENT FUNCTION, SO THE FUNCS STATEMENT DOES NOT
C      APPLY TO IT AND NO MESSAGE IS PRINTED. HOWEVER, THE EXTERNAL
20     FUNCTION PVAL IS REFERENCED WITHIN THE CODE FOR PVALSF,
C      AND THOSE REFERENCES ARE DIAGNOSED.
C      MULT AND AVG ARE NAMES AS ARGUMENTS TO PVALSF, HOWEVER, THE
C      FUNCTIONS ARE NOT ACTUALLY REFERENCED AND MESSAGES ARE NOT
C      PRINTED.
25     C
      STOP
      END

SUBROUTINE SET (A,M,V)
C      SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
      DIMENSION A(M)
      DO1I=1,M
5      A(I)=0.0
C
      ENTRY INC
C      INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
      DO2I=1,M
10     A(I)=A(I)+V
      RETURN
      END
```

```

SUBROUTINE IOTA (A,M)
C   IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C   THE ARRAY A
5   DIMENSION A(M)
1   DO1I=1,M
   A(I)=I
   RETURN
   END

FUNCTION PVAL(SIZE,WAY)
C   PVAL COMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALUE IS RETURNED
C   BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED. SIZE IS AN INTEGER
C   VALUE PASSED ON TO THE FUNCTION:
5   INTEGER SIZE
   C$   FUNCS(ABS)
   PVAL=ABS(WAY(SIZE))
C
C   WAY DOES NOT APPEAR IN THE ARGUMENT LIST FOR THE FUNCS STATEMENT,
10  C   SO ONLY THE REFERENCE TO ABS IS DIAGNOSED.
   C
   RETURN
   END

FUNCTION AVG(J)
C   AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON.
   COMMON A(100)
5   AVG=0.
   DO1I=1,J
1   AVG=AVG+A(I)
   C$   FUNCS
C
C   ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
10  C
   AVG=AVG/FLOAT(J)
   RETURN
   END

REAL FUNCTION MULT(J)
C   MULT COMPUTES A STRANGE AVERAGE. IT MULTIPLIES THE FIRST AND 12TH
C   ELEMENTS OF COMMON AND SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C   BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
5   C
   COMMON ARRAY(12)
   C$   FUNCS
C
C   ALL EXTERNAL FUNCTION REFERENCES WILL BE DIAGNOSED.
10  C
   MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
   RETURN
   E N D

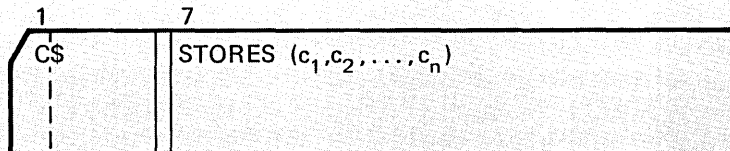
```

```

/DEBUG/ VARDIM2 AT LINE 8- ROUTINE SET CALLED AT LEVEL 0
/DEBUG/          AT LINE 9- ROUTINE SET RETURNS TO LEVEL 0
/DEBUG/          AT LINE 9- ROUTINE IOTA CALLED AT LEVEL 0
/DEBUG/          AT LINE 10- ROUTINE IOTA RETURNS TO LEVEL 0
/DEBUG/          AT LINE 10- ROUTINE INC CALLED AT LEVEL 0
/DEBUG/          AT LINE 11- ROUTINE INC RETURNS TO LEVEL 0
/DEBUG/          AT LINE 15- REAL FUNCTION PVAL CALLED AT LEVEL 0
/DEBUG/  AVG     AT LINE 11- REAL FUNCTION FLOAT CALLED AT LEVEL 2
/DEBUG/          AT LINE 11- REAL FUNCTION FLOAT RETURNS A VALUE OF 12.00000000 AT LEVEL 2
/DEBUG/  PVAL    AT LINE 7- REAL FUNCTION ABS CALLED AT LEVEL 1
/DEBUG/          AT LINE 7- REAL FUNCTION ABS RETURNS A VALUE OF 1.50000000 AT LEVEL 1
/DEBUG/  VARDIM2 AT LINE 15- REAL FUNCTION PVAL RETURNS A VALUE OF 1.50000000 AT LEVEL 0
/DEBUG/          AT LINE 16- REAL FUNCTION PVAL CALLED AT LEVEL 0
/DEBUG/  MULT    AT LINE 11- REAL FUNCTION AVG CALLED AT LEVEL 2
/DEBUG/          AT LINE 11- REAL FUNCTION AVG CALLED AT LEVEL 3
/DEBUG/  AVG     AT LINE 11- REAL FUNCTION FLOAT RETURNS A VALUE OF 6.00000000 AT LEVEL 3
/DEBUG/          AT LINE 11- REAL FUNCTION AVG RETURNS A VALUE OF -1.50000000 AT LEVEL 2
/DEBUG/  PVAL    AT LINE 7- REAL FUNCTION ABS CALLED AT LEVEL 1
/DEBUG/          AT LINE 7- REAL FUNCTION ABS RETURNS A VALUE OF 26.50000000 AT LEVEL 1
/DEBUG/  VARDIM2 AT LINE 16- REAL FUNCTION PVAL RETURNS A VALUE OF 26.50000000 AT LEVEL 0

```

## STORES STATEMENT



An argument list must be specified for the STORES statement.

( $c_1, \dots, c_n$ ) are variable names or expressions in the forms:

variable name

variable name .relational operator. constant

variable name .relational operator. variable name

variable name .checking operator.

Relational operators are .EQ., .NE., .GT., .GE., .LT., .LE.

Checking operators are .RANGE., .INDEF., .VALID.

Example:

```

C$   STORES(SUM, DGAMP, AX, NET.LT.4, ROWSUM.RANGE.)
C$   STORES(A1, AGAIN, I, A2.EQ.5.0, IAGAIN.LE.IVAR)
C$   STORES(C.EQ.(1., 1.), L.VALID., D.NE.10.004)
C$   STORES(G.RANGE., TR.EQ..FALSE.)

```

The STORES statement is used to record changes in value of specified variables or arrays. The STORES statement applies only to assignment statements. Values changed as a result of input/output, or use in DATA, ASSIGN, COMMON, or argument lists to subroutines and functions are not detected. The STORES statement does not apply to the index variable in a DO loop.

If the value of a variable in an EQUIVALENCE group is changed, the STORES statement will not detect changes to the value of other variables in the group.

## VARIABLE NAMES

In the first form of the STORES statement, a message is printed each time the value of a variable or an array element changes. The variable and name of the array must appear as arguments in the C\$ STORES statement.

Example:

```

                    PROGRAM STORES (INPUT,OUTPUT,DEBUG = OUTPUT)
                    LOGICAL L1,L2
C$ STORES (NSUM,DGAMP,AX)
5     NSUM = 20
        DGAMP = .5
        AX = 7.2 + DGAMP
        L1 = .TRUE.
        L2 = .FALSE.
        PLANT = 2.5
10    A = 7.5
        PRINT 3
3     FORMAT (1H0)
        STOP
        END

```

Each time the value of the variables NSUM, DGAMP and AX changes, a message is printed. The values of PLANT, A, L1 and L2 are not printed, since they do not appear in the argument list.

```

/DEBUG/ STORES AT LINE 4- THE NEW VALUE OF THE VARIABLE NSUM IS 20
/DEBUG/ AT LINE 5- THE NEW VALUE OF THE VARIABLE DGAMP IS .5000000000
/DEBUG/ AT LINE 6- THE NEW VALUE OF THE VARIABLE AX IS 7.7000000000

```

Array elements should not be specified in the parameter list of a STORES statement; the array name must be used. If an array element name appears, an informative diagnostic is printed.

Example:

```

                    PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
                    REAL A(10), B(4,2)
C$ STORES (A,B)
5     B(1,2) = 5.5
      B(4,2) = 0.
      DO 4 N = 1,3
4     A(N) = N+1
      PRINT 5
      5 FORMAT (1H0)
10    STOP
      END

```

```

/DEBUG/ STORAR AT LINE 4- THE NEW VALUE OF THE VARIABLE B IS 5.500000000
/DEBUG/ AT LINE 5- THE NEW VALUE OF THE VARIABLE B IS 0.
/DEBUG/ AT LINE 7- THE NEW VALUE OF THE VARIABLE A IS 2.000000000
/DEBUG/ AT LINE 7- THE NEW VALUE OF THE VARIABLE A IS 3.000000000
/DEBUG/ AT LINE 7- THE NEW VALUE OF THE VARIABLE A IS 4.000000000

```

The values stored into array elements B(1,2) and B(4,2) appear in the debug output under the array name B in both cases, and array elements A(1), A(2), and A(3) appear under the array name A.

## RELATIONAL OPERATORS

In the second form of the C\$ STORES statement, a message is printed only when the stored value satisfies the relation specified in the argument list. The two components of the relational expression must be of the same type.

```

                    PROGRAM ST3 (INPUT,OUTPUT,DEBUG=OUTPUT)
5     FORMAT (1H0)
      PRINT 5
      M = 5
C$ STORES (I.EQ.3,N.LE.M,ANT)
      I = 3
      I = 4
      N = 4
      N = 6
      J = 10
      ANT = 77.0
      END

```

```

/DEBUG/ ST3 AT LINE 6- THE NEW VALUE OF THE VARIABLE I IS 3
/DEBUG/ AT LINE 8- THE NEW VALUE OF THE VARIABLE N IS 4
/DEBUG/ AT LINE 11- THE NEW VALUE OF THE VARIABLE ANT IS 77.00000000

```

I appears in the debug output when it is equal to 3; N appears when it is less than or equal to M. Since no relational operator is specified with ANT, it is printed whenever the value changes.

## CHECKING OPERATORS

In the third form of the STORES statement, a message is issued only when the stored value is out of range, indefinite, or invalid as specified by the checking operator.

RANGE	Out of range
INDEF	Indefinite
VALID	Out of range or indefinite

For example:

```
C$ STORES (ROWSUM .RANGE., COLSUM .VALID.)
```

Whenever the value to be stored into ROWSUM is out of range, a message is printed. Whenever the value to be stored into COLSUM is out of range or indefinite, a message is printed.

## HOLLERITH DATA

Hollerith data stored in a variable of type integer is interpreted by the STORES statement as an integer number. Hollerith data stored in a variable of type real or double precision is interpreted as a real or double precision number.

In the following example, the three integer variables IHOLL, IRIGHT and ILEFT contain the characters PA in display code (20 and 01).

```
IHOLL      20015555555555555555
           P A  blank fill

IRIGHT     00000000000000002001
           zero fill      P A

ILEFT      20010000000000000000
           P A zero fill
```



Example:

```

PROGRAM DEMOL (INPUT,OUTPUT,DEBUG=OUTPUT)
CS  DEBUG
CS  STORES (IHOLL,IRIGHT,ILEFT,HOLL)
5
    IHOLL=2HPA
    IRIGHT=2RPA
    ILEFT=2LPA
    HOLL=2HPA
10  PRINT 1
    1 FORMAT (1H0)
    STOP
    END

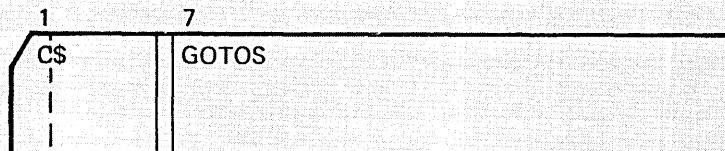
/DEBUG/ DEMOL AT LINE 6- THE NEW VALUE OF THE VARIABLE IHOLL IS *****
/DEBUG/ DEMOL AT LINE 7- THE NEW VALUE OF THE VARIABLE IRIGHT IS 1025
/DEBUG/ DEMOL AT LINE 8- THE NEW VALUE OF THE VARIABLE ILEFT IS *****
/DEBUG/ DEMOL AT LINE 9- THE NEW VALUE OF THE VARIABLE HOLL IS .4021071096E+15

```

The variables IHOLL, IRIGHT, and ILEFT are interpreted as integer numbers. Since the field width allocated by the STORES option (14 digits) is insufficient to contain the converted quantities represented by IHOLL and ILEFT, these fields are filled with asterisks. The variable IRIGHT is converted and printed out by the STORES option as 1025.

The variable HOLL is interpreted as a real number, and its value is printed out.

### GOTOS STATEMENT



No argument list can be specified with the C\$ GOTOS statement. The GOTOS statement initiates checking of all assigned GO TO statements to ensure that the statement label assigned to the integer variables is in the GO TO statement list. If no match is found, a message is printed and transfer of control continues.

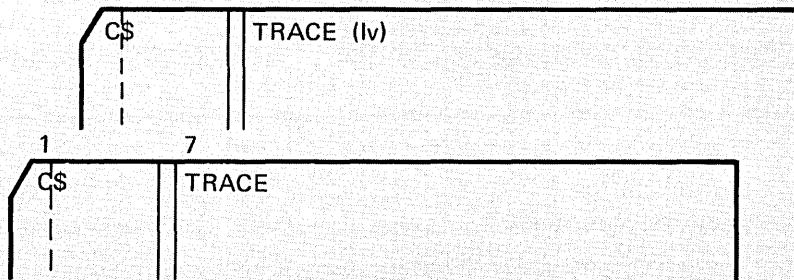
```

PROGRAM GO TOS (OUTPUT,DEBUG=OUTPUT)
INTEGER A
CS  GOTOS
*   (GOTOS NEVER USES AN ARGUMENT LIST)
5  *
    ASSIGN 1 TO A
    GO TO A (1, 2, 3)
*
*   IN THIS CASE NO MESSAGE IS PRINTED SINCE THE LABEL ASSIGNED TO
10 *   A IS IN THE GOTO LIST.
*
    4 PRINT 10
    10 FORMAT(*) --CONTROL TRANSFERED TO STATEMENT LABEL 4--*)
    STOP
15  1 ASSIGN 4 TO A
    GO TO A (1, 2, 3)
*
*   IN THIS CASE A MESSAGE IS PRINTED SINCE THE LABEL 4 IS NOT IN
20 *   THE GOTO LIST. CONTROL THEN TRANSFERS TO LABEL 4.
*
    2 CONTINUE
    3 CONTINUE
    END

/DEBUG/ GOTOS AT LINE 16- ASSIGNED GOTO INDEX CONTAINS THE ADDRESS 002151. NO MATCH FOUND IN STATEMENT LABEL ADDRESS LIST
--CONTROL TRANSFERED TO STATEMENT LABEL 4--

```

## TRACE STATEMENT



lv is a level number 0-49. If lv = 0, tracing occurs only outside DO loops. If lv = n, tracing occurs up to and including level n in a DO nest. If no level is specified, tracing occurs only outside DO loops.

The CS TRACE statement traces the following transfers of control within a program unit:

- GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- True side of logical IF

Transfers resulting from a return specified in a RETURNS list are not traced. (These can be checked by the CS CALLS statement.)

If an out-of-bound computed GO TO is executed, the value of the incorrect index is printed before the job is terminated.

Messages are printed each time control transfers during execution. The message contains the routine name, the line where the transfer took place, and the number of the line to which the transfer was made, as well as the statement number of this line, if present.

A message is printed each time control transfers at a level less than or equal to the one specified by lv. For example, if a statement CS TRACE(2) appears before a sequence of DO loops nested four deep, tracing takes place in the two outermost loops only.

TRACE messages are produced at execution time, but TRACE levels are assigned at compile time; therefore, the compile time environment determines the tracing status of any given statement. For example, a DO loop TRACE statement applies only to control transfers occurring between the DO statement and its terminal statement at compile time (physically between the two in the source listing).

Example:

```

PROGRAM P(OUTPUT,DEBUG=OUTPUT)
DATA J/0/
C$ TRACE(1)
IF (J.EQ.0) GO TO 11
level 0
5 level 1
11 DO 1 I1 = 1, 3
IF ( (J+1).EQ.I1 ) GO TO 12
level 2
12 J = 1
DO 2 I2 = 1, 5
J = J + I2
GO TO 2
2 CONTINUE
C$ TRACE(3)
DO 20 I2 = 1, 3
IF ( I2.EQ.3 ) GO TO 20
J = 2
level 3
15 DO 3 I3 = 1, 4
IF ( J.GT.I3 ) GO TO 31
level 4
20 DO 4 I4 = 1, ?
GO TO 4
4 CONTINUE
3 CONTINUE
20 CONTINUE
J = 0
1 CONTINUE
25 END

```

```

/DEBUG/ P AT LINE 4- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 4- CONTROL WILL BE TRANSFERRED TO STATEMENT 11 AT LINE 5
/DEBUG/ AT LINE 6- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 6- CONTROL WILL BE TRANSFERRED TO STATEMENT 12 AT LINE 7
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20 AT LINE 22
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20 AT LINE 22
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 17- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 17- CONTROL WILL BE TRANSFERRED TO STATEMENT 31 AT LINE 18
/DEBUG/ AT LINE 14- CONTROL TRANSFERRED TO THE TRUE SIDE OF LOGICAL IF EXPRESSION
/DEBUG/ AT LINE 14- CONTROL WILL BE TRANSFERRED TO STATEMENT 20 AT LINE 22

```

In the first level 2 loop no debug messages are printed since the TRACE(1) statement is in effect. However, when the TRACE(3) statement becomes effective, flow is traced up to and including level 3. There are no messages for transfers within the level 4 loop. To trace only inner loops, for example levels 3 and 4 in the above example, a C\$ TRACE(4) statement is placed immediately before the DO statement for the level 3 loop (line 16). A C\$ OFF (TRACE) statement is placed after the terminal line for the level 3 loop, so that subsequent program flow in levels 0, 1, and 2 is not traced.

The level number applies to the entire program unit; it is not relative to the position of the CS TRACE statement in the program. For example, to trace the level 4 DO loop in Program P:

```
C$ TRACE(4)
```

must be specified. Positioning the statement CS TRACE(1) before statement 31 would not achieve the same result.

Care must be taken with the use of debugging statements within DO loops. Since nested loops are executed more frequently, the quantity of debug output may quickly multiply.

The CS TRACE (lv) statement traces transfers of control within DO loops; however, transfers between the terminal statement and the DO statement are not traced.

Example:

```
DO 100 I = 1,10  
.  
.  
.  
100 CONTINUE
```

Transfers from statement 100 to the DO statement are not traced.

## NOGO STATEMENT



No argument list can be specified with this statement. The NOGO statement suppresses partial execution of a program containing compilation errors.

If a NOGO statement is present anywhere in the program, it applies to the entire program. It is therefore not affected by an OFF statement or by bounds in an AREA statement.

## DEBUG DECK STRUCTURE

Debugging statements may be interspersed with FORTRAN statements in the source deck of a program unit (main program, subroutine, function). The debugging statements apply to the program unit in which they appear. Interspersed debugging statements (figure 13-1) change the FORTRAN generated line numbers for a program.

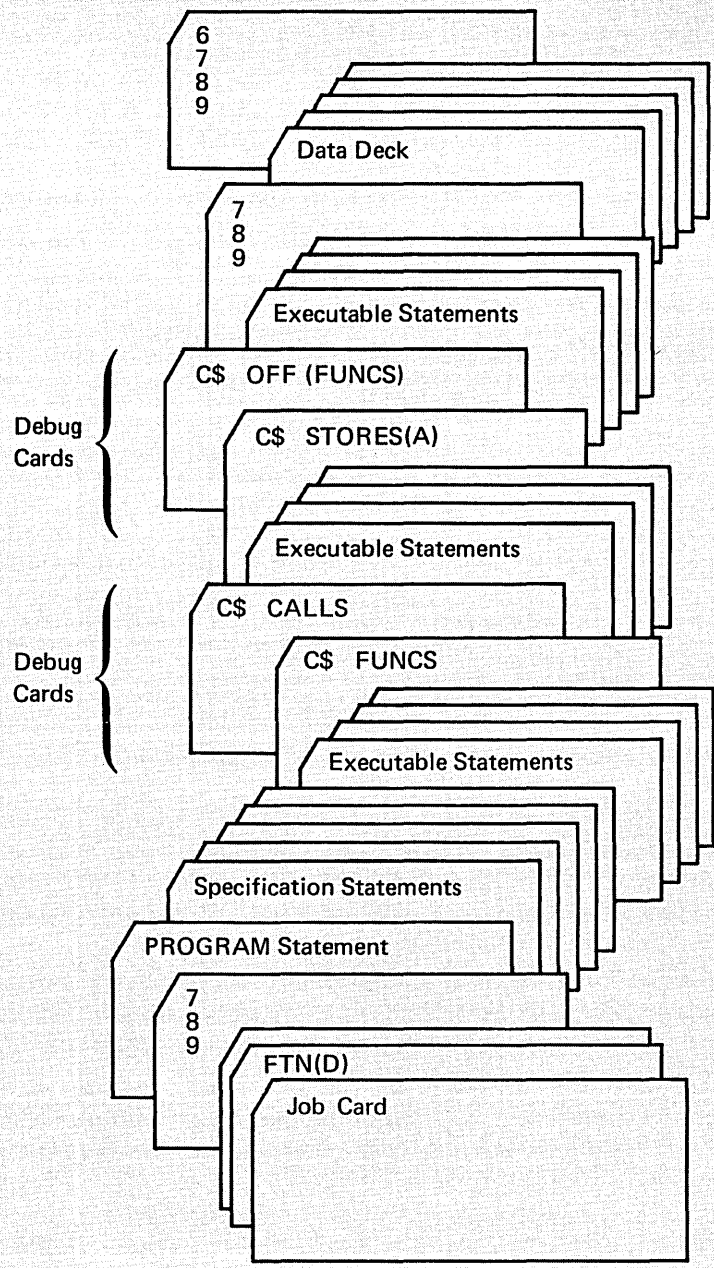
Debugging statements also may be grouped to form a debugging deck in one of the following ways:

As a deck placed immediately after the PROGRAM, SUBROUTINE or FUNCTION statement heading the routine to which the deck applies (internal debugging deck, figure 13-3). Any names specified in the DEBUG statement, other than the name of the enclosing routine, are ignored.

As a deck immediately preceding the first source deck in the job INPUT file (external debugging deck, figure 13-2).

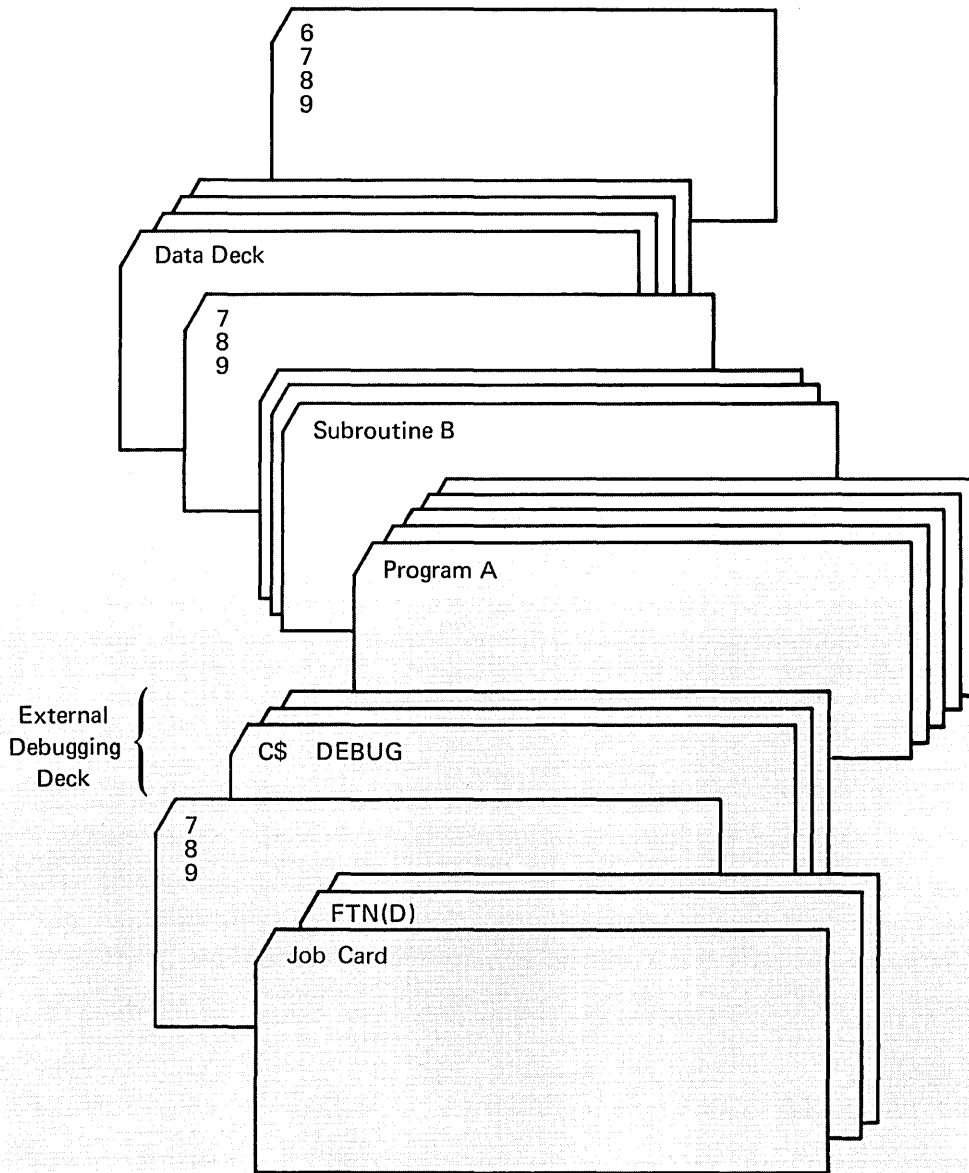
As one or more decks on the file specified by the D parameter on the FTN control card (external debugging deck, figure 13-4). When no name is specified by the D parameter, the INPUT file is assumed.

All debugging decks must be headed by a C\$ DEBUG card. In an internal debugging deck, the C\$ DEBUG card is used without an argument list, since the deck can only apply to the routine in which it is inserted. In an external debugging deck, a C\$ DEBUG may be used with or without an argument list. The statements in the external debugging deck apply to all program units in the compilation.



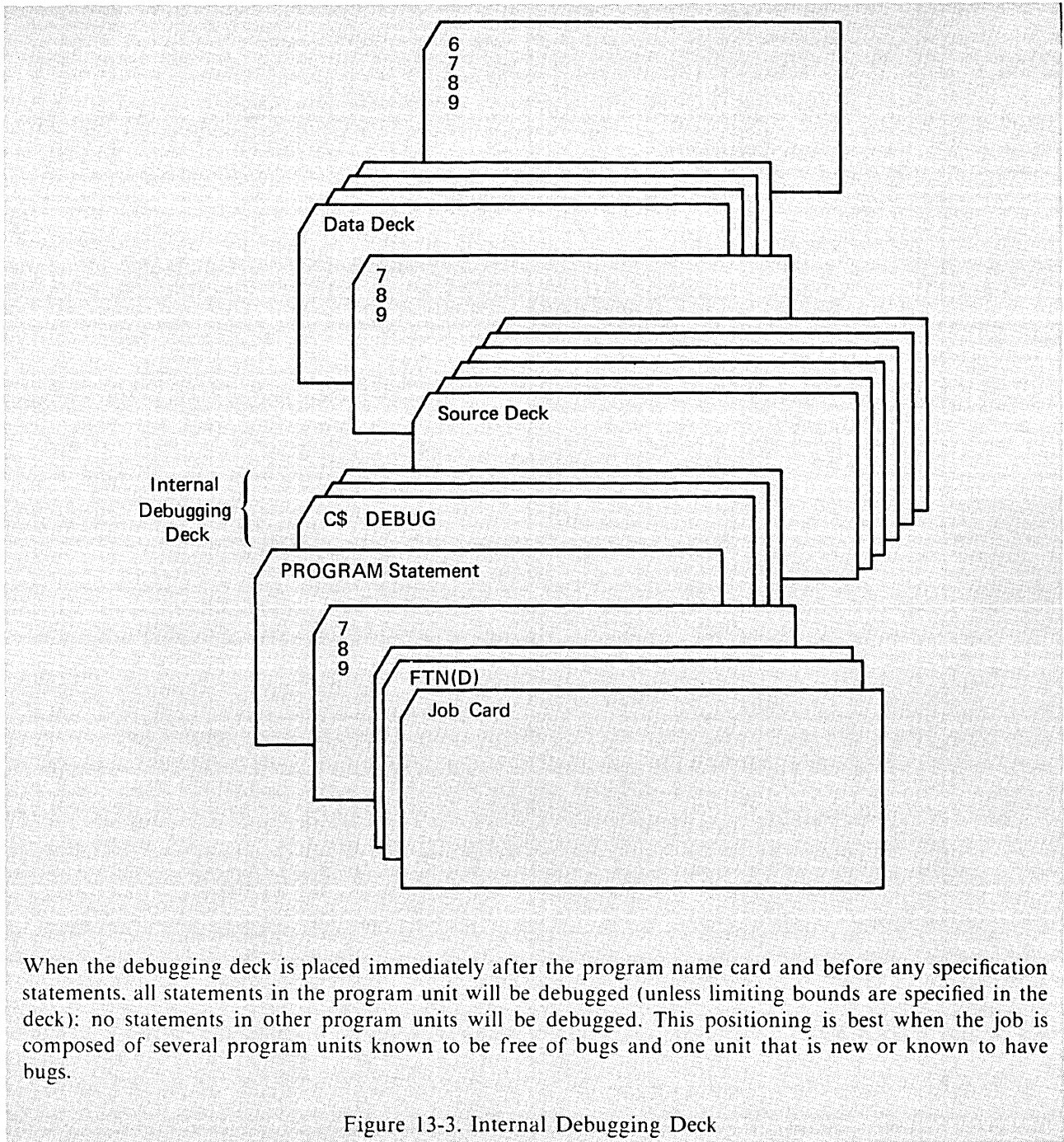
Debugging cards are interspersed; they are inserted at the point in the program where they will be activated.

Figure 13-1. Example of Interspersed Debugging Statements



The external debugging deck is placed immediately in front of the first source line. All program units (here, Program A and Subroutine B) will be debugged (unless limiting bounds are specified in the deck). This positioning is particularly useful when a program is to be run for the first time, since it ensures that all program units will be debugged.

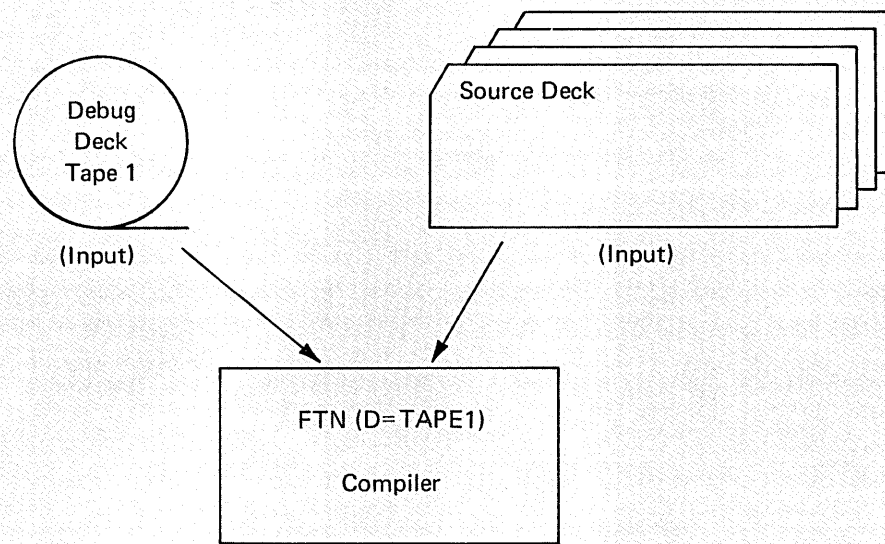
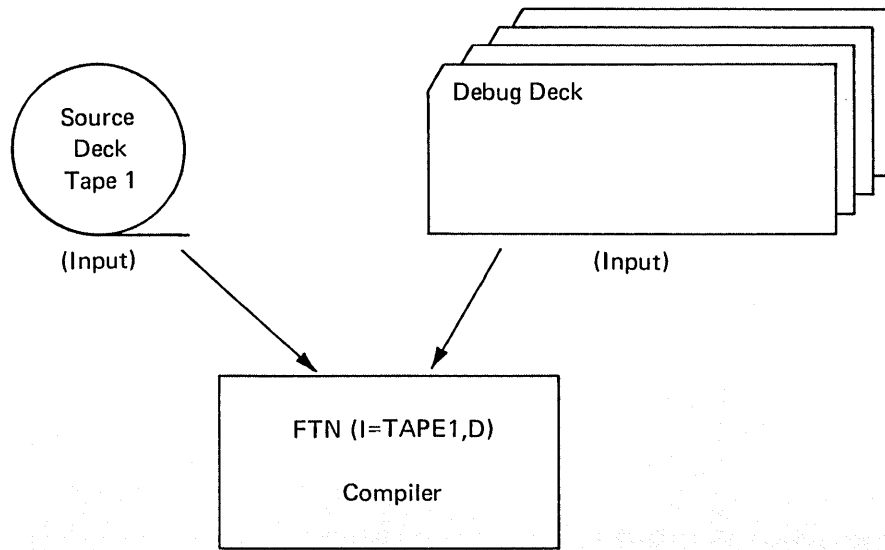
Figure 13-2. External Debugging Deck



When the debugging deck is placed immediately after the program name card and before any specification statements, all statements in the program unit will be debugged (unless limiting bounds are specified in the deck); no statements in other program units will be debugged. This positioning is best when the job is composed of several program units known to be free of bugs and one unit that is new or known to have bugs.

Figure 13-3. Internal Debugging Deck

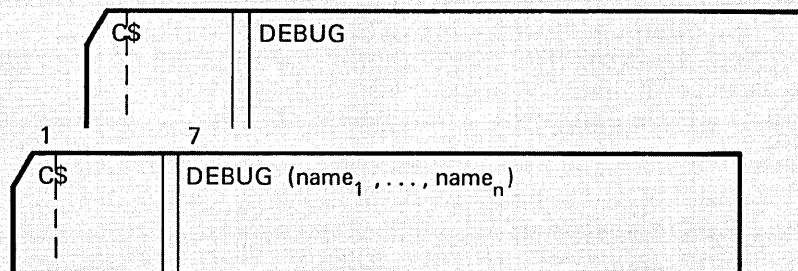




The debugging deck is placed on a separate file (external debugging deck) named by the D parameter on the FTN control card and called in during compilation. All program units will be debugged (unless the program units to be debugged are specified in the deck). This positioning is useful when several jobs can be processed using the same debugging deck.

Figure 13-4. External Deck on Separate File

## DEBUG STATEMENT



name<sub>1</sub>, ..., name<sub>n</sub> routines to which the debugging deck applies

Internal and external debugging decks start with a DEBUG statement and end with the first card other than a debugging statement or comment. Interspersed debugging statements do not require a DEBUG statement.

In an internal debugging deck, the first form CS DEBUG statement without an argument list is generally used, since the deck can apply only to the program unit in which it appears. If a name is specified it must be the name of the routine containing the debugging deck; if any other name is specified, an informative diagnostic is printed.

In an external debugging deck, if no names are specified, the deck applies to all routines compiled. Otherwise, it will apply to only those program units specified by name<sub>1</sub>, ..., name<sub>n</sub>; if any other name is specified, an informative diagnostic is printed.

Example:

In the following program, a DEBUG statement is not required since the debugging statement, CS STORES (A,B), is interspersed.

```

PROGRAM STORAR (INPUT,OUTPUT,DEBUG=OUTPUT)
REAL A(10), B(4,2)
  CS STORES (A,B)
5  B(1,2) = 5.5
   B(4,2) = 0.
   DO 4 N = 1,3
     4 A(N) = N+1
     PRINT 5
     5 FORMAT (1H0)
10  STOP
    END

```

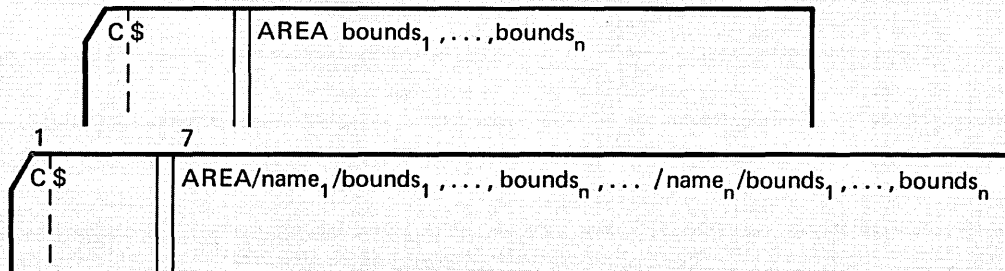
However, if the C\$ STORES statement immediately follows the PROGRAM statement, this is an internal debugging deck, and a C\$ DEBUG statement must appear.

```
                    PROGRAM DEHOL (INPUT,OUTPUT,DEBUG=OUTPUT)
                    C$  DEBUG
                    C$  STORES(IHOL,IRIGHT,ILEFT,HOLL)
5
                    IHOL=2HPA
                    IRIGHT=2RPA
                    ILEFT=2LPA
                    HOLL=2HPA
10                   PRINT 1
                    1  FORMAT (1H0)
                    STOP
                    END
```

There can be several DEBUG statements in an external deck, and a routine can be mentioned more than once.

```
C$  DEBUG
C$  STORES(I,J)
C$  DEBUG(MAIN,EXTRA,NAMES)
C$  ARRAYS(VECTAB,MLTAB)
C$  DEBUG(MAIN)
C$  TRACE
C$  CALLS(EXTRA,NAMES)
```

## AREA STATEMENT



C\$ AREA(bounds<sub>1</sub>,...,bounds<sub>n</sub>) is used in internal debugging decks only.

name<sub>1</sub>,name<sub>2</sub>,...,name<sub>n</sub> are the names of routines to which the bounds apply.

bounds are line positions defining the area to be debugged.

bounds can be written in one of the following forms:

(n <sub>1</sub> ,n <sub>2</sub> )	n <sub>1</sub>	Initial line position.
	n <sub>2</sub>	Terminal line position.
(n <sub>3</sub> )	n <sub>3</sub>	Single line position to be debugged.
(n <sub>1</sub> ,*)	n <sub>1</sub>	Initial line position.
	*	Last line of program.
(* ,n <sub>2</sub> )	*	First line of program.
	n <sub>2</sub>	Terminal line position.
(* ,*)	*	First line of program.
	*	Last line of program.

Line positions can be:

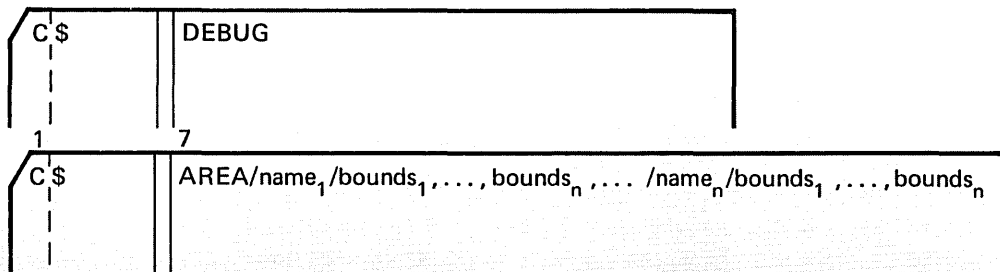
nnnnn	Statement label.
Lnnnn	Source program line number as printed on the source listing by the FORTRAN Extended compiler (source listing line numbers change when debugging cards are interspersed in the program).
id.n	UPDATE line identifier (defined in the UPDATE Reference Manual); id must begin with an alphabetic character and contain no special characters.

A comma must be used to separate the line positions, and embedded blanks are not permitted. Any of the line position forms can be combined and bounds can overlap.

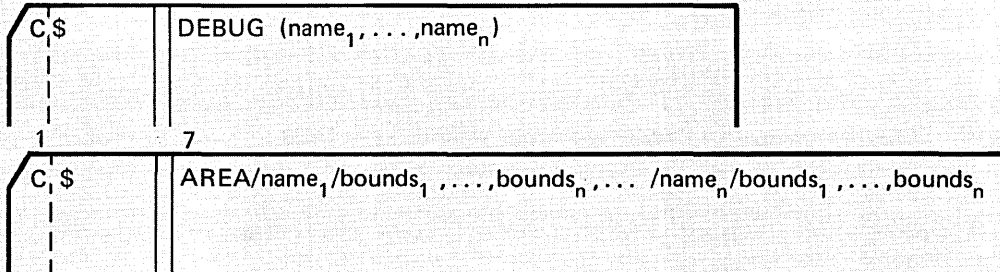
The AREA statement is used to specify an area to be debugged within a program unit. All debugging statements applicable to the program areas designated by the AREA statement must follow that statement. Each AREA statement cancels the preceding program AREA statement. An AREA statement (or contiguous set of AREA statements) specifies bounds for all debugging statements that occur between it and the next C\$ DEBUG, AREA statement, or FORTRAN source statement.

AREA statements may appear only in an external or an internal debugging deck (figures 13-2, 13-3, and 13-4). If they are interspersed in a FORTRAN source deck, they will be ignored.

In an external debugging deck, the following form, with /name<sub>i</sub>/ specified, must be used. It can be used with both forms of the DEBUG statement.

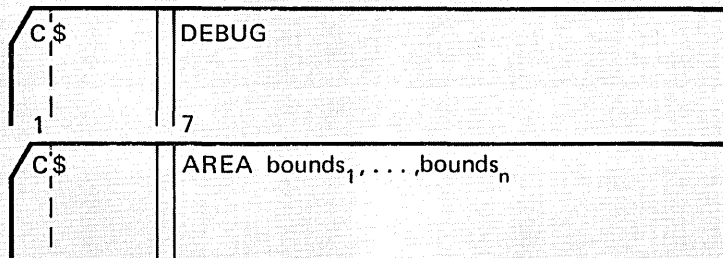


or



If /name<sub>i</sub>/ is omitted, or names in the /name<sub>i</sub>/ list do not appear in (name<sub>1</sub>,...,name<sub>n</sub>) in the DEBUG statement, the AREA statement is ignored.

In an internal debugging deck, the following form is used, and the bounds apply to the program unit that contains the deck.



Example:

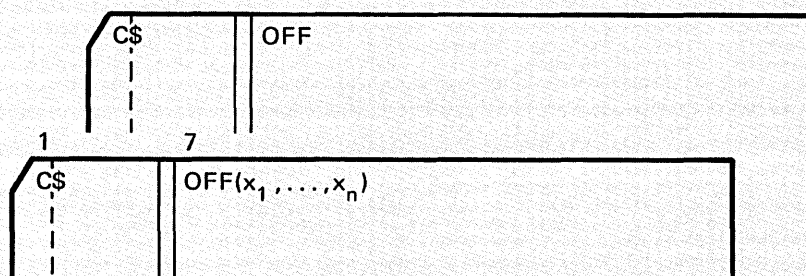
#### External deck

```
C$  DEBUG
C$  AREA/PROGA/(XNEW.10,XNEW.30)/SUB/* ,L50)
C$  ARRAYS (TAB,TITLE,DAYS)
C$  AREA/SUB/(15,99)
C$  STORES (DAYS)
```

#### Internal deck

```
C$  DEBUG
C$  AREA (L10,*)
C$  FUNCS (ABS)
```

## OFF STATEMENT



$x_1, \dots, x_n$  debug options

The OFF statement deactivates the options specified by  $x_i$  or all currently active options except NOGO, if no argument list exists. Only options activated by interspersed debugging statements are affected. Options activated in debug decks or by subsequent debugging statements are not affected.

The OFF statement is effective at compile time only. In a debugging deck, the OFF statement is ignored.

```

PROGRAM OFF (OUTPUT,DEBUG=OUTPUT)
C$ DEBUG
C$ STORES(C)
5 C$ INTEGER A, B, C
C$ STORES(A, B)

A = 1
B = 2
C = 3
10 *
* MESSAGES WILL BE PRINTED FOR STORES INTO A, B, AND C.
*
C$ OFF
*
15 A = 4
B = 5
C = 6
* THE OFF STATEMENT WILL ONLY AFFECT THE INTERSPERSED DEBUGGING
* STATEMENT, SO THERE WILL BE NO MESSAGES FOR STORES INTO
20 * A OR B. HOWEVER, C$ STORES(C) IN THE DEBUGGING DECK IS NOT
* AFFECTED, AND A MESSAGE IS PRINTED FOR A STORE INTO C.
*
END

```

/DEBUG/	OFF	AT LINE	7-	THE NEW VALUE OF THE VARIABLE A	IS	1
/DEBUG/		AT LINE	8-	THE NEW VALUE OF THE VARIABLE B	IS	2
/DEBUG/		AT LINE	9-	THE NEW VALUE OF THE VARIABLE C	IS	3
/DEBUG/		AT LINE	17-	THE NEW VALUE OF THE VARIABLE C	IS	6

## PRINTING DEBUG OUTPUT

Debug messages produced by the object routines are written to a file named **DEBUG**. The file is always printed upon job termination, as it has a print disposition. To intersperse debugging information with output, the programmer should equate **DEBUG** to **OUTPUT** on the program card. An FET and buffer are supplied automatically at load time if the programmer does not declare the **DEBUG** file in the **PROGRAM** statement. For overlay jobs, the buffer and FET will be placed in the lowest level of overlay containing debugging. If this overlay level would be overwritten by a subsequent overlay load, the debug buffer will be cleared before it is overwritten.

At object time, printing is performed by seven debug routines coded in **FORTRAN**. These routines are called by code generated at compile time when debugging is selected.

<b>Routine</b>	<b>Function</b>
<b>BUGARR</b>	Checks array subscripts
<b>BUGCLL</b>	Prints messages when subroutines are called and when return to calling program occurs
<b>BUGFUN</b>	Prints messages when functions are called and when return to calling program occurs
<b>BUGGTA</b>	Prints a message if the target of an assigned <b>GO TO</b> is not in the list
<b>BUGSTO</b>	Performs stores checking
<b>BUGTRC</b>	Flow trace printing except for true sides of logical <b>IF</b>
<b>BUGTRT</b>	Flow trace printing for true sides of logical <b>IF</b>

## STRACE ENTRY POINT

Traceback information from a current subroutine level back to the main level is available through a call to **STRACE**. **STRACE** is an entry point in the object routine **BUGCLL**. A program need not specify the **D** option on the **FTN** card to use the **STRACE** feature.

**STRACE** output is written on the file **DEBUG**; to obtain traceback information interspersed with the source program's output, **DEBUG** should be equivalenced to **OUTPUT** in the **PROGRAM** statement.

### PROGRAM MAIN

```
PROGRAM MAIN (OUTPUT,DEBUG=OUTPUT)
CALL SUB1
END
```



## SUBROUTINE SUB1

```
SUBROUTINE SUB1
CALL SUB2
RETURN
END
```

## SUBROUTINE SUB2

```
SUBROUTINE SUB2
I = FUNC1(2)
RETURN
END
```

## FUNCTION FUNC1

```
FUNCTION FUNC1 (K)
FUNC1 = K ** 10
CALL STRACE
RETURN
END
```

Output from STRACE:

```
/DEBUG/  FUNC1  AT LINE  3- TRACE ROUTINE CALLED
          FUNC1  CALLED BY SUB2  AT LINE  2, FROM  1 LEVELS BACK
          SUB2   CALLED BY SUB1  AT LINE  2, FROM  2 LEVELS BACK
          SUB1   CALLED BY MAIN  AT LINE  2, FROM  3 LEVELS BACK
```

A main program is at level 0; a subroutine or function called by the main program is at level 1; another subprogram called by a subprogram is at level 2, etc. Calls are shown in order of ascending level number, returns in order of descending level number.

For additional information regarding the debugging facility, refer to the FORTRAN Extended Debug User's Guide.



---

## PROGRAM OUT

Program OUT illustrates the WRITE and PRINT statements.

Features:

- Control cards

- WRITE and PRINT statements

- Carriage control

- PROGRAM statement

**PAT, T10, CM45000.**

The job card must precede every job. PAT is the job name. T10 specifies a maximum of 10 (octal) seconds central processor time, and CM45000 requests 45000 (octal) words of memory for the job.

**FTN.**

Specifies the FORTRAN Extended compiler and uses the default parameters. (section 11, part 1.)

**LGO.**

The binary object code is loaded and executed.

If no alternative files are specified on the FTN card, the FORTRAN Extended compiler reads from the file INPUT and outputs to two files: OUTPUT and LGO. Listings, diagnostics, and maps are output to OUTPUT and the relocatable object code to LGO.

**7/8/9**

The end-of-record card (EOR) or end-of-section card (EOS) separates control cards from the remainder of the INPUT file. The end-of-record card is a multipunch 7/8/9 in column 1; it must follow the control cards in every job.

PROGRAM OUT (OUTPUT,TAPE6=OUTPUT)

The PROGRAM card identifies this as the main program with the name OUT and specifies the file OUTPUT. Output unit 6 will be referenced in the program. All files used by a program must be specified in the PROGRAM card of the main program.

TAPE6=OUTPUT is included because output unit 6 is referenced in a WRITE statement. The unit number must be preceded by the letters TAPE. All data written to unit 6 will be placed in the file OUTPUT and output to the printer.

WRITE (6,200) INK

The WRITE statement outputs the variable INK to output unit 6. If a PRINT statement had been used instead of WRITE:

```
PRINT 200, INK
```

TAPE6=OUTPUT would not be needed in the PROGRAM card; PROGRAM OUT (OUTPUT) would be sufficient.

100 FORMAT (\*1 THIS WILL PRINT AT THE TOP OF A PAGE\*)

This FORMAT statement uses \* \* to delimit the literal. 1 is a carriage control character which causes the line to be printed at the top of a page.

200 FORMAT (I5,\* = INK OUTPUT BY WRITE STATEMENT\*)

Although the variable INK is 4 digits, a specification of I5 is given because the first character is always interpreted as a control. In this case, the carriage control character is a blank and output will appear on the next line.

6/7/8/9

This is the end of file (EOF) or end of partition card; a multipunch 6/7/8/9 in column 1. This card must appear as the last card in each job.

PAT,T10,CM45000.

FTN.

LGO.

7/8/9 in column 1

```
PROGRAM OUT (OUTPUT,TAPE 6=OUTPUT)
PRINT 100
100 FORMAT (*1 THIS WILL PRINT AT THE TOP OF A PAGE*)
   INK = 2000+4000
   WRITE (6,200) INK
200 FORMAT (I5,* = INK OUTPUT BY WRITE STATEMENT*)
   PRINT 300, INK
300 FORMAT (1H ,I4,30H = OUTPUT FROM PRINT STATEMENT)
STOP
END
```

6/7/8/9 in column 1

Output:

```
THIS WILL PRINT AT THE TOP OF A PAGE
6000 = INK OUTPUT BY WRITE STATEMENT
6000 = OUTPUT FROM PRINT STATEMENT
```

## PROGRAM B

Program B generates a table of 64 characters indicating which character set is being used. The internal bit configuration of any character can be determined by its position in the table. Each character occupies six consecutive bits.

Features:

Octal constants

Simple DO loop

PRINT statement

FORMAT with H,/,I,X and A elements

The print statement PRINT1 has no input/output list; it prints out the heading at the top of the page using the information provided by the FORMAT statement on line 3. 25H specifies a Hollerith field of 25 characters, 1 is the carriage control character, and the two slashes // cause one line to be skipped before the next Hollerith field is printed. The slash at the end of the FORMAT specification skips another line before the program output is printed.

```
NCHAR= 00 01 02 03 04 05 06 07 00 00B
```

This statement places an octal constant in NCHAR. The blanks and leading zeros could be omitted without affecting the program; they are included for readability. A computer word can hold ten 6-bit characters; but since this statement uses only 8 characters, the 4 zeros at the end of the octal constant position the 8 characters into the left 48 bits of the computer word. The 8 characters are left justified so they may be printed using A format.

```
DO 3 I=1,8  
J=I-1
```

These statements output numbers 0 through 7. A DO index cannot begin with a zero.

PRINT 2, J, NCHAR

Prints out 0 through 7 (the value of J) on the left and the 8 characters in NCHAR on the right. The first iteration of the DO loop prints NCHAR as it appears on line 4. The octal value 01 is a display code A, 02 is a B, 03 is a C, etc.

NCHAR=NCHAR 10 10 10 10 10 10 10 00 00B

The octal constant 101010101010100000B is added to NCHAR; and when this is printed on the second iteration of the DO loop, the octal value 10 is printed as a display code H, 11 as I, 12 as J, etc. Compare these values with the Character Set listed in Appendix A.

BBBBB,T10,CM70000,P15.

MAP(OFF)

FTN.

LGO.

7/8/9 in column 1

PROGRAM B (OUTPUT)

PRINT 1

1 FORMAT(25H1TABLE OF INTERNAL VALUES//12H 01234567,/) NCHAR= 00 01 02 03 04 05 06 07 00 00B

DO 3 I = 1,8

J=I-1

PRINT 2, J,NCHAR

2 FORMAT(I3,1X,A8)

3 NCHAR=NCHAR+10 10 10 10 10 10 10 00 00B

STOP

END

6/7/8/9 in column 1

Output:

TABLE OF INTERNAL VALUES

01234567

0 :ABCDEFG

1 HIJKLMNO

2 PQRSTUWV

3 XYZ01234

4 56789+-\*

5 /()\$= ,.

6 =[]%#^v^

7 ^<>><?;

## PROGRAM MASK

Program MASK reads names and home states from data cards ignoring all but the first two letters of the state name. If the state name starts with the letters CA, the name is printed.

Feature:

Masking

```
1  FORMAT (1H1,5X,4HNAME,///)
   PRINT 1
```

The printer is directed to start a new page, print the heading NAME, and skip 3 lines.

```
3  READ 2,LNAME,FNAME,ISTATE,KSTOP
   IF(KSTOP.EQ.1)STOP
```

The last name is read into LNAME, first name into FNAME, and home state into ISTATE. The last card in the deck contains a one which will be read into KSTOP as a stop indicator. The IF statement on line 6 tests for the stop indicator.

```
IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.77770000000000000000B)) GO TO 3
```

The relational operator .NE. tests to determine if the first two letters read from the data card into variable ISTATE match the two letters of the Hollerith constant CA. The last eight characters (48 bits) in ISTATE are masked and the two remaining characters are compared with the word containing the Hollerith constant CA, also similarly masked. If the bit string forming one word is not identical to the bit string forming the other word, ISTATE is not equal to CA and the IF statement test is true.

The bit configuration of CALIFORNIA, the Hollerith constant CA and the mask follows:

California

Hollerith	C	A	L	I	F	O	R	N	I	A
Octal	03	01	14	11	06	17	22	16	11	01
Bit	000011	000001	001100	001001	000110	001111	010010	001110	001001	000001



Constant CA

Hollerith	C	A	blank	blank	blank	blank	blank	blank	blank	blank
Octal	03	01	55	55	55	55	55	55	55	55
Bit	000011	000001	101101	101101	101101	101101	101101	101101	101101	101101

Mask

Octal	77	77	00	00	00	00	00	00	00	00
Bit	111111	111111	000000	000000	000000	000000	000000	000000	000000	000000

When the masking expression (ISTATE.AND.7777000000000000000B) is completed, the first two characters of CALIFORNIA remain the same and last eight characters are zeroed out. The AND operation follows:

000011	000001	001100	001001	000110	001111	010010	001110	001001	000001
111111	111111	000000	000000	000000	000000	000000	000000	000000	000000
000011	000001	000000	000000	000000	000000	000000	000000	000000	000000

When (2HCA.AND.7777000000000000000B) is evaluated, the same result is obtained. Thus, in both words, all bits but those forming the first two characters will be masked, making a valid basis for comparing the first two characters of both words. If the result of the mask is true, the last name and first name are printed (statement 10), otherwise the next card is read.

```

PROGRAM MASK (INPUT,OUTPUT)
1  FORMAT (1H1,5X,4HNAME,///)
   PRINT 1
2  FORMAT (3A10,11)
3  READ 2,LNAME,FNAME,ISTATE,KSTOP
   IF (KSTOP.EQ.1)STOP

C IF FIRST TWO CHARACTERS OF ISTATE NOT EQUAL TO CA READ NEXT CARD

   IF((ISTATE.AND.77770000000000000000B).NE.(2HCA.AND.77770000000000
11  K00000B)) GO TO 3
   FORMAT(5X,2A10)
10  PRINT 11,LNAME,FNAME
   GO TO 3
   END

```

Data cards:

BROWN,	PHILLIP M.	CA
BICARDI,	R. J.	KENTUCKY
CROWN,	SYLVIA	CAL
HIGENBERF,	ZELDA	MAINE
MUNCH,	GARY G.	CALIF.
SMITH	SIMON	CA
DEAN	ROGER	GEORGIA
RIPPLE	SALLY	NEW YORK
JONES	STAN	OREGON
HEATH	BILL	NEW YORK

1

Output:

NAME

BROWN,	PHILLIP M.
CROWN,	SYLVIA
MUNCH,	GARY G.
SMITH	SIMON

## PROGRAM EQUIV

Program EQUIV places values in variables that have been equivalenced and prints these values using the NAMELIST statement.

Features:

EQUIVALENCE statement

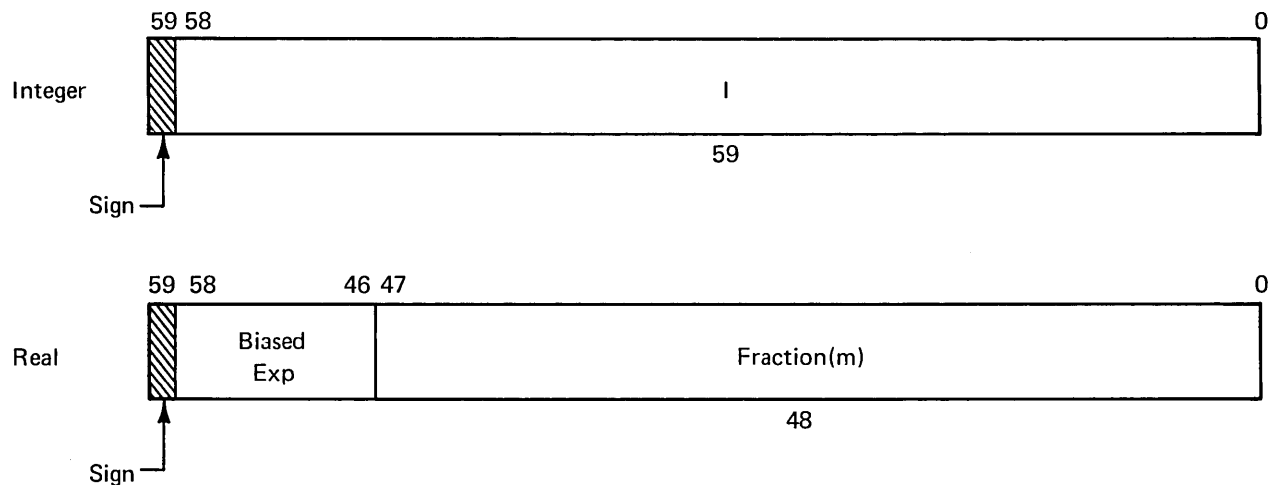
NAMELIST statement

EQUIVALENCE (X,Y),(Z,I)

Two real variables X and Y are equivalenced; the two variables share the same location in storage, which can be referred to as either X or Y. Any change made to one variable changes the value of the others in an equivalence group as illustrated by the output of the WRITE statement, in which both X and Y have the value 2. The storage location shared by X and Y contained first 1. (X=1.) then 2. (Y=2.).

The real variable Z and the integer variable I are equivalenced, and the same location can be referred to as either real or integer. Since integer and real internal formats differ, however, the output values will not be the same.

For example, the storage location shared by Z and I contained first 3. then the integer value 4. When I is output, no problem arises; an integer value is referred to by an integer variable name. However, when this same integer value is referred to by a real variable name, the value 0.0 is output. The internal format of real and integer values differ.



Although they can be referred to by names of different types, the internal bit configuration does not change. An integer value output as a real variable does not have an exponent and its value will be small.

When variables of different types are equivalenced, the value in the storage location must agree with the type of the variable name; or unexpected results may be obtained.

```
WRITE(6,OUTPUT)
```

This NAMELIST WRITE statement outputs both the name and the value of each member of the NAMELIST group OUTPUT defined in the statement NAMELIST/OUTPUT/X,Y,Z,I. The NAMELIST group is preceded by the group name, OUTPUT, and terminated by the characters \$END.

```
PROGRAM EQUIV (OUTPUT,TAPE6=OUTPUT)
EQUIVALENCE (X,Y),(Z,I)
NAMELIST/OUTPUT/X,Y,Z,I
X=1.
Y=2.
Z=3.
I=4
WRITE(6,OUTPUT)
STOP
END
```

Output:

```
$OUTPUT
X      = .2E+01,
Y      = .2E+01,
Z      = 0.0,
I      = 4,
$END
```

## PROGRAM COME

Program COME places variables and arrays in common and declares another variable and array equivalent to the first element in common. It places the numbers 1 through 12 in each element of the array A and outputs values in common using the NAMELIST statement.

Features:

COMMON and EQUIVALENCE statements

NAMELIST statement

```
COMMON A(1),B,C,D, F,G,H
```

Variables are stored in common in the order of appearance in the COMMON statement A(1),B,C,D,F,G,H. Variables can be dimensioned in the COMMON statement; and in this instance, A is dimensioned so that it can be subscripted later in the program. If A were not dimensioned, it could not be used as an array in statement 1.

```
INTEGER A,B,C,D,E(3,4),F,H
```

All variables with the exception of G are declared integer. G is implicitly typed real.

```
EQUIVALENCE(A,E,I)
```

The EQUIVALENCE statement assigns the first element of the arrays A and E and an integer variable I to the same storage location. Since A is in common, E and I will be in common. Variables and array elements are assigned storage as follows:

Relative Address	0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
I												
E(1,1)	E(2,1)	E(3,1)	E(1,2)	E(2,2)	E(3,2)	E(1,3)	E(2,3)	E(3,3)	E(1,4)	E(2,4)	E(3,4)	
A(1)	B	C	D	F	G	H						
	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)	A(11)	A(12)	

```

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

```

The DO loop places values 1 through 12 in array A. The first element of array A shares the same storage location with the first element of array E. Since B is equivalent to E(2,1), A(2) is equivalent to B, A(3) to C, A(4) to D, etc.

Any change made to one member of an equivalence group changes the value of all members of the group. When 1 is stored in A, both E(1,1) and I have the value 1. When 2 is stored in A(2), B and E(2,1) have the value 2. Although B and E(2,1) are not explicitly equivalenced to A(2), equivalence is implied by their position in common.

The implied equivalence between the array elements and variables is illustrated by the output.

```

NAMELIST/V/A,B,C,D,E,F,G,H,I

```

The NAMELIST statement is used for output. A NAMELIST group, V, containing the variables and arrays A,B,C,D,E,F,G,H,I is defined. The NAMELIST WRITE statement, WRITE(6,V), outputs all the members of the group in the order of appearance in the NAMELIST statement. Array E is output on one line in the order in which it is stored in memory. There is no indication of the number of rows and columns (3,4).

G is equivalent to E(3,2) and yet the output for E(3,2) is 6 and G 0.0. G is type real and E is type integer. When two names of different types are used for the same element, their values will differ because the internal bit configuration for type real and type integer differ (refer to Program EQUIV).

```

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

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

      WRITE (6,V)
      STOP
      END

```

Output:

```
$V
A      =  1,
B      =  2,
C      =  3,
D      =  4,
E      =  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
F      =  5,
G      =  0.0,
H      =  7,
I      =  1,
$END
```

## PROGRAM LIBS

Program LIBS illustrates library subroutines provided by FORTRAN Extended.

Features:

EXTERNAL used to pass a library subroutine name as a parameter to another library routine.

Division by zero.

LEGVAR used to test for overflow or divide error conditions.

Library functions used:

LOCF

LEGVAR

Library subroutines used:

DATE

TIME

SECOND

RANGET

DATE is a library subroutine which returns the date entered by the operator from the console. DATE is declared external because it is used as a parameter to the function LOCF. Declaring DATE external does not prevent its use as a library subroutine in this program.

```
PRINT2, TODAY, CLOCK  
2  FORMAT(*1TODAY=*Y, A10, * CLOCK=*, A10)
```

These statements print the date and time. The leading and trailing blanks appear with the 10 alphanumeric characters returned by the subroutine DATE because the operator typed in the date this way. However, since he may choose to use a 4-digit year, it may be prudent to use A11 in the output FORMAT specification to guarantee at least one leading space. The value returned by TIME is changed by the system once a second, and the position of the digits remain fixed; a leading blank always will appear. The format of DATE and TIME can be checked by observing any listing, as the routines DATE and TIME are used by the compiler to print out the date and time at the top of compiler output listings.



CALL SECOND(TYME)

When SECOND is called, the variable name TYME is used. A variable name cannot be spelled the same as a program unit name. If Program LIBS had not called the subroutine TIME, a variable name could be spelled TIME.

LOCATN=LOCF(DATE)

DATE is not a variable name as it appears in an EXTERNAL statement.

Library function LOCF returns the address of DATE.

CALL RANGET(SEED)

Library subroutine RANGET returns the seed used by the random number generator RANF if it is called. If RANGET is called after RANF has been used, RANGET will return the value currently being processed by the random number generator. With the library subroutine RANSET, this same value could be used to initialize the random number generator at a later date.

```
PRINT3, TYME, LOCATN, LOCATN, SEED, SEED
3  FORMAT(*OTHE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*// * LOCATION OF
1  DATE ROUTINE IS=*,015,* OR*,I7,* IN DECIMAL.*//OTHE INITIAL VALUE
2  OF THE RANF SEED IS *,022,*, OR*,G30.15,* IN G30.15 FORMAT.*)
```

These statements print out the values returned by the routines SECOND, LOCF, and RANGET.

Asterisks are used to delineate Hollerith fields in the format specification to illustrate the point that excessive use of asterisks can be extremely difficult to follow.

```
Y=0.0
WOW=7.2/Y
IF(0.NE. LEGVAR(WOW))PRINT4,WOW
```

These statements illustrate the use of the library function LEGVAR within an IF statement to test the validity of division by zero. LEGVAR checks the variable WOW. This function returns a result of -1 if the variable is indefinite, +1 if it is out of range, and 0 if it is normal. Comparing the value returned by LEGVAR with 0 shows that the number is either indefinite or out of range. The output R shows the variable is out of range.

Division by zero is allowed; representation for an infinite value is given in section III-4.‡

Division by zero causes an immediate overflow condition error.§

The line of `-*-*` on the output is produced by the FORMAT specification in statement number 4: `50(2H*-)`.

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

```

PROGRAM LIBS (OUTPUT)
C
EXTERNAL DATE
C
CALL DATE (TODAY)
CALL TIME (CLOCK)

PRINT 2, TODAY, CLOCK
2 FORMAT(*1TODAY=*, A10, * CLOCK=*, A10)
C
CALL SECOND(TYME)
LOCATN=LOC(DATE)
CALL RANGET(SEED)

PRINT 3, TYME, LOCATN, LOCATN, SEED, SEED
3 FORMAT(*0THE ELAPSED CPU TIME IS*,G14.5,* SECONDS.*//* LOCATION OF
1 DATE ROUTINE IS=*,015,* OR*,I7,* IN DECIMAL.*//*THE INITIAL VALUE
2 OF THE RANF SEED IS*,022,*, OR*,G30.15,* IN G30.15 FORMAT.*)
C
Y=0.0
WOW=7.2/Y
IF(0 .NE. LEGVAR(WOW))PRINT4,WOW
STOP
4 FORMAT(1H0,50(2H*-)/ * DIVIDE ERROR, WOW PRINTS AS=*,G10.2)
END

```

Output:

```

TODAY= 07/31/74  CLOCK= 15.47.33.
THE ELAPSED CPU TIME IS 1.0030 SECONDS.
LOCATION OF DATE ROUTINE IS=0000000000005347 OR 2791 IN DECIMAL.
THE INITIAL VALUE OF THE RANF SEED IS 17171274321477413155. OR .170998394044023 IN G30.15 FORMAT.
-----
DIVIDE ERROR, WOW PRINTS AS= R

```

## PROGRAM PIE

Program PIE calculates an approximation of the value of  $\pi$ .

Feature:

Library function RANF

The random number generator, RANF, is called twice during each iteration of the DO loop, and the values obtained are stored in the variables X and Y.

```
DATA CIRCLE,DUD/2*0.0/
```

The DATA statement initializes the variables CIRCLE and DUD with the value 0.0.

Each time the DO loop is iterated, a random number, uniformly distributed over the range 0-1, is returned by the library function RANF, and this value is stored in the variable X. The value of X will be  $0 \leq X < 1$ . DUD is a dummy argument which must be used when RANF is called.

```
Y=RANF(DUD)
```

RANF is referenced again; this time to obtain a value for Y.

```
IF(X*X+Y*Y.LE.1.)CIRCLE=CIRCLE+1.
```

The IF statement and the arithmetic expression  $4 * \text{CIRCLE} / 10000$ . calculate an approximation of the value of  $\pi$ . The value of  $\pi$  is calculated using Monte Carlo techniques. The IF statement counts those points whose distance from CIRCLE(0.0) is less than one. The ratio of the number of points within the quarter circle to the total number of points approximates  $1/4$  of  $\pi$ . The value PI is printed by the NAMELIST statement WRITE(6,OUT)

```
PROGRAM PIE (OUTPUT,TAPE6=OUTPUT)
DATA CIRCLE,DUD/2*0.0/
NAMELIST/OUT/PI
```

```
DO 1 I = 1,10000
X=RANF(DUD)
Y=RANF(DUD)
IF (X*X+Y*Y.LE.1.) CIRCLE=CIRCLE+1.
CONTINUE
```

```
PI=4.*CIRCLE/10000.
WRITE(6,OUT)
```

```
STOP
END
```

Output:

```
SOUT
```

```
PI      = .315E+01,
```

```
SEND
```

## PROGRAM ADD

Program ADD illustrates the use of the DECODE statement. The ENCODE and DECODE statements are simpler to understand when related to the READ and WRITE statements.

Features:

DECODE statement.

### DECODE (READ)

A READ statement places the image of each card read into an input buffer. The card image occupies eight computer words, each word containing ten display code characters. Compiler routines convert the character string in the card image into floating point, integer or logical values, as specified by the FORMAT statement, and store these values in the locations associated with the variables named in the list.

With DECODE, the array specified in the DECODE statement is used as the input buffer. The number of words moved to the input buffer from the array is determined by the record length.

With the READ statement, when the FORMAT specification indicates a new record is to be processed (by a slash or the final right parenthesis of the FORMAT statement), a new record is obtained by reading another card into the input buffer.

With the DECODE statement, when the FORMAT statement indicates a new record is to be processed (by a slash or final right parenthesis), the next part of the array is used as the input buffer. The record length indicates the number of words to move forward in the array.

### ENCODE (WRITE)

A WRITE statement causes the output buffer to be cleared to spaces (effectively). Data in the WRITE statement list is converted into a character string according to the format specified in the FORMAT statement, and placed in the output buffer. When the FORMAT statement indicates an end of a record with either a slash or the final right parenthesis, the character string is passed from the output to the output system; the output buffer area is reset to spaces (effectively), and the next string of characters is placed in the buffer.

The ENCODE statement is processed by compiler routines in the same way as the WRITE statement, but with the array specified within the parentheses of the ENCODE statement used as the output buffer. The number of words per record in the array is determined by the record length.

The number of computer words in each ENCODE or DECODE record is determined by dividing the record length by 10 and rounding up. For example, a record length of 33 requires 4 words, and a record length of 71 requires 8 words.

As a mnemonic aid, it may be useful to remember READ ends with a D and corresponds to DECODE, WRITE ends with an E and corresponds to ENCODE.

In the following program, the format of data on the input cards is specified in column 1. If column 1 is a one, each of the remaining columns is a data item. If column 1 is a two, each pair of the remaining columns is a data item. If column 1 is a three or greater, each triplet of the remaining columns is a data item. Based on the information in column 1, the correct DECODE statement (the proper format and item count) are selected. The program then totals and prints out the items in each input card.

```

PROGRAM ADD                                000
1 (INPUT,OUTPUT,TAPES=INPUT,TAPE6=OUTPUT) 001
INTEGER CARD(8),IN(79),TOTAL              002
10 READ(5,11)KEY,CARD                      003
11 FORMAT(11,7A10,A9)                     004
IF (EOF(5).NE.0)STOP                       005
KEY=MAX0(1,MIN0(KEY,3))                   006
GOTO(1,2,3),KEY                            007
1 DECODE(79,91,CARD)IN                     008
91 FORMAT(79I1)                             009
N=79                                        010
GOTO40                                      011
2 DECODE(78,92,CARD)(IN(I),I=1,39)        012
92 FORMAT(39I2)                             013
N=39                                        014
GOTO40                                      015
3 DECODE(78,93,CARD)(IN(I),I=1,26)        016
93 FORMAT(26I3)                             017
N=26                                        018
40 TOTAL=0                                  019
DO41I=1,N                                  020
41 TOTAL=TOTAL+IN(I)                       021
WRITE(6,12)TOTAL,N,KEY,CARD,(IN(I),I=1,N) 022
12 FORMAT(/16,20H IS THE TOTAL OF THE ,13,20H NUMBERS ON THE CARD/, 023
112,7A10,A9/16H THE NUMBERS ARE/(20I4))    024
GOTO10                                      025
END                                         026
7/8/9 IN COLUMN 1.                         027
21322554766988775533210332245666877965541233322112365478965412365547896541236028
30214456699877456632214455666655233655222144455663325566699885666554778854887029
55566663223666552332214455666998877655222144455611223303324456669988774558896030
10234566688899887789965554444556665533222111233023333669985555222114444777885031
000

```

Output:

```
1900 IS THE TOTAL OF THE 39 NUMBERS ON THE CARD
21322554766986775533210332245666877965541233322112365478965412365547896541236028
THE NUMBERS ARE
13 22 55 47 66 98 87 75 53 32 10 33 22 45 66 68 77 96 55 41
23 33 22 11 23 65 47 89 65 41 23 65 54 78 96 54 12 36 2
```

```
14380 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
30214456699877456632214455666655233655222144455663325566699885666554778854887029
THE NUMBERS ARE
21 445 669 987 745 663 221 445 566 665 523 365 522 214 445 566 332 556 669 988
566 655 477 885 488 702
```

```
13840 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
355666632236665523322144556669988776552221444556611223303324456669988774558896030
THE NUMBERS ARE
556 666 322 366 655 233 221 445 566 699 887 765 522 214 445 561 122 330 332 445
666 998 877 455 889 603
```

```
370 IS THE TOTAL OF THE 79 NUMBERS ON THE CARD
1023456668889988778996555444455666553322211123302333669985555222114444777885031
THE NUMBERS ARE
0 2 3 4 5 6 6 6 8 8 8 9 8 8 7 7 8 9 9
6 5 5 5 4 4 4 4 5 5 6 6 5 5 3 3 2 2 2
1 1 1 2 3 3 0 2 3 3 3 6 6 9 9 8 5 5 5
5 2 2 2 1 1 4 4 4 4 7 7 7 8 8 5 0 3 1
```

INTEGER CARD(8), IN(79), TOTAL

CARD is dimensioned 8 to receive the 79 characters in columns 2 through 80. IN is dimensioned 79 to receive the numeric values of the input items.

```
10 READ(5,11)KEY,CARD
11 FORMAT(I1,7A10,A9)
```

The first column of the card is read into KEY under I format, and the remaining 79 characters are read into the array CARD under A format; so they can be converted later to I format with a DECODE statement.

```
IF (EOF(5).NE.0) STOP
```

Tests for the end of data in which case the program simply stops.

```
KEY=MAX0(1,MIN0(KEY,3))
```

Guarantees that the value of KEY is greater than zero and less than or equal to three.

```
40 TOTAL=0
DO41 I=1,N
41 TOTAL=TOTAL+IN(I)
```

Adds up to correct number of items and leaves the total in TOTAL.

```
WRITE(6,12)TOTAL,N,KEY,CARD,(IN(I),I= 1,N)
12  FORMAT(/I6,20H IS THE TOTAL OF THE ,I3,20H NUMBERS ON THE CARD/
112,7A10,A9/16H THE NUMBERS ARE/(20I4))
```

Outputs the results.

```
GOTO10
```

Goes back to process the next card.

## PROGRAM PASCAL

Program PASCAL produces a table of binary coefficients (Pascal's triangle).

Features:

- Nested DO loops

- DATA statement

- Implied DO loop

```
INTEGER L(11)
```

L is defined as an 11-element integer array.

```
DATA L(11)/1/
```

The DATA statement stores the value 1 in the last element of the array L. When the program is executed L(11) has the initial value 1.

```
PRINT 4,(I,I=1,11)
```

This statement prints the headings. The implied DO loop generates the values 1 through 11 for the column headings.

```
PRINT 3,(L(J),J=K,11)
```

This is a more complicated example of an implied DO loop. The index value J is used as a subscript instead of being printed. The end of the array is printed from a variable starting position. The 1, which appears on the diagonal in the output is not moving in the array; it is always in L(11); but the starting point is moving.



```
DO 2 I=1,10
K=11-I
```

These statements illustrate the technique of going backwards through an array. As I goes from 1 to 10, K goes from 10 to 1. The increment value in a DO statement must be positive, therefore,

```
DO 2 I=1,10
K=11-I
```

provides a legal method of writing the illegal statement DO 2 K = 10,1,-1.

```
DO 1 J=K,10
1 L(J)=L(J)+L(J+1)
```

This inner DO loop generates the line of values output by statement number 2. When control reaches statement 2, the variable J can be used again because statement number 2 is outside the inner DO loop. However, if I were used in statement 2 instead of J, the statement 2 PRINT 3,(L(I),I=K,11) would be an error. Statement 2 is inside the inner DO loop and would change the value of the index from within the DO loop. Changing the value of a DO index from inside the loop is illegal and will cause a fatal error or a never ending loop.

```
PROGRAM      FASCAL
              PROGRAM FASCAL (OUTPUT)
              INTEGER L(11)
              DATA L(11) /1/
C
5             PRINT 4, (I,I=1,11)
4             FORMAT(44F10COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-//
S11I5)
              DO 2 I = 1,10
              K=11-I
10            L(K)=1
              DO 1 J = K,10
1             L(J)=L(J)+L(J+1)
2             PRINT 3, (L(J),J=K,11)
3             FORMAT(11I5)
15
              STOP
              END
```

COMBINATIONS OF M THINGS TAKEN N AT A TIME.

		-N-									
	1	2	3	4	5	6	7	8	9	10	11
1											
2	1										
3	3	1									
4	6	4	1								
5	10	10	5	1							
6	15	20	15	6	1						
7	21	35	35	21	7	1					
8	28	56	70	56	28	8	1				
9	36	84	126	126	84	36	9	1			
10	45	120	210	252	210	120	45	10	1		
11	55	165	330	462	462	330	165	55	11	1	

## PROGRAM X

Program X references a function EXTRAC which squares the number passed as an argument.

Features:

Referencing a function

Function type

Program X illustrates that a function type must agree with the type associated with the function name in the calling program.

```
K=EXTRAC(7)
```

Since the first letter of the function name EXTRAC is E, the function is implicitly typed real. EXTRAC is referenced, and the value 7 is passed to the function as an argument. However, the function subprogram is explicitly defined integer, INTEGER FUNCTION EXTRAC(K), and the conflicting types produce erroneous results.

The argument 7 is integer which agrees with the type of the dummy argument K in the subprogram. The result 49 is correctly computed. However, when this value is returned to the calling program, the integer value 49 is returned to the real name EXTRAC; and an integer value in a real variable produces an erroneous result (refer to program EQUIV).

This problem arises because the programmer and the compiler regard a program from different viewpoints. The programmer often considers his complete program to be one unit whereas the compiler treats each program unit separately. To the programmer, the statement

```
INTEGER FUNCTION EXTRAC(K)
```

defines the function EXTRAC integer. The compiler, however, compiles integer function EXTRAC and the main program separately. In the subprogram EXTRAC is defined integer, in the main program it is defined real. Information which the main program needs regarding a subprogram must be supplied in the main program - in this instance the type of the function.

There is no way for the compiler to determine if the type of a program unit agrees with the type of the name in the calling program; therefore, no diagnostic help can be given for errors of this kind.

The second time, the program was run with EXTRAC declared integer in the calling program, and the correct result was obtained.

```

PROGRAM X (OUTPUT)
C WITH EXTRAC DECLARED INTEGER THE RESULT SHOULD BE 49, OTHERWISE IT
C WILL BE ZERO
K = EXTRAC(7)
PRINT 1, K
1 FORMAT (1H1,I5)
STOP
END

```

```

INTEGER FUNCTION EXTRAC (K)
EXTRAC = K*K
RETURN
END

```

Output:

0

```

PROGRAM      X
              PROGRAM X (OUTPUT)
              C WITH EXTRAC DECLARED INTEGER THE RESULT SHOULD BE 49, OTHERWISE IT
              C WILL BE ZERO
              INTEGER EXTRAC
5             K = EXTRAC(7)
              PRINT 1, K
              1 FORMAT (1H1,I5)
              STOP
              END

```

```

FUNCTION     EXTRAC
              INTEGER FUNCTION EXTRAC (K)
              EXTRAC = K*K
              RETURN
              END

```

Output:

49

## PROGRAM VARDIM

Program VARDIM illustrates the use of variable dimensions to allow a subroutine to operate on arrays of differing size.

Features:

Passing an array to a subroutine as a parameter.

A subroutine name used as a parameter passes the address of the beginning of the array and no dimension information.

```
COMMON X(4,3)
```

Array X is dimensioned (4,3) and placed in common.

```
REAL Y(6)
```

Array Y dimensioned (6) is explicitly typed real. It is not in common.

```
CALL IOTA(X,12)
```

The subroutine IOTA is called. The first parameter to IOTA is array X, and the second parameter is the number of elements in that array, 12. The number of elements in the array rather than the dimensions (4,3) is used which is legal.

```
SUBROUTINE IOTA(A,M)  
DIMENSION A(M)
```

Subroutine IOTA has variable dimensions. Array A is given the dimension M. Whenever the main program calls IOTA, it can provide the name and the dimensions of the array; since A and M are dummy arguments, IOTA can be called repeatedly with different dimensions replacing M at each call.

```
CALL IOTA(X,12)
```

When IOTA is called by the main program, the actual argument X replaces A; and 12 replaces M.

```

      DO 1 I=1,M
1   A(I)=I

```

The DO loop places the numbers 1 through 12 in consecutive elements of array X.

```
CALL IOTA(Y,6)
```

When IOTA is called again, Y replaces A and 6 replaces M; and numbers 1 through 6 are placed in consecutive elements of array Y. Notice the type of the arguments in the calling program agree with the type of the arguments in the subroutine. X and A are real, 12 and M are integer.

Names used in the subroutine are related to those in the calling program only by their position as arguments. If a variable I was in the calling program, it would be completely independent of the variable I in the subroutine IOTA.

The WRITE statement outputs the arrays X and Y.

```

PROGRAM      VARDIM
              PROGRAM VARDIM (OUTPUT,TAPE6=OUTPUT)
              COMMON X(4,3)
              REAL Y(6)
              CALL IOTA(X,12)
5             CALL IOTA(Y,6)
              WRITE (6,100) X,Y
100          FORMAT (*1ARRAY X = *,12F6.0/*0ARRAY Y = *6F6.0)
              STOP
              END

SUBROUTINE   IOTA
              SUBROUTINE IOTA (A,M)
              IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
              STARTING AT 1
              DIMENSION A(M)
5             DO 1 I = 1,M
              1   A(I)=I
              RETURN
              END

```

Output:

```

ARRAY X =   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.  12.
ARRAY Y =   1.   2.   3.   4.   5.   6.

```

## **PROGRAM VARDIM2**

VARDIM2 is an extension of program VARDIM. Subroutine IOTA is used; in addition, another subroutine and two functions are used.

Features:

- Multiple entry points
- Variable dimensions
- EXTERNAL statement
- COMMON used for communication between program units
- Passing values through COMMON
- Use of library functions ABS and FLOAT
- Calling functions through several levels
- Passing a subprogram name as an argument

Program VARDIM2 describes the method of a main program calling subprograms and subprograms calling each other. Since the program is necessarily complex, each subprogram is described separately followed by a description of the main program.

### **SUBROUTINE IOTA**

SUBROUTINE IOTA is described in program VARDIM.

### **SUBROUTINE SET**

SUBROUTINE SET(A,M,V) places the value V into every element of the array A. The dimension of A is specified by M.

Subroutine SET has an alternate entry point INC. When SET is entered at ENTRY INC, the value V is added to each element of the array A. The dimension of A is specified by M.

The DO loop in subroutine SET clears the array to zero.

## FUNCTION AVG

This function computes the average of the first J elements of common. J is a value passed by the main program through the function PVAL.

This function subprogram is an example of a main program and a subprogram sharing values in common. The main program declares common to be 12 words and FUNCTION AVG declares common to be 100 words. Function AVG and the main program share the first 12 words in common. Values placed in common by the main program are available to the function subprogram.

The number of values to be averaged is passed to FUNCTION PVAL by the statement `AA = PVAL(12,AVG)` and function PVAL passes this number to function AVG: `PVAL = ABS(WAY(SIZE))`

```
COMMON A(100)
```

Function AVG declares common 100 so that varying lengths (less than 100) can be used in calls. In this instance, only 12 of the 100 words are used.

```
      DO 1 I=1,J  
1    AVG=AVG+A(I)
```

The DO loop adds the 12 elements in common.

```
AVG=AVG/FLOAT(J)
```

This statement finds the average. The library function FLOAT is used to convert the integer 12 to a floating point (real) number to avoid mixed mode arithmetic.

The average is returned to the statement `PVAL = ABS(WAY(SIZE))` in function PVAL.

## FUNCTION PVAL

Function PVAL references a function specified by the calling program to return a value to the calling program. This value is forced to be positive by the library function ABS.

The main program first calls PVAL with the statement `AA=PVAL(12,AVG)`, passing the integer value 12 and the function AVG as parameters.

INTEGER SIZE

PVAL declares SIZE integer - the type of the argument in the main program (integer 12) agrees with the corresponding dummy argument (SIZE) in the subprogram.

`PVAL=ABS(WAY(SIZE))`

The value of PVAL is computed. This value will be returned to the main program through the function name PVAL. Two functions are referenced by this statement; the library function ABS and the user written function AVG. The actual arguments 12 and AVG replace SIZE and WAY.

`PVAL=ABS(AVG(12))`

Function AVG is called, and J is given the value 12. The average of the first 12 elements of common are computed by AVG and returned to function PVAL. Library function ABS finds the absolute value of the value returned by AVG.

`AM=PVAL(12,MULT)`

In this statement in the main program, PVAL is referenced again. This time the function MULT replaces WAY.

## FUNCTION MULT

MULT multiplies the first and twelfth words in COMMON and subtracts the product from the average (computed by the function AVG) of the first J/2 words in common.

COMMON ARRAY(12)

Common is declared 12; MULT shares the first 12 words of common with the main program.



```
MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
```

The twelfth and first element in common are multiplied and the average of J/2 is subtracted. This is an example of a subprogram calling another subprogram - the function AVG is used to compute the average.

## **MAIN PROGRAM – VARDIM2**

The main program calls the subroutines and functions described.

```
COMMON X(4,3)
```

Twelve elements in the array X are declared to be in common.

```
REAL Y(6)
```

The real array Y is dimensioned 6.

```
EXTERNAL MULT, AVG
```

Function names MULT and AVG are declared EXTERNAL. Before a subprogram name is used as an argument to another subprogram, it must be declared in an EXTERNAL statement in the calling program. Otherwise it would be treated by the compiler as a variable name.

```
CALL SET(Y,6,0.)
```

Subroutine SET is called. The arguments (Y,6,0.) replace the dummy arguments (A,M,V).

```
    DIMENSION Y (6)  
    DO 1 I = 1,6  
1  Y(I) = 0.0
```

The array Y is set to zero. The NAMELIST output shows the 6 elements of Y contain zero.

```
CALL IOTA(X,12)
```

Subroutine IOTA is called. X and 12 replace the dummy arguments A and M

```
    DIMENSION X (12)
    DO 1 I=1,12
1   X(I) = I
```

the value of the subscript is placed in each element of the array X. Program VARDIM output shows the value of X is 1 through 12.

```
CALL INC(X,12,-5.)
```

Subroutine SET is called, this time through entry point INC. The arguments (X,12,-5.) replace the dummy arguments (A,M,V)

```
    DO 2 I=1,12
2   X(I) = X(I) + -5.
```

-5. is added to each element in the array X. Program VARDIM2 output shows X is now -4.-3.-2.-1,0,1,2,3,4,5,6,7

```
AA=PVAL(12,AVG)
```

Function PVAL is called and its value replaces AA.

```
AM=PVAL(12,MULT)
```

Function PVAL is called again with different arguments and the value replaces AM.

```
C   PROGRAM VARDIM2(OUTPUT,TAPE6=OUTPUT,DEBUG=OUTPUT)
      THIS PROGRAM USES VARIABLE DIMENSIONS AND MANY SUBPROGRAM CONCEPTS
      COMMON X(4,3)
      REAL Y(6)
      EXTERNAL MULT, AVG
      NAMELIST/V/X,Y,AA,AM
      CALL SET(Y,6,0.)
      CALL IOTA(X,12)
      CALL INC(X,12,-5.)
      AA=PVAL(12,AVG)
      AM=PVAL(12,MULT)
      WRITE(6,V)
      STOP
      END
```

```

SUBROUTINE SET (A,M,V)
C   SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
   DIMENSION A(M)
   DO 1 I=1,M
C   1 A(I)=0.0
C
   ENTRY INC
C   INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
   DO 2 I = 1,M
C   2 A(I) = A(I) + V
   RETURN
   END

```

```

SUBROUTINE IOTA (A,M)
C   IOTA PUTS CONSECUTIVE INTEGERS STARTING AT 1 IN EVERY ELEMENT OF
C   THE ARRAY A
   DIMENSION A(M)
   DO 1 I=1,M
C   1 A(I)=I
   RETURN
   END

```

```

FUNCTION PVAL(SIZE,WAY)
C   PVAL COMPUTES THE POSITIVE VALUE OF WHATEVER REAL VALUE IS RETURNED
C   BY A FUNCTION SPECIFIED WHEN PVAL WAS CALLED. SIZE IS AN INTEGER
C   VALUE PASSED ON TO THE FUNCTION.
   INTEGER SIZE
   PVAL=ABS(WAY(SIZE))
   RETURN
   END

```

```

FUNCTION AVG(J)
C   AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF CCOMMON.
   COMMON A(100)
   AVG=0.
   DO 1 I = 1,J
C   1 AVG=AVG+A(I)
   AVG=AVG/FLOAT(J)
   RETURN
   END

```

```
REAL FUNCTION MULT(J)
C   MULT MULTIPLIES THE FIRST AND TWELTH ELEMENTS OF COMMON AND
C   SUBTRACTS FROM THIS THE AVERAGE (COMPUTED
C   BY THE FUNCTION AVG) OF THE FIRST J/2 WORDS IN COMMON.
C
```

```
COMMON ARRAY(12)
MULT=ARRAY(12)*ARRAY(1)-AVG(J/2)
RETURN
E N D
```

```
SV
```

```
X   = -0.4E+01, -0.3E+01, -0.2E+01, -0.1E+01,  0.0,  0.1E+01,  0.2E+01,      0.3E+01,  0.4E+01,  0.5E+01,
0.6E+01,  0.7E+01,
```

```
Y   =  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
```

```
AA  =  0.15E+01,
```

```
AM  =  0.265E+02,
```

```
SEND
```

## PROGRAM CIRCLE

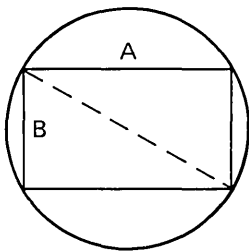
Program CIRCLE finds the area of a circle which circumscribes a rectangle.

Features:

Definition and use of both FUNCTION subprograms and statement functions.

This program has a hidden bug. We suggest you read the text from the start if you intend to find it.

A programmer wrote the following program to find the area of a circle which circumscribes a rectangle, and wrote a function named DIM to compute the diameter of the circle.



The area of a circle is  $\pi R^2$ , which is approximately the same as  $3.1416/4 * \text{Diameter}^2$ .

```
PROGRAM CIRCLE (OUTPUT)
A=4.0
B=3.0
AREA=3.1416/4.0*DIM(A,B)**2
PRINT 1, AREA
1  FORMAT(G20.10)
STOP
END
FUNCTION DIM(X,Y)
DIM=SQRT(X*X+Y*Y)
RETURN
END
```

Output:

**.7854000000**

The programmer was completely baffled by the result; he felt the area of a circle circumscribing a rectangle 12 square inches should be more than .785! He consulted another programmer who quickly pointed out that a simple function like DIM should have been written as a statement function. Since FORTRAN Extended compiles statement functions inline, it would execute much faster because no jump nor return jump would be generated by the function.

The programmer rewrote his program as follows:

```
PROGRAM CIRCLE (OUTPUT)
DIM(X,Y)=SQRT(X*X+Y*Y)
A=4.0
B=3.0
AREA=3.1416/4.0*DIM(A,B)**2
PRINT 1, AREA
1 FORMAT (G20.10)
STOP
END
```

and obtained the correct result.

When the programmer wrote his function subprogram, he used the same name as a library intrinsic function. If the name of an intrinsic function is used for a user written function, the user written function is ignored.

## PROGRAM OCON

Program OCON illustrates some problems that may occur with octal or Hollerith constants.

Features:

### Octal Constants in expressions

The compiler generally treats both octal and Hollerith constants as having no type; therefore, no mode conversion is done when they are used in expressions. If, however, the compiler is forced to assume a type for an octal or Hollerith constants, it will treat them as integer. When an expression contains only operands having no type, integer arithmetic is used. For example:

```
B=10B+10B
```

The expression is evaluated using integer arithmetic. Furthermore, for subsequent operations, the result of integer arithmetic is treated as true integer. Thus, in the above example, the expression on the right is evaluated using integer arithmetic; and the integer result is converted to real before the value is stored in B. Comparing the values produced in OCON for A and B illustrates this effect.

With REAL arithmetic whenever the left 12-bits of the computer word are all zeros or all ones, the value of that number is zero. (See section III-4 discussion of Underflow.) This explains why the output value of A from OCON is zero.

```
C=B+10B
```

REAL arithmetic is used to evaluate the expression; and the octal constant 10B is used without type conversion, making its value zero. Note in the output from OCON, the values of B and C are equal.

```
D=I+10B
```

No problem arises in the above expression as it is evaluated with integer arithmetic; then the result is converted to REAL and stored in D.

```
E=B+I+10B
```

The compiler, in scanning the above expression left to right, encounters the REAL variable B and uses REAL arithmetic to evaluate the expression. Again, the octal constant 10B has the REAL value of zero.

If the expression were written as:

```
E=10B+I+B    or    E=I+10B+B
```

The first two terms would be added using integer arithmetic; then that result would be converted to REAL and added to B. In this case, the octal constant 10B would effectively have the value eight.

This is similar to the mode conversion which occurs in:

$$X=Y*3/5 \quad \text{or} \quad Z=3/5*Y$$

The above expressions would give different values for X and Z. More information on the evaluation of mixed mode expressions is in section I-3.

```
F=A.EQ.77B
```

REAL arithmetic is used to compare the values because A is a type REAL name. The value in A and the constant 77B both have all zeros in the leftmost 12 bits; both have value zero for real arithmetic; therefore, the value assigned to F is .TRUE.

To avoid the confusion illustrated in this example, simply use integer names for values that come from octal or Hollerith constants or character data that is input using A or R format elements. To illustrate, this program was rerun with the names A, B, C, D, and E all as type INTEGER.

All these examples use octal constants; however, the same problem occurs with Hollerith, especially when it is right-justified. The following coding illustrates the point:

```
.  
. .  
REAL ANS  
. .  
READ 2, ANS  
2 FORMAT(R3)  
IF(ANS .EQ. 3RNO )PRINT3  
3 FORMAT (*-NEGATIVE RESPONSE*)  
. .  
.
```

PRINT3 of the logical IF is always executed independently of information in the data cards.



WITH REAL VARIABLES

	PROGRAM OCON(OUTPUT,TAPE6=OUTPUT)	\$OUT	
	LOGICAL F		
	NAMelist/OUT/A,B,C,D,E,F	A	= 0.0,
5	A=20B	B	= .16E+02,
	B=10B+10B	C	= .16E+02,
	C=B+10B	D	= .13E+02,
	I=5	E	= .21E+02,
	D=I+10B	F	= T,
10	E=B+I+10B		
	F=A.EQ.77B		
	WRITE(6,OUT)		
	STOP		
	END		
		\$END	

WITH INTEGER VARIABLES

	PROGRAM OCON(OUTPUT,TAPE6=OUTPUT)	\$OUT	
	INTEGER A,B,C,D,E		
	LOGICAL F	A	= 16,
	NAMelist/OUT/A,B,C,D,E,F	B	= 16,
5	A=20B	C	= 24,
	B=10B+10B	D	= 13,
	C=B+10B	E	= 29,
	I=5	F	= F,
	D=I+10B		
10	E=B+I+10B		
	F=A.EQ.77B		
	WRITE(6,OUT)		
	STOP		
	END		
		\$END	

## LIST DIRECTED INPUT/OUTPUT

List directed input/output eliminates the need for fixed data fields. It is especially useful for input since the user need not be concerned with punching data in specific columns. List directed input does not require the user to name each item as does NAMELIST input.

Used in combination, list directed input and NAMELIST output simplify program design. Such a program is easy to write, even for persons just learning the language; knowledge of the FORMAT statements is not required. This facility is particularly useful when FORTRAN programs are being run from a remote terminal.

Example:

```
H2,T10.
MAP(OFF)
FTN(R=0)
LGO.
7/8/9
PROGRAM EASY IO (INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
COMPUTE THE AREA AND RADIUS OF AN INSCRIBED CIRCLE OF ANY TRIANGLE.
REAL SIDES(3)
EQUIVALENCE (SIDES(1),A), (SIDES(2),B), (SIDES(3),C)
NAMELIST/OUT/SIDES,AREA,RADIUS
3 READ(5,*)SIDES
IF (EOF(5).NE.0) STOP
S=(A+B+C)/2.
AREA=SQRT(S*(S-A)*(S-B)*(S-C))
RADIUS=AREA/S
WRITE(6,OUT)
GOTO3
END
7/8/9
3 4 5
6,7,8
3*1
4
5
6
12.5321452, 22.4536,25
6/7/8/9
```

Output:

```
$OUT
SIDES = .3E+01, .4E+01, .5E+01,
AREA = .6E+01,
RADIUS = .1E+01,
$END
```

```
$OUT
SIDES = .6E+01, .7E+01, .8E+01,
AREA = .20333162567589E+02,
RADIUS = .19364916731037E+01,
$END
```

```
$OUT
SIDES = .1E+01, .1E+01, .1E+01,
AREA = .43301270189222E+00,
RADIUS = .28867513459481E+00,
$END
```

```
$OUT
SIDES = .4E+01, .5E+01, .6E+01,
AREA = .99215674164922E+01,
RADIUS = .13228756555323E+01,
$END
```

```
$OUT
SIDES = .125321452E+02, .224536E+02, .25E+02,
AREA = .14040422058737E+03,
RADIUS = .46812528582998E+01,
$END
```

The user may enter the three input values in whatever way is convenient for him; such as: one item per line (or card), one item per line with each item followed by a comma, all items on a single line with spaces separating each item, all items on a line with a comma and several spaces separating each item, or any combination of the foregoing. Furthermore, even though all input items are real, the decimal point is not required when input value is a whole number.



---

The cross reference map is a dictionary of all programmer created symbols appearing in a program unit, with the properties of each symbol and references to each symbol listed by source line number. The symbol names are grouped by class and listed alphabetically within the groups. The reference map begins on a separate page following the source listing of the program and the error dictionary.

## OPTIMIZING COMPILATION MODES (OPT=0,1,2)

The kind of reference map produced is determined by the R option on the control card:

- R = 0 No map
- R = 1 Short map (symbols, addresses, properties, and a DO loop map)
- R = 2 Long map (short map plus references by line number)
- R = 3 Long map and printout of common block members and equivalence classes
- R Implies R = 2

If R is not specified, the default option is R = 1; however L = 0 forces R = 0.

Fatal errors in the source program will cause certain parts of the map to be suppressed, incomplete, or inaccurate. Fatal to execution (FE) and fatal to compilation (FC) errors will cause the DO-loop map to be suppressed, and assigned addresses will be different; symbol references may not be accumulated for statements containing syntax errors.

For the long map, it may be necessary to increase field length by 1000(octal).

The number of references that can be accumulated and sorted for mapping is: field length minus 20000 (octal) minus 4 times the number of symbols. For example, in a source program containing 1000 (decimal) symbols, approximately 8000 (decimal) references can be accumulated with a field length of 50000 octal.

Examples from the cross-reference map produced by the program which follows are interspersed with the general format discussions.

The source program and the reference maps produced for both R = 1 and R = 3 follow. A complete set of maps for R = 2 is not included, but samples are shown with the discussion.

The header line that appears at the top of each page of compiler output contains: the program unit type, the compiling machine, the target machine, control card options, version and mod-level of the compiler, date, time, and page number.

# SOURCE PROGRAM

## Main Program

```

PROGRAM MAPS          74/74  OPT=1          FTN 4.4+REL.      02/28/75  09.32.57.      PAGE    1

      PROGRAM MAPS
      1 (INPUT,OUTPUT,TAPES=INPUT,TAPE6=OUTPUT)
      INTEGER SIZE1, S1, SIZE2, S2 ,STRAY
      EQUIVALENCE (SIZE1,S1), (SIZE2,S2)
      5  NAMELIST/PARAMS/SIZE1,SIZE2
      DATA S1,S2/12,12/
      100 READ (5,PARAMS)
      WRITE (6,PARAMS)
      PRINT1
      10  1  FORMAT(=0SAMPLE PROGRAM TO ILLUSTRATE THE VARIOUS COMPILER MAPS.#)
      CALL PASCAL (S1)
      PRINT2
      2  FORMAT(=0THE FOLLOWING WILL HAVE NO HEADINGS.#)
      15  CALL NOHEAD (S2)
      STOP
      END

```

## Block Data Subprogram

```

BLOCK DATA          74/74  OPT=1          FTN 4.4+REL.      02/28/75  09.32.57.      PAGE    1

      BLOCK DATA
      COMMON/ANARRAY/X(22)
      INTEGER X
      5  DATA X(22)/1/
      END

```

## Subprogram with second entry

```

SUBROUTINE PASCAL    74/74  OPT=1          FTN 4.4+REL.      02/28/75  09.32.57.      PAGE    1

      SUBROUTINE PASCAL (SIZE)
      INTEGER L (22),SIZE
      COMMON/ANARRAY/L
      PRINT4, (I,I=1,SIZE)
      5  4  FORMAT(44H0COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-//
      $22I6)
      ENTRY NOHEAD
      M=MIN0 (21,MAX0 (2,SIZE-1))
      DO21=1,M
      10  K=22-I
      L(K)=1
      DO1J=K,21
      1  L(J)=L(J)+L(J+1)
      2  PRINT3, (L(J),J=K,22)
      15  3  FORMAT (22I6)
      RETURN
      END

```

7/8/9 in column 1.

## Namelist data

```

$PARAMS
      SIZE2 = 7,
$END
6/7/8/9 in column 1.

```

R=1 MAPS

PROGRAM MAPS 74/74 OPT=1 FTN 4.4+REL. 02/28/75 09.32.57. PAGE 1

SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS  
4111 MAPS

VARIABLES	SN	TYPE	RELOCATION	4177	SIZE2	INTEGER
4176	SIZE1	INTEGER				
4175	STWAY	INTEGER	*UNDEF	4176	S1	INTEGER
4177	S2	INTEGER				

FILE NAMES	MODE	2041	OUTPUT	FMT	0	TAPES	NAME	2041	TAPE6	NAME
0	INPUT									

EXTEKNALS	TYPE	ARGS	PASCAL	1
NOHEAD		1		

NAMELISTS  
PARAMS

STATEMENT LABELS	4153	1	FMT	4166	2	FMT	0	100	INACTIVE

STATISTICS	PROGRAM LENGTH	75B	61
	BUFFER LENGTH	4103B	2115

BLOCK DATA 74/74 OPT=1 FTN 4.4+REL. 02/28/75 09.32.57. PAGE 1

SYMBOLIC REFERENCE MAP (R=1)

VARIABLES	SN	TYPE	RELOCATION
0	X	INTEGER	ARRAY ANARRAY

COMMON BLOCKS	LENGTH
ANARRAY	22

STATISTICS	PROGRAM LENGTH	0B	0
	CM LABELED COMMON LENGTH	26B	22

SUBROUTINE PASCAL 74/74 OPT=1 FTN 4.4+REL. 02/28/75 09.32.57. PAGE 1

SYMBOLIC REFERENCE MAP (R=1)

ENTRY POINTS  
27 NOHEAU 3 PASCAL

VARIABLES	SN	TYPE	RELOCATION	120	J	INTEGER	ARRAY	ANARRAY
115	I	INTEGER		0	L	INTEGER		
117	K	INTEGER		0	SIZE	INTEGER		F.P.
116	M	INTEGER						

FILE NAMES	MODE
OUTPUT	FMT

INLINE FUNCTIONS	TYPE	ARGS	MINO	INTEGER	0	INTRIN
MAX0	INTEGER	0	INTRIN			

STATEMENT LABELS	0	1	0	2	113	3	FMT
76	4	FMT					

LOOPS	LABEL	INDEX	FROM-TO	LENGTH	PROPERTIES
21		* I	4 4	4B	EXT REFS
44	2	* I	9 14	24B	EXT REFS NOT INNER
52	1	J	12 13	3B	INSTACK

COMMON BLOCKS	LENGTH
ANARRAY	22

STATISTICS	PROGRAM LENGTH	123B	83
	CM LABELED COMMON LENGTH	26B	22

R=2/R=3 MAPS

PROGRAM MAPS 74/74 OPT=1 FTN 4.4\*REL. 02/28/75 09.36.38.

SYMBOLIC REFERENCE MAP (R=3)

ENTRY POINTS	DEF LINE	REFERENCES				
4111 MAPS	1					
VARIABLES	SN	TYPE	RELOCATION	REFS		
4176 SIZE1		INTEGER		3	4	5
4177 SIZE2		INTEGER		3	4	5
4175 STRAY		INTEGER	*UNDEF	3		
4176 S1		INTEGER		3	4	11 DEFINED
4177 S2		INTEGER		3	4	14 DEFINED
FILE NAMES	MODE					
0 INPUT						
2041 OUTPUT	FMT		WRITES	9	12	
0 TAPES	NAME		READS	7		
2041 TAPE6	NAME		WRITES	6		
EXTERNALS	TYPE	ARGS	REFERENCES			
NO-HEAD		1	14			
PASCAL		1	11			
NAMELISTS	DEF LINE	REFERENCES				
PAPAMS	5	7	8			
STATEMENT LABELS	DEF LINE	REFERENCES				
4153 1	FMT	10	9			
4166 2	FMT	13	12			
0 100	INACTIVE	7				

EQUIV CLASSES	LENGTH	MEMBERS - BIAS NAME(LENGTH)
SIZE1	1	0 S1 (1)
SIZE2	1	0 S2 (1)

← omitted from R=2 map

STATISTICS			
PROGRAM LENGTH	75R	61	
BUFFER LENGTH	4103R	2115	

BLOCK DATA 74/74 OPT=1 FTN 4.4\*REL. 02/28/75 09.36.38. PAGE 2

SYMBOLIC REFERENCE MAP (R=3)

VARIABLES	SN	TYPE	RELOCATION	REFS			
0 X		INTEGER	ARRAY ANARRAY	2	3	DEFINED	4
COMMON BLOCKS	LENGTH	MEMBERS - BIAS NAME(LENGTH)					
ANARRAY	22	0 X (22)					
STATISTICS							
PROGRAM LENGTH	0B	0					
CM LABELED COMMON LENGTH	26B	22					

← omitted from R=2 map

SUBROUTINE PASCAL 74/74 OPT=1 FTN 4.4\*REL. 02/28/75 09.36.38. PAGE 2

SYMBOLIC REFERENCE MAP (R=3)

ENTRY POINTS	DEF LINE	REFERENCES				
27 NOHEAD	7	16				
3 PASCAL	1					
VARIABLES	SN	TYPE	RELOCATION	REFS		
115 I		INTEGER		4	10	DEFINED
120 J		INTEGER		3*13	14	DEFINED
117 K		INTEGER		11	12	DEFINED
0 L		INTEGER	ARRAY ANARRAY	2	3	2*13
116 M		INTEGER		9	8	DEFINED
0 SIZE		INTEGER	F.P.	2	4	DEFINED
FILE NAMES	MODE					
OUTPUT	FMT		WRITES	4	14	
INLINE FUNCTIONS	TYPE	ARGS	DEF LINE	REFERENCES		
MAX0	INTEGER	0 INTRIN		8		
MIN0	INTEGER	0 INTRIN		8		
STATEMENT LABELS	DEF LINE	REFERENCES				
0 1		13	12			
0 2		14	9			
113 3	FMT	15	14			
76 4	FMT	5	4			
LOOPS LABEL	INDEX	FROM-TO	LENGTH	PROPERTIES		
21	I	4 4	48	EXT REFS		
44	2	9 14	248	EXT REFS NOT INNER		
52	J	12 13	38	INSTACK		
COMMON BLOCKS	LENGTH	MEMBERS - BIAS NAME(LENGTH)				
ANARRAY	22	0 L (22)				
STATISTICS						
PROGRAM LENGTH	123B	83				
CM LABELED COMMON LENGTH	26B	22				

← omitted from R=2 map



OUTPUT

\$PARAMS

SIZE1 = 12,

SIZE2 = 7,

\$END

SAMPLE PROGRAM TO ILLUSTRATE THE VARIOUS COMPILER MAPS.

COMBINATIONS OF M THINGS TAKEN N AT A TIME.

-N-											
1	2	3	4	5	6	7	8	9	10	11	12
2	1										
3	3	1									
4	6	4	1								
5	10	10	5	1							
6	15	20	15	6	1						
7	21	35	35	21	7	1					
8	28	56	70	56	28	8	1				
9	36	84	126	126	84	36	9	1			
10	45	120	210	252	210	120	45	10	1		
11	55	165	330	462	462	330	165	55	11	1	
12	66	220	495	792	924	792	495	220	66	12	1

THE FOLLOWING WILL HAVE NO HEADINGS.

2	1										
3	3	1									
4	6	4	1								
5	10	10	5	1							
6	15	20	15	6	1						
7	21	35	35	21	7	1					

### General Format:

Each class of symbol is preceded by a subtitle line that specifies the class and the properties listed.

Formats for each symbol class are different, but printouts contain the following information:

The octal address associated with each symbol relative to the origin of the program unit.

Properties associated with the symbol

List of references to the symbol (for R=2 and R=3 only)

All line numbers in the reference list refer to the line of the statement in which the reference occurs. Multiple references in a statement are printed as n\*i where n is the number of references on line i.

All numbers to the right of the name are decimal integers unless they are suffixed with B to indicate octal.

Names of symbols generated by the compiler (such as system library routines called for input/output) do not appear in the reference map.

### ENTRY POINTS

Entry point names include program and subprogram names and names appearing in ENTRY statements. The format of this map is:

	ENTRY POINTS	DEFINITION	REFERENCES
	addr name	def	ref
<b>addr</b>	Relative address assigned to the entry point.		
<b>name</b>	Entry point name as defined in FORTRAN source.		
<b>def</b>	Line number on which entry point name is defined (PROGRAM statement, SUBROUTINE statement, ENTRY statement, etc.). (Not on R=1 maps.)		
<b>ref</b>	In subprograms only, line number of RETURN statements. (Not on R=1 maps.)		

R=1:

```
ENTRY POINTS      3 PASCAL
 27 NOHEAD
```

R=2 and R=3:

```
ENTRY POINTS      DEF LINE  REFERENCES
 27 NOHEAD         7        16
 3 PASCAL          1
```

## VARIABLES

Variable names include local and COMMON variables and arrays, formal parameters, RETURNS names, and for FUNCTION subprograms, the defined function name when used as a variable. The format of this map is:

VARIABLES	SN	TYPE	RELOCATION
addr      name	*	type      prop	block      refs
<b>addr</b>	Relative address assigned to variable <b>name</b> . If <b>name</b> is a member of a COMMON block, <b>addr</b> is relative to the start of <b>block</b> .		
<b>name</b>	Variable name as it appears in FORTRAN source listing. Variables are listed in alphabetical order.		
<b>*</b>	SN = stray name flag. (No entry appears under SN when R=1 is specified.) Variable names that appear only once in a subprogram are indicated by * under the SN headline. Such variable names are likely keypunch errors, misspellings, etc. In the long map, DO loops where the index variable is not referenced cause the index variable to be flagged as a (legal) stray name.		
<b>type</b>	LOGICAL, INTEGER, REAL, COMPLEX, or DOUBLE. Gives the arithmetic mode associated with the variable <b>name</b> . RETURNS appears if <b>name</b> is a RETURNS formal parameter.		
<b>prop</b>	Properties associated with variable <b>name</b> and printed by keywords in this column: <b>*UNDEF</b> Variable <b>name</b> has not been defined. A variable is defined if any of the following conditions holds: <b>name</b> appears in a COMMON or DATA statement. is EQUIVALENCED to a variable that is defined. appears on the left side of an assignment statement at the outermost parenthesis level. is the index variable in a DO loop. appears as a stand-alone actual parameter in a subroutine or function call. appears in an input list (READ, BUFFERIN, etc.).  Otherwise, the variable is considered undefined; however variables which are used (in arithmetic expressions, etc.) before they are defined (by an assignment statement or subprogram call) are not flagged.  <b>ARRAY</b> Variable <b>name</b> is dimensioned.  <b>*UNUSED</b> <b>name</b> is an unused formal parameter.		
<b>block</b>	Name of COMMON block in which variable <b>name</b> appears. If blank, <b>name</b> is a local variable. <b>//</b> indicates <b>name</b> is in blank COMMON. <b>F.P.</b> indicates <b>name</b> is a formal parameter.		

refs

(Does not appear in short map, R=1.)

References and definitions associated with variable name are listed by line number, beginning with the following in-line subheadings:

REFS All appearances of name in declarative statements or statements where the value of name is used.

DEFINED All appearances of name where its value may be altered such as in DATA, ASSIGN, READ, ENCODE, or DECODE, BUFFER IN, assignment statements, or as a DO loop index.

IO REFS All appearances of name in use as a variable file name in I/O statements.

R=1: This map form uses a double column format to conserve space. Headings appear only on the first columns.

VARIABLES	SN	TYPE	RELOCATION
115 I		INTEGER	120 J
117 K		INTEGER	0 L
116 H		INTEGER	0 SIZE

R=2 and R=3:

VARIABLES	SN	TYPE	RELOCATION	REFS	DEFINED	IO REFS	IO DEFINED
115 I		INTEGER		4	10		4
120 J		INTEGER		REFS 3*13	14	DEFINED	12
117 K		INTEGER		REFS 11	12	DEFINED	14
0 L		INTEGER	ARRAY ANARRAY	REFS 2	3	2*13	14
116 H		INTEGER		REFS 9	DEFINED 0		DEFINED 10
0 SIZE		INTEGER	F.P.	REFS 2	4	0	DEFINED 1

## FILE NAMES

File names include those explicitly defined in the PROGRAM header card as well as those implicitly defined (in subprograms) through usage in I/O statements. The format of this map is:

	FILE NAMES	MODE
	addr name	mode refs
<b>addr</b>	Relative address of the file information table (FIT) associated with the file name. The file's buffer starts at <b>addr+34B</b> . This column appears only in main programs (where the file is actually defined). In subprograms, this column is blank.	
<b>name</b>	Name of the file as defined in PROGRAM statement or implied from usage in I/O statements. For example, in a subprogram, WRITE(2) implies a reference to file TAPE2.	
<b>mode</b>	Indicates the mode of the file, as implied from its usage. One of the following will be printed:	
	FMT	Formatted I/O e.g. READ(2,901)
	FREE	List Directed I/O READ(2,*)
	UNFMT	Unformatted I/O READ(2)
	NAME	Namelist Name I/O READ(2,NAMEIN)
	BUF	Buffer I/O BUFFER IN(2,0)
	MIXED	Some combination of the above.
	blank	Mode cannot be determined.
<b>refs</b>	(Does not appear in short map, R=1.) References are divided into three categories by in-line subheadings:	
	READS	followed by list of line numbers referencing file name in input operations.
	WRITES	line numbers of output operations on file name.
	MOTION	line numbers of positioning operations (REWIND, BACKSPACE, ENDFILE) on file name.

R=1:

FILE NAMES	MODE							
0 INPUT		20*1 OUTPUT	FMT	0 TAPES	NAME	20*1 TAPE6	NAME	

R=2 and R=3:

FILE NAMES	MODE			
0 INPUT				
20*1 OUTPUT	FMT	WRITES	9	12
0 TAPES	NAME	READS	7	
20*1 TAPE6	NAME	WRITES	8	

When a variable is used as a unit number in an I/O statement the following message is printed:

VARIABLE USED AS FILE NAMES, SEE ABOVE

## EXTERNAL REFERENCES

External references include names of functions or subroutines called explicitly from a program or subprogram, as well as names declared in an EXTERNAL statement. Implicit external references, such as those called by certain FORTRAN source statements (READ, ENCODE, etc.) are not listed. The format of this map is:

	EXTERNALS	TYPE	ARGS	prop	REFERENCES
	name	type	args		refs
<b>name</b>	Name defined EXTERNAL as it appears in source listing.				
<b>type</b>	Applies to externals used as functions. Possible keywords are: <b>REAL, INTEGER, COMPLEX, DOUBLE, LOGICAL</b> Gives the arithmetic mode of external function. <b>NO TYPE</b> No specific arithmetic mode defined. Applies to certain library functions listed as externals in T mode. (T mode is implied when OPT=0 or D mode is selected.) This column will be blank for all externals used as subroutines in CALL statements.				
<b>args</b>	Number of arguments in call to external name.				
<b>prop</b>	Special properties associated with external name: <b>F.P</b> name is a formal parameter (applies only for references within a program). <b>LIBRARY</b> name is a library function called by value. In T compile modes, no LIBRARY entries appear since all references to library functions (SIN, COS, etc.) will be by name. (OPT=0 or D mode automatically implies T mode.)				
<b>refs</b>	Line number on which name is referenced. (Does not appear in short map, R=1.)				

R=1:

EXTERNALS	TYPE	ARGS		
NOHEAD		1	PASCAL	1

R=2 and R=3:

EXTERNALS	TYPE	ARGS	REFERENCES
NOHEAD		1	14
PASCAL		1	11

## INLINE FUNCTIONS

Inline functions include names of intrinsic and statement functions appearing in the subprogram. The subtitle line is:

	INLINE FUNCTIONS	TYPE	ARGS	DEF	LINE	REFERENCES
	name	mode	args	ftype	def	refs
<b>name</b>	Symbol name as it appears in the listing.					
<b>mode</b>	Arithmetic mode, NO TYPE means no conversion in mixed mode expressions.					
<b>args</b>	Number of arguments with which the function is referenced.					
<b>ftype</b>	INTRIN	Intrinsic function.				
	SF	Statement function.				
<b>def</b>	Blank for intrinsic functions; the definition line for statement functions.					
<b>refs</b>	Lines on which function is referenced.					

R=1:

```

  INLINE FUNCTIONS  TYPE  ARGS
    MAX0          INTEGER  0  INTRIN
                                MIND  INTEGER  0  INTRIN

```

R=2 and R=3:

```

  INLINE FUNCTIONS  TYPE  ARGS  DEF LINE  REFERENCES
    MAX0          INTEGER  0  INTRIN
    MIND          INTEGER  0  INTRIN
                                6
                                6

```

## NAMELISTS

	NAMELISTS	DEF LINE	REFERENCES
	name	def	refs
<b>name</b>	Namelist group name as defined in FORTRAN source.		
<b>def</b>	Line on which namelist is defined.		} (Does not appear in short map.)
<b>refs</b>	Line numbers of references to <b>name</b> .		

R=1:

```

  NAMELISTS
    PARAMS

```

R=2 and R=3:

```

  NAMELISTS  DEF LINE  REFERENCES
    PARAMS   5         7         6

```

## STATEMENT LABELS

The statement label map includes all statement labels defined in the program or subprogram. The format of this map is:

	STATEMENT LABELS	DEF LINE	REFERENCE
	addr      label      type	act      def	refs
<b>addr</b>	Relative address assigned to statement <b>label</b> . Inactive labels will have <b>addr</b> zero. Terminal statements of a DO loop also will have <b>addr</b> zero (unless referenced as the object of a transfer of control). 400 000 will be shown if no address is assigned; usually, a fatal error occurred and the final phase of compilation did not take place.		
<b>label</b>	Statement label from FORTRAN source program. Statement labels are listed in numerical order.		
<b>type</b>	One of the following keywords:		
	FMT	Statement label is a FORMAT statement.	
	UNDEF	Statement label is undefined. refs will list all references to this undefined label.	
	blank	Statement label appears on a valid executable statement.	
<b>act</b>	One of the following keywords:		
	INACTIVE	<b>label</b> is considered inactive. It may have been deleted by optimization. Inactive labels will have <b>addr</b> zero.	
	NO REFS	<b>label</b> is not referenced by any statements. This label may be removed safely from the FORTRAN source program.	
	blank	<b>label</b> is active or referenced.	
<b>def</b>	Line number on which <b>label</b> was defined. (Does not appear in short map.)		
<b>refs</b>	Line numbers on which <b>label</b> was referenced. (Does not appear in short map.)		

R=1:

```
STATEMENT LABELS
  0  1
 76  4      FMT
                0  2
                113  3      FMT
```

R=2 and R=3:

```
STATEMENT LABELS      DEF LINE  REFERENCES
  0  1                13    12
  0  2                14    9
 113  3      FMT      15    14
 76  4      FMT                5    4
```



## DO-LOOPS

The DO-loop map includes all DO loops as well as implied DO loops not in DATA statements that appear in the program and lists their properties. This map is suppressed if fatal errors have been detected in the source program or if Q was specified on the FTN control card. Loops are listed in order of appearance in the program. The format of this map is:

LOOPS	LABEL	INDEX	FROM-TO	LENGTH	PROPERTIES	
<b>fwa</b>	<b>term</b>	<b>mf</b>	<b>index</b>	<b>first-last</b>	<b>len</b>	<b>prop</b>
<b>fwa</b>	Relative address assigned to the start of loop body.					
<b>term</b>	Statement label defined as end of loop, or blank for implied DO-loops in I/O statements.					
<b>mf</b>	*	Indicates <b>index</b> is materialized (value of <b>index</b> in memory is the current value of loop count).				
	blank	Indicates <b>index</b> is not materialized ( <b>index</b> is not used directly and is updated in a register only; value in memory will not correspond to current loop count).				
<b>index</b>	Variable name used as control index for loop, as defined by DO statement.					
<b>first-last</b>	Line numbers of the <b>first</b> and <b>last</b> statements of the loop.					
<b>len</b>	Number of computer words generated for the body of the loop (octal).					
<b>prop</b>	Various keyword prints are possible, describing optimization properties of the loop:					
	OPT	Loop has been optimized.				
	INSTACK	Loop fits into instruction stack (less than or equal to 7‡ or 10§ words); likely to run two to three times as fast as a comparable loop that does not fit into the stack.				
	EXT REFS	Loop not optimized because it contains references to an external subprogram, or it is the implied loop of an I/O statement.				
	ENTRIES	Loop not optimized because it contains entries from outside its range.				
	NOT INNER	Loop not optimized because it is not the innermost loop in a nest.				
	EXITS	Loop not optimized because it contains references to statement labels outside its range.				

R=1, R=2, and R=3:

LOOPS	LABEL	INDEX	FROM-TO	LENGTH	PROPERTIES
20		* I	4	4B	EXT REFS
43	2	* I	9 14	20B	EXT REFS
52	1	J	12 13	2B	INSTACK NOT INNER

‡ Applies only to CONTROL DATA CYBER 70/Model 74 and 6600 computers.

§ Applies only to CONTROL DATA CYBER 70/Model 76, CYBER 170/Model 175, and 7600 computers.

## COMMON BLOCKS

The common block map lists common blocks and their members as defined in the source program. The format of this map is:

	COMMON BLOCKS	LENGTH	MEMBER – BIAS NAME(LENGTH)
	block	storage type    blen	bias    member    (size)
<b>block</b>	Common block name as defined in COMMON statement. //            represents blank common.		
<b>storage type</b>	Hardware type of storage device where the block is located: ECS, LCM, or blank (blank indicates CM or SCM).		
<b>blen</b>	Total length of <b>block</b> in decimal.		

If the long map is specified (R=3) the following details are printed for each member of each block:

<b>bias</b>	Relative position of <b>member</b> in <b>block</b> ; in decimal, gives the distance from the block origin.
<b>member</b>	Variable name defined as a member of <b>block</b> .
<b>size</b>	Number of words allocated for <b>member</b> .

Only variables defined as members of a common block explicitly by a COMMON statement are listed in this map. Variables which become implicit members of a common block by EQUIVALENCE statements are listed in the EQUIV CLASS map and the variable map.

R=1 and R=2:

```
COMMON BLOCKS    LENGTH
  ANARRAY        22
```

R=3:

```
COMMON BLOCKS    LENGTH    MEMBERS - BIAS NAME(LENGTH)
  ANARRAY        22        0 L        (22)
```

## EQUIVALENCE CLASSES

This map appears only when R=3 is selected. All members of an equivalence class of variables explicitly equated in EQUIVALENCE statements are listed. Variables added through linkage to common blocks are not included. The format of the map is:

	EQUIV CLASSES	LENGTH	MEMBERS – BIAS NAME (LENGTH)
	cbase    base	clen	bias member    (size)
<b>cbase</b>	Common base. A variable name appears here if the equivalence class is in a common block. In such a case, <b>cbase</b> is the variable name of the first member in that common block. *ERROR* Indicates this class is in error because more than one member is in common or the origin of the block is extended by equivalence.		
<b>base</b>	If the class is local (not in a common block), <b>base</b> is the name of the variable with the lowest address. If the class is in a common block, <b>base</b> is the name of the variable in that common block to which other variables were linked through an EQUIVALENCE statement.		
<b>clen</b>	Number of words allocated for <b>base</b> , (considered the class length).		
<b>bias</b>	Position of <b>member</b> relative to <b>base</b> ; <b>bias</b> is in decimal.		
<b>member</b>	Variable name defined as a member of an equivalence class. (Members having the same <b>bias</b> which are associated with the same <b>base</b> and thus occupy the same locations.)		
<b>size</b>	Size of <b>member</b> as defined by DIMENSION, etc.		

R=3 only:

EQUIV CLASSES	LENGTH	MEMBERS – BIAS NAME (LENGTH)
SIZE1	1	0 S1 (1)
SIZE2	1	0 S2 (1)

## PROGRAM STATISTICS

At the end of the reference map, the statistics are printed in octal and decimal. The format is:

### STATISTICS

<b>PROGRAM LENGTH</b>	Length of program including code, storage for local variables, arrays, constants, temporaries, etc., but excluding buffers and common blocks.
<b>BUFFER LENGTH</b>	Total space occupied by I/O buffers and FIT/FET.
<b>CM LABELED COMMON LENGTH</b>	Total length of common, excluding blank common, in CM‡/SCM§ and ECS‡/LCM§. Maximum of two entries.
<b>BLANK COMMON</b>	Length of blank common in CM‡/SCM§ or ECS‡/LCM§.

R=1, R=2, and R=3:

<b>STATISTICS</b>			
<b>PROGRAM LENGTH</b>		<b>1238</b>	<b>83</b>
<b>CM LABELED COMMON LENGTH</b>		<b>268</b>	<b>22</b>

## ERROR MESSAGES

The following error messages are printed if sufficient storage is not available:

CANT SORT THE SYMBOL TABLE      INCREASE FL BY NNNB

or

REFERENCES AFTER LINE NNN LOST INCREASE FL BY NNNB

## DEBUGGING (Using the Reference Map)

New Program:

The reference map can be used to find names that have been punched incorrectly as well as other items that will not show up as compilation errors. The basic technique consists of using the compiler as a verifier and correcting the FE errors until the program compiles.

Using the listing, the R=3 reference map, and the original flowcharts, the following information should be checked by the programmer:

Names incorrectly punched

Stray name flag in the variable map

Functions that should be arrays

Functions that should be inline instead of external

---

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

Variables or functions with incorrect type

Unreferenced format statements

Unused formal parameters

Ordering of members in common blocks

Equivalence classes

Existing Program:

The reference map can be used to understand the structure of an existing program. Questions concerning the loop structure, external references, common blocks, arrays, equivalence classes, input/output operations, and so forth, can be answered by checking the reference map.

## TS MODE

In TS mode, the reference map appears immediately following the source listing of the program (regardless of the BL parameter).

The kind of reference map produced is determined by the R option on the FTN control card:

R=0    No map

R=1    Short map (symbols, addresses, properties)

R=2 }  
R=3 }    Long map (short map plus references by page and line number)

R        Implies R=2

If R is omitted, an R=1 map is produced (unless L=0 is specified on the FTN control card).

The header line that appears at the top of each page of compiler output contains: the first line of the program unit, version and mod-level of the compiler, date, time, and page number.

On the following pages appear examples of a short and a long map. Portions of these maps appear in the subsequent format discussion.

R=1 MAPS

```

SUBROUTINE PASCAL(SIZE)
/7-
FTN 4.3*F383
08/01/74 14.13.03.
TS
PAGE 1
/73

SUBROUTINE PASCAL(SIZE)
INTEGER L(22),SIZE
COMMON/ANARRAY/L
PRINT4, (I,I=1,SIZE)
FORMAT(44H0COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-//
$22I6)
ENTRY NOHEAD
M=MIN0(21,MAX0(2,SIZE-1))
DO2I=1,M
K=22-I
L(K)=1
DO1J=K,21
L(J)=L(J)+L(J+1)
PRINT3,(L(J),J=K,22)
FORMAT(22I6)
RETURN
END
MAPS 026
MAPS 027
MAPS 028
MAPS 029
MAPS 030
MAPS 031
MAPS 032
MAPS 033
MAPS 034
MAPS 035
MAPS 036
MAPS 037
MAPS 038
MAPS 039
MAPS 040
MAPS 041
MAPS 042

```

-- COMMON BLOCKS --

26 /ANARRAY/

-- ENTRY POINTS --

15 NOHEAD 56 PASCAL

-- EXTERNALS --

OUTCI. OUTCR. OUTPUTE

-- STATEMENT LABELS --

.1 ID 0 .2 ID 40 .3 F 72 .4 F 62

-- VARIABLE MAP --

I	I	U	107	J	I	U	110		
K	I		111	L	I		0	/ANARRAY/	22
M	I		112	MAX0	I			INTRINSIC	
MIN0	I			NOHEAD	-		15	ENTRY	
OUTCI.	-			OUTCR.	-			EXTERNAL.	
OUTPUTE	-			PASCAL	-		56	ENTRY	
SIZE	I	AU	0						

BLOCK ADDRESS LENGTH	CODE	LITERALS	FORMATS	TEMPS	ARGS	NAMELIST	VARIABLES	BUFFERS
	3	62	62	74	74	107	107	113
	57	0	12	0	13	0	4	0
37100	STORAGE USED	MODEL 74	17 SOURCE STATEMENTS	17 SYMBOLS				
113	PROGRAM-UNIT LENGTH	COMPILATION	.136 SECONDS					

R=2, R=3 MAPS

```

SUBROUTINE PASCAL(SIZE)
  /7
  SUBROUTINE PASCAL(SIZE)
  INTEGER L(22),SIZE
  COMMON/ANARRAY/L
  PRINT*, (I,I=1,SIZE)
  4  FORMAT(4#HOCOMBINATIONS OF M THINGS TAKEN N AT A TIME, //20X,3H-N- /
  $22I6)
  ENTRY NOHEAD
  M=MIND(2I,MAX0(2,SIZE-1))
  21  DO2I=1,M
  26  K=22-I
  26  L(K)=1
  26  DO1J=K,21
  32  1  L(J)=L(J)+L(J+1)
  32  2  PRINT3,4L(J),J=K,22)
  52  3  FORMAT(22I6)
  52  RETURN
  52  END
  MAPS 123
  MAPS 124
  MAPS 125
  MAPS 126
  MAPS 127
  MAPS 128
  MAPS 129
  MAPS 130
  MAPS 131
  MAPS 132
  MAPS 133
  MAPS 134
  MAPS 135
  MAPS 136
  MAPS 137
  MAPS 138
  MAPS 139

```

-- COMMON BLOCKS --

26 /ANARRAY/

-- ENTRY POINTS --

15 NOHEAD 56 PASCAL

-- EXTERNALS --

OUTCI. OUTCR. OUTPUTE

-- STATEMENT LABELS --

.1	ID	0	1/12 D	1/13 L
.2	ID	40	1/09 O	1/14 L
.3	F	72	1/14 W	1/15 L
.4	F	62	1/04 W	1/05 L

-- VARIABLE MAP --

I	I U	107		1/04 C	1/04 W	1/09 C	1/10		
J	I U	110		1/12 C	1/13 S	1/13 S	1/13 S	1/14 C	1/14 S
K	I	111		1/10 =	1/11 S	1/12 C	1/14 C		
L	I	0	/ANARRAY/ 22	1/02 D	1/03 D	1/11 =	1/13	1/13 =	1/14 W
M	I	112		1/08 =	1/09 C				
MAX0	I		INTRINSIC	1/08 A					
MIND	I		INTRINSIC	1/08					
NOHEAD	-	15	ENTRY	1/07 E					
OUTCI.	-		EXTERNAL.	1/04 W	1/14 W				
OUTCR.	-		EXTERNAL.	1/04 W	1/04 W	1/14 W	1/14 W		
OUTPUTE	-		EXTERNAL.	1/04	1/04 W	1/14	1/14 W		
PASCAL	-	56	ENTRY	1/01 E					
SIZE	I AU	0		1/01 A	1/02 D	1/04 C	1/08 A		

BLOCK ADDRESS LENGTH	37300 113	CODE 3 57	LITERALS 62 0	FORMATS 62 12	TEMPS 74 0	ARGS 74 13	NAMELIST 107 0	VARIABLES 107 4	BUFFERS 113 0
		STORAGE USED	PROGRAM-UNIT LENGTH	MODEL 74	COMPILATION	17 SOURCE STATEMENTS	.152 SECONDS	17 SYMBOLS	48 REFERENCES

## COMMON BLOCKS

The common block map lists common blocks as defined in the source program. The format of this map is:

-- COMMON BLOCKS --

**length** /block/

**length** Length (in octal) of common block.

**block** Common block name as defined in COMMON statement.  
// represents blank common.

R=1, R=2, R=3:

-- COMMON BLOCKS --

26 /ANARRAY/

## ENTRY POINTS

This map lists names of program units, names appearing in ENTRY statements, and (for a main program) all file names defined in the PROGRAM statement. The format is:

-- ENTRY POINTS --

**addr** name

**addr** Relative address (in octal) of the entry point in the program unit.

**name** Entry point name as defined in source program.

R=1, R=2, R=3:

-- ENTRY POINTS --

15 NOWHEAD

56 PASCAL

## EXTERNAL REFERENCES

External references include names of functions or subroutines called explicitly from a program or subprogram, names declared in an EXTERNAL statement, and external references generated by the compiler. The format of this map is:

-- EXTERNALS --

**name**

**name** Name of routine externally referenced.



R=1, R=2, R=3:

-- EXTERNALS --

OUTCI. OUTCR. OUTPUT

## STATEMENT LABELS

This map includes all statement labels defined in the program or subprogram. The format is:

-- STATEMENT LABELS --

label properties addr references

**label** Statement label, preceded by a period. Labels are listed in ascending numerical order.

**properties** Properties as follows:

F label references a format statement.  
D label references a terminal statement of a DO loop.  
I label is inactive (never referenced by transfer or input/output statement).  
blank None of the above properties.

**addr** Relative address (in octal) assigned to this label. Some inactive labels will have an addr of zero.

**references** Page number, line number, and type of reference to statement label. References do not appear in the short map (R=1). The type can be:

L label appears in label field.  
D label referenced in a DO statement.  
R label referenced in a READ statement.  
W label referenced in a WRITE or PRINT statement.  
F label referenced in a FORMAT statement.  
A label referenced in an ASSIGN statement.  
blank Any other reference.

R=1:

-- STATEMENT LABELS --

.1	ID	0	.2	ID	40	.3	F	72	.4	F	62
----	----	---	----	----	----	----	---	----	----	---	----

R=2, R=3:

-- STATEMENT LABELS --

.1	ID	0	1/12 D	1/13 L
.2	ID	40	1/09 D	1/14 L
.3	F	72	1/14 W	1/15 L
.4	F	62	1/04 W	1/05 L

## VARIABLES

All symbolic names referenced in the program unit are listed here. The format of this map is:

### -- VARIABLE MAP --

name	type	properties	addr	block	length	references
<b>name</b>		Name of variable as it appears in source listing.				
<b>type</b>	Variable type:					
	I	INTEGER				
	R	REAL				
	D	DOUBLE PRECISION				
	Z	COMPLEX				
	L	LOGICAL				
	N	NAMELIST name				
	-	No type				
<b>properties</b>	Properties as follows:					
	A	Variable is used as a formal parameter.				
	U	Variable is undefined.				
	≡	Variable is equivalenced to a defined variable.				
	blank	None of the above.				
<b>addr</b>	Relative address (in octal) assigned to this variable.					
<b>block</b>	Name of common block in which variable appears, or (if no address is specified) a description of the type of symbolic name:					
	ENTRY	name is an entry point.				
	SUBROUTINE	name is a user supplied SUBROUTINE subprogram or a library utility subprogram.				
	INTRINSIC	name is an intrinsic function.				
	STAT-FUNC	name is a statement function.				
	B.E.F.	name is a basic external function.				
	FUNCTION	name is a user supplied FUNCTION subprogram.				
	EXTERNAL	name appears in an EXTERNAL statement or is a compiler generated external reference.				
<b>length</b>	Array length (in decimal) for dimensioned variables.					
<b>references</b>	Page number, line number, and type of reference to variable. References do not appear in the short map (R=1). The type can be:					
	A	Variable appears as argument to subroutine or function.				
	C	Variable appears as DO loop control variable.				
	D	Variable appears in specification statement.				
	E	Variable used as entry point.				
	F	Variable appears in IF statement.				

I Variable appears in DATA statement.  
 R Variable appears in READ statement.  
 S Variable appears as subscript.  
 W Variable appears in WRITE or PRINT statement.  
 X Variable appears as an external reference.  
 = Variable appears on the left side of an arithmetic replacement statement.  
 blank Variable appears on the right side of an arithmetic replacement statement.

R=1:

```

-- VARIABLE MAP --

I      I U  107
K      I   111
M      I   112
MINO   I   0  INTRINSIC
OUTCI. -   -  EXTERNAL.
OUTPUT= -   -  EXTERNAL.
SIZE   I AU  0

J      I U  110
L      I   0  /ANARRAY/ 22
MAXO   I   -  INTRINSIC
NOHEAD -   15  ENTRY
OUTCR. -   -  EXTERNAL.
PASCAL -   56  ENTRY
  
```

R=2, R=3:

```

-- VARIABLE MAP --

I      I U  107
J      I U  110
K      I   111
L      I   0  /ANARRAY/ 22
M      I   112
MAXO   I   -  INTRINSIC
MINO   I   -  INTRINSIC
NOHEAD -   15  ENTRY
OUTCI. -   -  EXTERNAL.
OUTCR. -   -  EXTERNAL.
OUTPUT= -   -  EXTERNAL.
PASCAL -   56  ENTRY
SIZE   I AU  0

1/04 C  1/04 W  1/09 C  1/10
1/12 C  1/13 S  1/13 S  1/13 S  1/14 C  1/14 S
1/10 =  1/11 S  1/12 C  1/14 C
1/02 D  1/03 D  1/11 =  1/13  1/13  1/13 =  1/14 W
1/08 =  1/09 C
1/08 A
1/08 I
1/07 E
1/04 W  1/14 W
1/04 W  1/04 W  1/14 W  1/14 W
1/04 W  1/04 W  1/14  1/14 W
1/01 E
1/01 A  1/02 D  1/04 C  1/08 A
  
```

## BLOCKS

The address and length of the various blocks comprising the object code are listed, regardless of the R parameter. The format of this map is:

BLOCK	CODE	LITERALS	FORMATS	TEMPS	ARGS	NAMELIST	VARIABLES	BUFFERS
ADDRESS								
LENGTH								

CODE	Generated executable code.
LITERALS	Constants used by the program.
FORMATS	Storage for format declarations.
TEMPS	Temporary storage locations generated by the compiler.
ARGS	Subprogram call argument lists.

**NAMelist**      Namelist storage.

**BUFFERS**      I/O buffers defined.

R=0, R=1, R=2, R=3:

<b>BLOCK ADDRESS LENGTH</b>	<b>CODE</b>	<b>LITFRALS</b>	<b>FORMATS</b>	<b>TEMPS</b>	<b>ARGS</b>	<b>NAMelist</b>	<b>VARIABLES</b>	<b>BUFFERS</b>
	3	62	62	74	74	107	107	113
	57	0	12	0	13	0	4	0

Diagnostic messages are produced by the FORTRAN Extended compiler during both compilation and execution to inform the user of errors in the source program, input data or intermediate results.

## COMPILATION DIAGNOSTICS

The compile time diagnostics issued by FTN are different in OPT=0,1,2 than in TS mode. The description of error message format for TS mode appears later in Compilation Diagnostics, TS Mode. The following information applies to compilation under OPT=0,1,2.

Errors detected during compilation are noted on the source listing immediately following the END statement. The format of the message is as follows:

CARD NO.	SEVERITY	DETAILS	DIAGNOSTIC
n	e	a	error message
n			Card number where error was detected. This number is assigned by the FORTRAN Extended compiler. Some declarative statement diagnostics will show the line number of the first non-declarative statement; END line number is used for undefined statement number diagnostics.
e			Indicates the type of diagnostic. In the following pages, compile time diagnostics are listed alphabetically by error type.
	I		Informative message which indicates minor syntax errors or omissions which have no effect upon compilation or execution.
	FC		When an error of this type is encountered during compilation, the remaining portion of the program is checked for syntax errors only. Program is not executed.
	FE		Error fatal to execution. Program compiles but does not execute.
	ANSI		Usage does not conform to ANSI standard FORTRAN (X3.9 - 1966). ANSI diagnostics are not listed unless the EL=A parameter is specified on the FTN control card.
a			Information in this column will differ according to the type of error encountered. For example, if the same statement label is used more than once, the label number is printed. If a message of the format cn CD n appears, cn is the column number in which the error was detected, and n is the card number.
error message			Error message printed by FORTRAN Extended compiler.

Example:

```

100 WRITE (6,8)
8  FORMAT (52H FOLLOWING IS A LIST OF PRIME NUMBERS FROM 1 TO 1000/
119X,1H1/19X,1H3)
5  101 I=5
8  A=I
102 A=SGRT(A)
103 J=A
104 DO 1 K=3,J,2
105 L=1/KEXCEEDS
10  106 IF(L*K-I)1,2,4
1  GO TO 108
107 WRITE (6,9)
5  FORMAT (I20)
2  I=I+2
15  108 IF(1000-I)7,4,3
4  WRITE (6,7)
9  FORMAT (14H PROGRAM ERROR)
7  WRITE (6,6)
20  6  FORMAT (31H THIS IS THE END OF THE PROGRAM)
109 STOP
END

```

CARD NO.	SEVERITY	DIAGNOSTIC
1	I	START. ASSUMED PROGRAM NAME WHEN NO HEADER STATEMENT APPEARS
2	FE	07 CD 3 ZERO LEVEL RIGHT PARENTHESIS MISSING. SCANNING STOPS.
3	FE	UNRECOGNIZED STATEMENT
5	FE	DUPLICATE STATEMENT LABEL
9	FE	SYMBOLIC NAME HAS TOO MANY CHARACTERS
9	FE	/ THE OPERATOR INDICATED (-,+,*,/, OR **) MUST BE FOLLOWED BY A CONSTANT, NAME, OR LEFT PARENTHESIS.
11	FE	A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT
16	FE	7 PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES
21	FE	UNDEFINED STATEMENT NUMBERS, SEE BELOW

UNDEFINED LABELS

3

ANSI A COMMENT LINE WITHIN A CONTINUED STATEMENT IS NON-ANSI.  
ANSI A RELATIONAL HAS A COMPLEX OPERAND.  
ANSI AN EXPRESSION IN AN OUTPUT STATEMENT I/O LIST IS NON ANSI USAGE.  
ANSI ARRAY NAME OPERAND NOT SUBSCRIPTED. FIRST ELEMENT WILL BE USED.  
ANSI ARRAY NAME REFERENCED WITH FEWER SUBSCRIPTS THAN DIMENSIONALITY OF ARRAY.  
ANSI ATTEMPT TO BACK UP BEFORE COLUMN ONE CAUSES POSITIONING TO BE SET AT COLUMN ONE.  
ANSI BACKING UP WITH X SPECIFICATION IS NON-ANSI.  
ANSI DOLLAR SIGN STATEMENT SEPARATOR IS NON-ANSI USAGE.  
ANSI END STATEMENT ACTING AS A RETURN IS NON-ANSI.  
ANSI ENTRY STATEMENT IS NON-ANSI.  
ANSI FLOATING POINT DESCRIPTOR EXPECTED AFTER SCALE FACTOR DESIGNATOR.  
ANSI GO TO STATEMENT CONTAINS NON-ANSI USAGES.  
ANSI HOLLERITH CONSTANT APPEARS OTHER THAN IN AN ARGUMENT LIST OF A CALL STATEMENT OR IN A DATA STATEMENT.  
ANSI HOLLERITH STRING DELIMITED BY SYMBOLS IS NON-ANSI.  
ANSI IMPLICIT STATEMENT IS NON-ANSI.  
ANSI LOGICAL OPERATOR OR CONSTANT USAGE IS NON-ANSI.  
ANSI MASKING EXPRESSION IS NON-ANSI.  
ANSI MULTIPLE REPLACEMENT STATEMENT IS NON-ANSI.  
ANSI NAMELIST STATEMENT IS NON-ANSI.  
ANSI NON-ANSI BLANK LINES OCCURRED IN THIS PROGRAM UNIT.  
ANSI NON-ANSI FORM OF DATA STATEMENT.  
ANSI NON-ANSI FORM OF TYPE DECLARATION.  
ANSI NON-STANDARD SUBSCRIPT IS NON-ANSI.  
ANSI OCCURRENCES OF ASTERISK OR DOLLAR SIGN NON-ANSI COMMENT LINES.  
ANSI OCTAL CONSTANT OR R,L FORMS OF HOLLERITH CONSTANT IS NON-ANSI.  
ANSI OMISSION OF FIELD SEPARATOR AFTER HOLLERITH STRING IS NON-ANSI.  
ANSI ONE OF THE FOLLOWING NON-ANSI FORMS HAS BEEN USED -- EW.DDE, EW.DEE, IW.Z, OW.Z.

ANSI PLUS SIGN IS A NON-ANSI CHARACTER.

ANSI PRECEDING FIELD DESCRIPTOR IS NON-ANSI.

ANSI RETURNS PARAMETERS IN CALL STATEMENT.

ANSI TAB SETTING DESIGNATOR IS NON-ANSI.

ANSI THE EXPRESSION IN AN IF STATEMENT IS TYPE COMPLEX.

ANSI THE FORMAT OF THIS END LINE DOES NOT CONFORM TO ANSI SPECIFICATIONS.

ANSI THE NON-STANDARD RETURN STATEMENT IS NON-ANSI.

ANSI THE TYPE COMBINATION OF THE OPERANDS OF AN EQUAL-SIGN OPERATOR IS NON-ANSI.

ANSI THE TYPE COMBINATION OF THE OPERANDS OF AN EXPONENT OPERATOR IS NON-ANSI.

ANSI THE TYPE COMBINATION OF THE OPERANDS OF A RELATIONAL OR ARITHMETIC OPERATOR (OTHER THAN \*\*) IS NON-ANSI.

ANSI THIS FORM OF AN I/O STATEMENT DOES NOT CONFORM TO ANSI SPECIFICATIONS.

ANSI THIS FORMAT DECLARATION IS NON-ANSI.

ANSI THIS STATEMENT TYPE IS NON-ANSI.

ANSI TWO-BRANCH IF STATEMENT IS NON-ANSI.

ANSI USE OF A NUMBER AS LABELED COMMON BLOCK NAME IS NON-ANSI.

ANSI 7 CHARACTER SYMBOLIC NAME IS NON-ANSI.

FC ERROR TABLE OVERFLOW.

FC MEMORY OVERFLOW DURING ASF EXPANSION.

FC NOT ENOUGH ROOM IN WORKING STORAGE TO HOLD ALL OVERLAY CONTROL CARD INFORMATION.

FC SYMBOL TABLE OVERFLOW.

FC TABLE OVERFLOW, INCREASE FL.

FC TABLES OVERLAP, INCREASE FL.

FC THIS SUBPROGRAM HAS TOO MANY DO LOOPS.

FE .NOT. MAY NOT BE PRECEDED BY NAME, CONSTANT, OR RIGHT PARENS.

FE + OR - SIGN MUST BE FOLLOWED BY A CONSTANT.

FE A COMMA, LEFT PAREN., =, .OR., OR .AND. MUST BE FOLLOWED BY A NAME, CONSTANT, LEFT PAREN., -, .NOT., OR +.

FE A COMPLEX BASE MAY ONLY BE RAISED TO AN INTEGER POWER.



FE A CONSTANT ARITHMETIC OPERATION WILL GIVE AN INDEFINITE OR OUT-OF-RANGE RESULT.

FE A CONSTANT CANNOT BE CONVERTED. CHECK CONSTANT FOR PROPER CONSTRUCT.

FE A CONSTANT DO PARAMETER MUST BE GREATER THAN OR EQUAL TO 1 AND LESS THAN OR EQUAL TO 131071.

FE A CONSTANT MAY NOT BE FOLLOWED BY AN EQUAL SIGN, NAME, OR ANOTHER CONSTANT.

FE A CONSTANT OPERAND OF A REAL OPERATION IS OUT OF RANGE OR INDEFINITE.

FE A DO LOOP MAY NOT TERMINATE ON A FORMAT STATEMENT.

FE A DO LOOP MAY NOT TERMINATE ON THIS TYPE OF STATEMENT.

FE A DO PARAMETER MUST BE A POSITIVE INTEGER CONSTANT OR AN INTEGER VARIABLE.

FE A FUNCTION REFERENCE REQUIRES AN ARGUMENT LIST.

FE A NAME MAY NOT BE FOLLOWED BY A CONSTANT.

FE A PREVIOUS STATEMENT MAKES AN ILLEGAL TRANSFER TO THIS LABEL.

FE A PREVIOUSLY MENTIONED ADJUSTABLE SUBSCRIPT IS NOT TYPE INTEGER.

FE A REFERENCE TO THIS ARITHMETIC STATEMENT FUNCTION HAS UNBALANCED PARENTHESIS WITHIN THE PARAMETER LIST.

FE A REFERENCE TO THIS ASF HAS A PARAMETER MISSING.

FE A VARIABLE DIMENSION OR THE ARRAY NAME WITH A VARIABLE DIMENSION IS NOT A FORMAL PARAMETER.

FE ALL ECS ITEMS MUST APPEAR IN A COMMON BLOCK.

FE AN ARRAY REFERENCE HAS TOO MANY SUBSCRIPTS.

FE APPEARED WHERE A VARIABLE WAS EXPECTED.

FE ARG TO LOCF MAY NOT BE AN EXPRESSION.

FE ARGUMENT NOT FOLLOWED BY COMMA OR RIGHT PARENTHESIS.

FE ARITHMETIC STATEMENT FUNCTION REDEFINED.

FE ARRAY HAS MORE THAN THREE SUBSCRIPTS.

FE ARRAY OR COMMON VARIABLE MAY NOT BE DECLARED EXTERNAL.

FE ARRAY WITH ILLEGAL SUBSCRIPTS.

FE ASF HAS MORE DUMMY PARAMETERS THAN ALLOWED.

FE BAD SUBSCRIPT IN EQUIV STMT.

FE BAD SYNTAX ENCOUNTERED.

FE BASIC EXTERNAL OR INTRINSIC FUNCTION CALLED WITH WRONG TYPE ARGUMENT.  
FE BASIC OR INTRINSIC FUNCTION WITH AN INCORRECT ARGUMENT COUNT.  
FE COMMON BLOCK LENGTH EXCEEDS 131071 WORDS.  
FE COMMON VARIABLE IS FORMAL PARAMETER OR PREVIOUSLY DECLARED IN COMMON OR ILLEGAL NAME.  
FE COMMON-EQUIVALENCE ERROR.  
FE CONFLICTING LEVEL DECLARATIONS EXIST IN THIS COMMON BLOCK.  
FE CONSTANT DATA ITEM MUST BE FOLLOWED BY A , / OR RIGHT PAREN.  
FE CONSTANT SUBSCRIPT VALUE EXCEEDS ARRAY DIMENSIONS.  
FE CONSTANT TABLE CONSTORS OVERFLOWED-STATEMENT TRUNCATED.ENLARGE TABLE OR SIMPLIFY STATEMENT.  
FE DATA ITEM LISTS MAY ONLY BE NESTED 1 DEEP.  
FE DATA VARIABLE LIST SYNTAX ERROR.  
FE DEBUG EXECUTION OPTION SUPPRESSED DUE TO NATURE OF ABOVE FATAL ERRORS.  
FE DECLARATIVE STATEMENT OUT OF SEQUENCE.  
FE DEFECTIVE HOLLERITH CONSTANT. CHECK FOR CHARACTER COUNT ERROR, MISSING ≠ DELIMITER OR LOST CONTIN CARD.  
FE DIVISION BY CONSTANT ZERO.  
FE DO LIMIT OR REP FACTOR MUST BE AN INTEGER OR OCTAL CONSTANT BETWEEN 1 AND 131K.  
FE DO LOOPS TERMINATING ON THIS LABEL ARE IMPROPERLY NESTED.  
FE DOUBLY DEFINED FORMAL PARAMETER.  
FE DUMMY PARAMETER IN ASF DEFINITION OCCURED TWICE.  
FE DUPLICATE LOOP INDEX OR DOESNT MATCH ANY SUBSCRIPT VARIABLE.  
FE DUPLICATE STATEMENT LABEL.  
FE ECS/LCM REFERENCE MUST BE A STAND-ALONE ARGUMENT TO AN EXTERNAL ROUTINE.  
FE EITHER OR BOTH OF FWA AND LWA ARE FORMAL PARAMETERS.  
FE ENTRY POINT NAMES MUST BE UNIQUE - THIS ONE HAS BEEN PREVIOUSLY USED IN THIS SUBPROGRAM.  
FE ENTRY STATEMENT MAY NOT APPEAR IN A PROGRAM.  
FE ENTRY STATEMENT MAY NOT BE LABELED.  
FE ENTRY STATEMENTS MAY NOT OCCUR WITHIN THE RANGE OF A DO STATEMENT.

FE EQUATED FILENAME NOT PREVIOUSLY DEFINED.  
FE EQUIVALENCED COMMON BLOCK EXCEEDS 131071 WORDS.  
FE EXPRESSION TRANSLATOR TABLE (ARLIST) OVERFLOWED. SIMPLIFY THE EXPRESSION.  
FE EXPRESSION TRANSLATOR TABLE (FRSTB ) OVERFLOWED. SIMPLIFY THE EXPRESSION.  
FE EXPRESSION TRANSLATOR TABLE (OPSTAK) OVERFLOWED. SIMPLIFY THE EXPRESSION.  
FE F.P. WITH VARIABLE DIMENSIONS NOT ALLOWED IN A NAMELIST STATEMENT.  
FE FIELD WIDTH IS GREATER THAN 131,071. SCANNING STOPS.  
FE FILENAME IS GREATER THAN 6 CHARACTERS.  
FE FILENAME PREVIOUSLY DEFINED.  
FE FIRST WORD AND LAST WORD ADDRESSES OF DATA TRANSMISSION BLOCK MUST BE IN THE SAME LEVEL.  
FE FOLLOWED BY AN ILLEGAL ITEM.  
FE FORMAL PARAMETERS MAY NOT APPEAR IN COMMON OR EQUIV STMTS.  
FE FORMAT REFERENCE ILLEGAL.  
FE FORMAT STATEMENT ENDS BEFORE END OF HOLLERITH STRING. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE FORMAT STATEMENT ENDS BEFORE LAST HOLLERITH COUNT IS COMPLETE. ERROR SCAN FOR THIS FORMAT STOPS AT H .  
FE FUNCTION NAME DOES NOT APPEAR AS A VARIABLE IN THIS SUBPROGRAM.  
FE FWA AND LWA NOT IN SAME ARRAY, EQUIVALENCE CLASS, OR COMMON BLOCK.  
FE GO TO STATEMENT - SYNTAX ERROR.  
FE GROUP NAME NOT SURROUNDED BY SLASHS.  
FE GROUP NAME PREVIOUSLY REFERENCED IN ANOTHER CONTEXT.  
FE HEADER CARD NOT FIRST STATEMENT.  
FE HEADER CARD SYNTAX ERROR.  
FE ILLEGAL BLOCK NAME.  
FE ILLEGAL CHARACTER BOUND IN IMPLICIT STATEMENT.  
FE ILLEGAL CHARACTER FOLLOWS PRECEDING A,I,L,O,R OR Z DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE ILLEGAL CHARACTER FOLLOWS PRECEDING FLOATING POINT DESCRIPTOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE ILLEGAL CHARACTER FOLLOWS PRECEDING SIGN CHARACTER. ERROR SCAN FOR THIS FORMAT STOPS HERE.

FE ILLEGAL CHARACTER FOLLOWS TAB SETTING DESIGNATOR. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE ILLEGAL CHARACTER. THE REMAINDER OF THIS STATEMENT WILL NOT BE COMPILED.  
FE ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN.  
FE ILLEGAL FORM INVOLVING THE USE OF A COMMA.  
FE ILLEGAL LABELS IN IF STATEMENT.  
FE ILLEGAL LIST ITEM ENCOUNTERED IN AN I/O LIST SEQUENCE.  
FE ILLEGAL NAMELIST VARIABLE.  
FE ILLEGAL RETURNS PARAMETER.  
FE ILLEGAL SEPARATOR ENCOUNTERED.  
FE ILLEGAL SEPARATOR IN EXTERNAL STATEMENT.  
FE ILLEGAL SYNTAX AFTER INITIAL KEYWORD OR NAME.  
FE ILLEGAL SYNTAX IN CALL STATEMENT.  
FE ILLEGAL SYNTAX IN COMMON DECLARATION.  
FE ILLEGAL SYNTAX IN IF STATEMENT.  
FE ILLEGAL SYNTAX IN IMPLICIT STATEMENT.  
FE ILLEGAL TYPE SPECIFIED IN IMPLICIT STATEMENT.  
FE ILLEGAL USE OF A FUNCTION NAME.  
FE ILLEGAL USE OF THE EQUAL SIGN.  
FE ILLEGAL VARIABLE NAME FIELD IN ASSIGN OR ASSIGNED GOTO.  
FE IMPROPER FORM OF ENTRY STATEMENT. ONLY ALLOWABLE FORM IS [ ENTRY NAME ].  
FE INTRINSIC FUNCTION REFERENCE MAY NOT USE A FUNCTION NAME AS AN ARGUMENT.  
FE INVALID LEVEL NUMBER SPECIFIED.  
FE INVALID USE OF A CHARACTER STRING.  
FE INVOLVED IN CONTRADICTIONARY EQUIVALENCING.  
FE ITEMS IN DIFFERENT LEVELS OF STORAGE MAY NOT BE EQUIVALENCED.  
FE LEFT SIDE OF REPLACEMENT STATEMENT IS ILLEGAL.  
FE LEVEL 3 VARIABLE MAY NOT APPEAR IN AN EQUIVALENCE STATEMENT.

FE LOADER DIRECTIVE OUT OF SEQUENCE. MUST PRECEDE PROGRAM UNIT HEADER LINE.  
FE LOGICAL AND NON-LOGICAL OPERANDS MAY NOT BE MIXED.  
FE LOGICAL EXPRESSION IN 3-BRANCH IF STATEMENT.  
FE LOGICAL OPERAND USED WITH NON-LOGICAL OPERATORS.  
FE LOOP BEGINNING AT THIS CARD NO IS ENTERED FROM OUTSIDE ITS RANGE AND HAS NO EXITS.  
FE LOOPS ARE NESTED MORE THAN 50 DEEP.  
FE MAXIMUM PARENTHESIS NESTING LEVEL EXCEEDED. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE MAY NOT BE FUNCTION, EXTERNAL, F.P. OR IN BLANK COMMON.  
FE MISSING OR SYNTAX ERROR IN LIST OF TRANSFER LABELS.  
FE MISSING, BAD, OR OUT OF RANGE LABEL ON DO STATEMENT.  
FE MORE THAN ONE RELATIONAL OPERATOR IN A RELATIONAL EXPRESSION.  
FE MORE THAN 50 FILES ON PROGRAM CARD OR 63 PARAMETERS ON A SUBROUTINE OR FUNCTION CARD.  
FE MORE THAN 63 ARGUMENTS IN ARGUMENT LIST.  
FE NAMELIST STATEMENT SYNTAX ERROR.  
FE NO MATCHING LEFT PARENTHESIS.  
FE NO MATCHING RIGHT PARENTHESIS IN ARGUMENT LIST.  
FE NO MATCHING RIGHT PARENTHESIS IN SUBSCRIPT.  
FE NO MATCHING RIGHT PARENTHESIS.  
FE NON DIMENSIONED NAME APPEARS FOLLOWED BY LEFT PAREN.  
FE NON-INNER LOOP BEGINNING AT THIS CARD IS ENTERED FROM OUTSIDE ITS RANGE.  
FE NON-STANDARD RETURN STATEMENT MAY NOT APPEAR IN A FUNCTION SUBPROGRAM.  
FE NUMBER OF ACTUAL PARAMETERS PLUS RETURNS EXCEED 63.  
FE NUMBER OF CHARACTERS IN AN ENCODE/DECODE STATEMENT MUST BE AN INTEGER CONSTANT OR VARIABLE.  
FE NUMBER OF SUBSCRIPTS IS INCOMPATIBLE WITH THE NUMBER OF DIMENSIONS DURING EQUIVALENCING.  
FE ONLY ONE SYMBOLIC NAME IN EQUIVALENCE GROUP.  
FE PARAMETER ON NON-STANDARD RETURN STATEMENT IS NOT A RETURNS FORMAL PARAMETER.  
FE PRECEDING CHARACTER ILLEGAL AT THIS POINT IN STRING. ERROR SCAN FOR THIS FORMAT STOPS HERE.

FE PRECEDING CHARACTER ILLEGAL. SCALE FACTOR EXPECTED. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE PRECEDING HOLLERITH COUNT IS EQUAL TO ZERO. ERROR SCAN FOR THIS FORMAT STOPS HERE.  
FE PRECEDING HOLLERITH INDICATOR IS NOT PRECEDED BY A COUNT. SCANNING STOPS HERE.  
FE PRESENT USE OF THIS LABEL CONFLICTS WITH PREVIOUS USES.  
FE PROGRAM OR SUBROUTINE NAME MAY NOT BE REFERENCED IN A DECLARATIVE STATEMENT.  
FE RECORD LENGTH IS GREATER THAN 131,071.  
FE REFERENCED LABEL IS MORE THAN FIVE CHARACTERS.  
FE RETURN STATEMENT APPEARS IN MAIN PROGRAM.  
FE RETURNS LIST ERROR.  
FE RETURNS OR EXTERNAL NAMES MAY NOT APPEAR IN DECLARATIVE STATEMENTS.  
FE RIGHT PARENTHESIS FOLLOWED BY A NAME, CONSTANT, OR LEFT PARENTHESIS.  
FE SIMPLE VARIABLE OR CONSTANT FOLLOWED BY LEFT PARENTHESIS.  
FE STATEMENT TOO LONG.  
FE SUBROUTINE NAME REFERRED TO BY CALL IS USED ELSEWHERE AS A NON-SUBROUTINE NAME.  
FE SYMBOLIC NAME HAS TOO MANY CHARACTERS.  
FE SYNTAX ERROR IN ASF DEFINITION.  
FE SYNTAX ERROR IN DATA ITEM LIST.  
FE SYNTAX ERROR IN DATA STATEMENT.  
FE SYNTAX ERROR IN DUMMY ARGUMENT LIST OF STATEMENT FUNCTION.  
FE SYNTAX ERROR IN EQUIVALENCE STATEMENT.  
FE SYNTAX ERROR IN IMPLIED DO NEST.  
FE SYNTAX ERROR IN INPUT/OUTPUT STATEMENT.  
FE SYNTAX ERROR IN LOADER DIRECTIVE.  
FE SYNTAX ERROR IN SUBSCRIPT LIST, MUST BE OF FORM CON1\*IVAR+CON2.  
FE TAB SETTING IS GREATER THAN 131,071. SCANNING STOPS.  
FE THE CONTROL VARIABLE OF A DO OR DO IMPLIED LOOP MUST BE A SIMPLE INTEGER VARIABLE.  
FE THE EXPRESSION IN A LOGICAL IF IS NOT TYPE LOGICAL.  
FE THE FIELD FOLLOWING STOP OR PAUSE MUST BE 5 OR LESS OCTAL DIGITS OR A QUOTE-DELIMITED STRING.

FE THE OPERATOR INDICATED (-,+,\*,/, OR \*\*) MUST BE FOLLOWED BY A CONSTANT, NAME, OR LEFT PARENTHESIS.

FE THE STATEMENT IN A LOGICAL IF MAY BE ANY EXECUTABLE STATEMENT OTHER THAN A DO OR ANOTHER LOGICAL IF.

FE THE SYNTAX OF DO PARAMETERS MUST BE I=M1,M2,M3 OR I=M1,M2.

FE THE TERMINAL STATEMENT OF THIS DO PRECEDES IT.

FE THE TYPE OF THIS IDENTIFIER IS NOT LEGAL FOR ANY EXPRESSION.

FE THE VALUE OF THE PARITY INDICATOR IN A BUFFER I/O STATEMENT MUST BE 0 OR 1.

FE THIS ASSIGN STATEMENT HAS IMPROPER FORMAT, ONLY ALLOWABLE IS (ASSIGN LABEL TO VARIABLE ).

FE THIS NAME MAY NOT BE USED IN A DATA STMT.

FE THIS OPERATOR (.NOT. OR A RELATIONAL) MUST BE FOLLOWED BY A CONSTANT, NAME, LEFT PAREN, + OR -.

FE THIS PROGRAM UNIT CALLS ITSELF.

FE THIS STATEMENT MAKES AN ILLEGAL TRANSFER INTO A PREVIOUS DO LOOP.

FE THIS STATEMENT TYPE IS ILLEGAL IN BLOCK DATA SUBPROGRAM.

FE TOO MANY LABELED COMMON BLOCKS, ONLY 125 BLOCKS ARE ALLOWED.

FE TOO MANY SUBSCRIPTS IN ARRAY REFERENCE.

FE TOTAL RECORD LENGTH IS GREATER THAN 131,071. SCANNING STOPS.

FE UNDEFINED STATEMENT NUMBERS, SEE BELOW.

FE UNIT NUMBER MUST BE BETWEEN 1 AND 99 INCLUSIVE.

FE UNIT NUMBER OR PARITY INDICATOR MUST BE AN INTEGER CONSTANT OR VARIABLE.

FE UNMATCHED PARAMETER COUNT IN A REFERENCE TO THIS STATEMENT FUNCTION.

FE UNMATCHED PARENTHESIS.

FE UNRECOGNIZED STATEMENT.

FE USE OF THIS PROGRAM OR SUBROUTINE NAME IN AN EXPRESSION.

FE VALUE OF ARRAY SUBSCRIPT IS .LT. 1 OR .GT. DIMENSIONALITY IN IMPLIED DO NEST.

FE VARIABLE IN ASSIGN OR ASSIGNED GO TO IS ILLEGAL.

FE VARIABLE SUBSCRIPTS MAY NOT APPEAR WITHOUT DO LOOPS.

FE WAS LAST CHARACTER SEEN AFTER TROUBLE. REMAINDER OF STATEMENT IGNORED.

FE WORKING STORAGE EXCEEDED. FORMAT SCAN HALTED.

FE ZERO IS SPECIFIED AS REPEAT COUNT. SCANNING STOPS.

FE ZERO LEVEL RIGHT PARENTHESIS MISSING. SCANNING STOPS.

FE ZERO STATEMENT LABELS ARE ILLEGAL.

I \*\*\* DUE TO THE MANY ERRORS NOTED, ONLY THOSE WHICH ARE FATAL WILL BE LISTED HEREAFTER.

I \*PROGRAM START. (INPUT,OUTPUT)\* ASSUMED WHEN HEADER STATEMENT IS DEFECTIVE OR OMITTED.

I A HOLLERITH CONSTANT IS AN OPERAND OF AN ARITHMETIC OPERATOR.

I A TYPE WAS DECLARED PREVIOUSLY FOR THIS VARIABLE OR FUNCTION. THIS DECLARATION IGNORED.

I AN IF STATEMENT MAY BE MORE EFFICIENT THAN A 2 OR 3 BRANCH COMPUTED GO TO STATEMENT.

I ARGUMENT COUNT INCONSISTENT WITH PRIOR USAGE.

I ARRAY NAME OPERAND NOT SUBSCRIPTED, FIRST ELEMENT WILL BE USED.

I ARRAY REFERENCE OUTSIDE DIMENSION BOUNDS.

I CHARACTER BOUNDS REVERSED IN IMPLICIT STATEMENT.

I COMMA MISSING BEFORE VARIABLE INDICATED.

I CONSTANT LENGTH .GT. VARIABLE LENGTH, CONSTANT TRUNCATED.

I CONSTANT TOO LONG. HIGH ORDER DIGITS RETAINED, BUT SOME PRECISION LOST.

I CONTROL VARIABLE IN COMMON OR EQUIVALENCED, OPTIMIZATION MAY BE INHIBITED.

I DATA ITEM LIST EXCEEDS VARIABLE LIST, EXCESS CONSTANTS IGNORED.

I DATA VARIABLE LIST EXCEEDS ITEM LIST, EXCESS VARIABLES NOT INITIALIZED.

I DECIMAL DIGITS EXPECTED AFTER DECIMAL POINT. DEPENDING ON THE DESCRIPTOR, A ONE OR A ZERO IS ASSUMED.

I DECIMAL POINT SPECIFICATION MISSING FROM FLOATING POINT DESCRIPTOR.

I DIMENSIONAL RANGE IS EXTENDED FOR EQUIVALENCING PURPOSES.

I FIELD WIDTH IS GREATER THAN 137 CHARACTERS. IT MAY EXCEED THE I/O DEVICE CAPACITY.

I FIELD WIDTH OF A CONVERSION DESCRIPTOR SHOULD BE AS LARGE AS THE MINIMUM SPECIFIED FOR THAT DESCRIPTOR.

I FILE LENGTH REQUESTED IS TOO LARGE. STANDARD LENGTH OF 2000B SUBSTITUTED.

I FWA AND LWA NOT IN SAME ARRAY OR EQUIVALENCE CLASS.

I I/O BUFFER LENGTH SPECIFICATION IS NOT MEANINGFUL--VALUE IGNORED.

I ILLEGAL CHARACTERS AFTER TERMINATING RIGHT PARENTHESIS IGNORED.

I ILLEGAL LABEL FIELD.

I LEVEL CONFLICTS WITH PREVIOUS DECLARATION. ORIGINAL LEVEL RETAINED.

I LOWER LIMIT .GE. UPPER LIMIT, ONE TRIP LOOP.

I MASK ARGUMENT MUST BE NONNEGATIVE AND LESS THAN 61.



I MAY NOT USED IN A DEBUG STATEMENT.  
I MISSING I/O LIST OR SPURIOUS COMMA.  
I MORE STORAGE REQUIRED BY DO STATEMENT PROCESSOR FOR OPTIMIZATION.  
I NO DIGIT PRECEDED X-FIELD. 1X ASSUMED.  
I NO END CARD, END LINE ASSUMED.  
I NOT ALL ITEMS IN THIS COMMON BLOCK OCCUR IN LEVEL STATEMENTS.  
I NUMERIC FIELD FOLLOWING TAB SETTING DESIGNATOR IS EQUAL TO ZERO. COLUMN ONE WILL BE ASSUMED.  
I NUMERIC FIELD OMITTED FROM PRECEDING SCALE FACTOR. ZERO SCALE FACTOR ASSUMED.  
I PRECEDING FIELD WIDTH IS ZERO.  
I PRECEDING FIELD WIDTH SHOULD BE 7 OR MORE.  
I PRECEDING SCALE FACTOR EXCEEDS THE LIMIT OF REPRESENTATION WITHIN THE MACHINE.  
I PRESENT USE IN CONTEXT OF THIS NAME DOES NOT MATCH PREVIOUS OCCURRENCES IN DEBUG STMTS.  
I PREVIOUSLY DIMENSIONED ARRAY. FIRST DIMENSIONS WILL BE RETAINED.  
I SEPARATOR MISSING. SEPARATOR ASSUMED HERE.  
I SHIFT ARGUMENT MUST BE GREATER THAN -61 AND LESS THAN 61.  
I SINGLE WORD CONSTANT MATCHED WITH DOUBLE OR COMPLEX VARIABLE. PRECISION LOST.  
I SPURIOUS CHARACTERS AFTER CONTINUE IGNORED.  
I SUPERFLUOUS SCALE FACTOR ENCOUNTERED BEFORE THE CURRENT SCALE FACTOR.  
I TAB SETTING MAY EXCEED RECORD SIZE, DEPENDING ON USE.  
I THE UPPER LIMIT AND CONTROL VARIABLES OF THIS DO ARE THE SAME, PRODUCING A NON-TERMINATING LOOP.  
I THERE IS NO PATH TO THIS STATEMENT.  
I THIS IF DEGENERATES INTO A SIMPLE TRANSFER TO THE LABEL INDICATED.  
I THIS STATEMENT BRANCHES TO ITSELF.  
I THIS STATEMENT FORM IS OBSOLETE. USE A LEVEL 3 STATEMENT.  
I THIS STATEMENT MAY REDEFINE A CURRENT LOOP CONTROL VARIABLE OR PARAMETER, OPTIMIZATION INHIBITED.  
I THIS STATEMENT REDEFINES A CURRENT LOOP CONTROL VARIABLE OR PARAMETER.  
I TOTAL RECORD LENGTH IS GREATER THAN 137 CHARACTERS. IT MAY EXCEED THE I/O DEVICE CAPACITY.  
I TRIVIAL EQUIVALENCE GROUP, IGNORED.  
I X-FIELD PRECEDED BY A ZERO. NO SPACING OCCURS.

## SPECIAL COMPILATION DIAGNOSTICS

When a compilation is aborted or prematurely terminated for internal reasons, one or more of the following messages appears if D or OPT=0, 1 or 2 has been selected. The message listed last is supplied only when TS mode is selected.

### DAYFILE MESSAGES

nnnn ASSEMBLY ERRORS IN program

A compiler, operating system or hardware error has occurred while compiling program.

COMPILING program

LAST STATEMENT BEGAN AT LINE nnnnn

ERROR AT aaaaa IN ddddddd

LAST OVERLAY LOADED - (p,s)

A compiler, operating system or hardware error has occurred while compiling program. Variable parameters in this message are as follows:

program	Name of the source program unit.
nnnnn	Approximate compiler-assigned source line number where the difficulty arose. During transitions from one phase of the compilation to another, the END line number may be displayed.
ddddddd	Name of the compiler internal deck where the abort occurred. May be RA+0 if control was accidentally transferred to the control point job communications area.
aaaaaa	Address relative to the origin of the internal deck (as above) where the abort occurred.
p,s	Primary and secondary level numbers of the overlay last loaded before the abort occurred. The overlays are:  0,0 - Control card cracker; global communication and control 1,0 - TS-mode compilation overlay 2,0 - Optimizing compilation batch controller 2,1 - Optimizing compilation normal pass 1 (lexical scan, parse, intermediate language generation) 2,2 - Optimizing compilation pass 2 (global and local optimization, object code generation) 2,3 - Optimizing compilation diagnostic phase (occurs between pass 1 and pass 2) 2,4 - Optimizing compilation C\$ DEBUG pass 1 2,5 - Optimizing compilation reference map generation and object code assembly phase)

#### DEAD CODE IN program

Please refer to the message STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED in the following section entitled Compiler Output Listing Message for a full description of this error condition.

#### ECS READ ERROR ECS WRITE ERROR

An ECS/LCM read or write parity error has been detected. During compilation, this difficulty can occur only when OPT=2 has been selected.

#### FTN/FBV - BLOCK READ ERROR

A compiler, operating system or hardware error has occurred. The most probable cause is a disk or READNS (read non-stop) I/O command error. During compilation, this difficulty can occur only when OPT=2 has been selected.

#### NULL PROGRAM IGNORED AFTER program

A program unit has been detected that does not contain a single executable statement. It is ignored.

#### OBJECT CODE END LINE MISSING

A compiler, operating system or hardware error has occurred.

#### \*\* PASS 2 MEMORY OVERFLOW \*\*

Please refer to the similar message entry in the Compiler Output Listing Messages description that follows.

#### \*\* PREMATURE EOF ON -REFMAP- FILE.

A compiler, operating system or hardware error has occurred. This error can occur only when a long reference map has been selected by R=2 or 3 on the control card.

(TS MODE ONLY)

#### EMPTY INPUT FILE. NO COMPILATION.

An end-of-record, end-of-partition, or end-of-section was encountered on the first read of the input file.

## COMPILER OUTPUT LISTING MESSAGES

The following error messages can appear in the body of the compilation listing. If present, they will be located after the source program and standard error summary listings. They may appear before, during or after the reference map and object code listings, depending on the exact error condition. The message format is quite different from that of the standard error summary; each message is usually left-justified on the output listing page, and may be preceded by several blank lines or a page eject.

### CANT SORT SYMBOL TABLE INCREASE FL BY fffb

Not enough CM/SCM field length was available to generate a reference map. fffb is a rough estimate of the additional field length that is necessary for generating the map. This error cannot occur if the reference map has been suppressed by selecting R=0 on the control card.

### \*\*\* MEMORY OVERFLOW IN -FAX-

Not enough CM/SCM field length was available for final assembly of the binary object code. No estimate can be provided for the additional memory required; 10K to 20K increments are suggested.

### PASS 2 MEMORY OVERFLOW AT SOURCE LINE nnnn IN compnam

This message may indicate a genuine memory overflow, or may result from a compiler error. If compnam is JAM-ERR, a compiler error has occurred. Otherwise, not enough CM/SCM field length was available for completing pass 2 of an optimizing compilation. Since field length requirements increase with higher degrees of optimization, the problem can be eliminated either by increasing the field length or by decreasing the optimization level selection.

### REFERENCES AFTER LINE nnnn LOST INCREASE FL BY fffb

Not enough CM/SCM field length was available to generate a complete long reference map. nnnnn is the approximate compiler-assigned source line number where the difficulty arose. fffb is a rough estimate of the additional field length needed for generating the complete map. This error can occur only when a long reference map has been selected by R=2 or 3 on the control card.

STATEMENTS BEGINNING AT THE BELOW LINE  
NUMBERS ARE UNREACHABLE (DEAD CODE),  
AND WILL NOT BE PROCESSED.

One or more executable statements in the source program can never be executed, due to the program flow of control. No object code has been compiled for any dead statements. Due to the possible severity of the error, it is accompanied by the dayfile message, DEAD CODE IN program. The error condition is detected only when OPT=2 has been selected.

A simple example of the error follows:

```
          A=2.  
          GO TO 30  
C          THE NEXT STATEMENT CANNOT BE EXECUTED.  
          A=A+1.  
30        STOP  
          END
```

A more subtle example is:

```
          A=2.  
          ASSIGN 40 TO J  
          ASSIGN 50 TO J  
          ASSIGN 60 TO J  
          GO TO J, (40,50)  
C          THE NEXT STATEMENT CANNOT BE EXECUTED, BECAUSE  
C          ITS LABEL DOES NOT APPEAR IN THE GO-TO TRANSFER  
C          LIST.  
60        A=A+1.  
40        STOP  
50        STOP  
          END
```

## COMPILATION DIAGNOSTICS, TS MODE

When TS mode is selected, error messages are intermixed with the source listing as they are detected. The format of the error message is:

severity \* text

The severity can be:

- FATAL            Error is fatal to execution.
- WARNING        Error is severe, but not fatal. Syntax is incorrect, but probable meaning is presumed.
- NOTE            Minor syntax error or omission.
- ANSI            Usage does not conform to ANSI X3.9 - 1966 FORTRAN specification. Listed only if EL=A list option is specified.

In addition to the above, certain unsuppressible non-fatal diagnostics may be listed (regardless of the EL specification on the FTN control card).

On the following pages appear the compilation diagnostics produced in TS mode, grouped according to the processor that detects the error. Ellipses denoted by ..... are replaced in an actual message by items from the relevant source statement, distinguished by a preceding ↗ (or \_). Micro names delimited by ≠ pairs (such as ≠MAX.SARG≠) are replaced by numerical values supplied by the system.

Example:

```

SUBROUTINE SUB(A,B)                               FTN 4.3*F383      08/01/74 15.00.16.  PAGE 1
                                                    TS                /73
3          SUBROUTINE SUB(A,B)
3          DIMENSION A(2)
3          COMMEN B(4)
----- *
*FATAL *      MISPELLED KEYWORD -- COMMON ASSUMED
              USAGE CONFLICT -- B IS DUMMY-ARG AND CANNOT BE COMMON
3          DO 10, I=1,4
WARNING *      COMMA AFTER DO LABEL IGNORED
A N S I *      COMMA NOT PERMITTED AFTER DO LABEL
3          WRITE(4,1) A,B
3          11  FORMAT(2A10)
3          VARIABL=A+B
A N S I *      ARRAY A MISSING SUBSCRIPT -- FIRST ELEMENT ASSUMED
A N S I *      ARRAY B MISSING SUBSCRIPT -- FIRST ELEMENT ASSUMED
3          CONTINUE
NOTE *        CONTINUE WITH NO STATEMENT LABEL -- IGNORED
----- *
A N S I *      END LINE ABSENT
*FATAL *      CONTROL FLOW INTO END LINE NOT PERMITTED
*FATAL *      STATEMENT LABEL .10 REFERENCED BUT NOT DEFINED
*FATAL *      DO LOOP .10 NOT TERMINATED BEFORE END OF PROGRAM

BLOCK          CODE  LITERALS  FORMATS  TEMPS  ARGS  NAMELIST  VARIABLES  BUFFERS
ADDRESS        3        3        3        5     5      12        12        14
LENGTH         0        0        2        0     5       0         2         0

37100 STORAGE USED          MODEL 74          8 SOURCE STATEMENTS
14     PROGRAM-UNIT LENGTH  COMPILATION      .021 SECONDS

3 FORTRAN ERRORS IN SUB

```

## FTN(TS) COMPILE-TIME ERROR MESSAGES

## ARGUMENT PROCESSOR.

FATAL ONLY #MAX.SARG# DUMMY ARGUMENTS ARE PERMITTED -- EXCESS IGNORED  
 FATAL DUMMY ARGUMENT ..... PREVIOUSLY DEFINED  
 FATAL DUMMY ARGUMENT ..... MUST BEGIN WITH LETTER  
 FATAL SYNTAX -- EXPECTED RETURNS FOUND .....  
 FATAL SYNTAX -- EXPECTED LEFT PAREN OR COMMA FOUND .....  
 WARNING ILLEGAL CHARACTER AFTER RIGHT PAREN  
 WARNING MISSPELLED KEYWORD -- ..... RETURNS ASSUMED  
 FATAL SYNTAX IN ARGUMENT LIST -- EXPECTED LEFT PAREN FOUND .....  
 FATAL SYNTAX ERROR IN ARGUMENT LIST  
 FATAL FUNCTION MUST HAVE AT LEAST ONE DUMMY ARGUMENT  
 FATAL RETURNS LIST NOT PERMITTED IN FUNCTION STATEMENT  
 FATAL SYNTAX -- EXPECTED E.O.S. OR RETURNS PARAMETER FOUND .....  
 FATAL NAME ..... DID NOT APPEAR IN RETURNS LIST

## ASF PROCESSOR

FATAL ILLEGAL STATEMENT FUNCTION SYNTAX -- TROUBLE STARTED AT .....  
 FATAL STATEMENT FUNCTION ..... -- ARGUMENT SYNTAX  
 FATAL STATEMENT FUNCTION ..... -- MISPLACED EQUAL SIGN  
 NOTE STATEMENT FUNCTION ..... HAS NULL DEFINITION -- IGNORED  
 FATAL STATEMENT FUNCTION ..... REFERENCE -- CLOSING PAREN MISSING  
 FATAL RECURSIVE DEFINITION OF STATEMENT FUNCTION .....  
 NOTE ARGUMENT .....2 IS NEVER USED WITHIN STATEMENT FUNCTION .....  
 FATAL DUMMY ARGUMENT .....2 CAN OCCUR ONLY ONCE IN ..... DEFINITION  
 FATAL REFERENCE TO STATEMENT FUNCTION ..... HAS A NULL PARAMETER  
 FATAL UNMATCHED PARAMETER COUNT TO STATEMENT FUNCTION .....  
 FATAL STATEMENT FUNCTION SYNTAX IN DUMMY ARGUMENT LIST -- EXPECTED COMMA FOUND .....  
 FATAL STATEMENT FUNCTION SYNTAX -- EXPECTED COMMA FOUND E.O.S.  
 FATAL STATEMENT FUNCTION SYNTAX IN DUMMY PARAMETER LIST -- NULL ARGUMENT  
 FATAL DUMMY PARAMETER ..... OF STATEMENT FUNCTION NOT SIMPLE VARIABLE  
 WARNING PREVIOUS DEFINITION OF STATEMENT FUNCTION ..... IS OVERRIDDEN

## ASSORTED ANSI ERRORS

ANSI ANSI REQUIRES THE WORD PRECISION  
 ANSI CONTROL FLOW INTO END LINE NOT PERMITTED  
 ANSI STATEMENT IS NOT DEFINED IN ANSI  
 ANSI STATEMENT IS NOT DEFINED IN ANSI  
 ANSI RETURN IN MAIN PROGRAM  
 NOTE RETURN ACTS AS END

FTN(TS) COMPILE-TIME ERROR MESSAGES

ASSIGN STATEMENT.

WARNING \*TO\* ASSUMED FOR .....  
 WARNING VARIABLE ..... NOT INTEGER  
 FATAL STATEMENT LABEL ..... MUST BE NUMERIC  
 WARNING COMMA AFTER STATEMENT LABEL IGNORED

ARITH PROCESSOR.

FATAL ASSIGNMENT STATEMENT REQUIRES A VARIABLE ON LEFT OF EQUAL SIGN  
 FATAL LOGICAL AND NON- LOGICAL OPERANDS MAY NOT BE MIXED  
 FATAL LOGICAL OPERAND USED WITH NON- LOGICAL OPERATOR  
 FATAL ILLEGAL USE OF OPERATOR / OPERAND -- .....  
 NOTF HOLLERITH CONSTANT IN EXPRESSION EXCEEDS 10 CHARACTERS  
 FATAL ILLEGAL FORM INVOLVING THE USE OF A COMMA  
 FATAL ILLEGAL USE OF ASSIGNMENT OPERATOR  
 ANSI MASK EXPRESSION NON- ANSI  
 ANSI HOLLERITH CONSTANT IN EXPRESSION NON- ANSI  
 ANSI MULTIPLE ASSIGNMENT IS NON- ANSI

CALL STATEMENTS.

FATAL CALL STATEMENT MISSING ROUTINE NAME  
 FATAL SYNTAX -- EXPECTED LEFT PAREN OR COMMA AFTER ROUTINE NAME FOUND .....  
 FATAL RETURNS PARAMETER ..... MUST BE NUMERIC LABEL

COMMON PROCESSOR.

FATAL SYNTAX ERROR IN BLOCK NAME  
 ANSI NUMERIC BLOCK NAME NOT PERMITTED  
 FATAL PREMATURE E.O.S. -- EXPECTED BLOCK NAME  
 FATAL ONLY #MAX.BLK# COMMON BLOCK ARE PERMITTED -- USE // INSTEAD  
 FATAL SYNTAX -- EXPECTED COMMA OR SLASH FOUND .....  
 WARNING COMMON STATEMENT WITHOUT A LIST IS IGNORED  
 FATAL ILLEGAL BLOCK NAME IN COMMON STATEMENT  
 FATAL PREMATURE E.O.S. -- EXPECTED SYMBOL

CONSTANT REDUCTION.

NOTE CONSTANT TERM OF ZERO -- IGNORED  
 NOTE CONSTANT MULTIPLY BY ZERO -- RESULTS SET TO ZERO  
 FATAL CONSTANT DIVIDE BY ZERO -- RESULTS SET TO INFINITE  
 NOTE INTEGER DIVIDE BY ZERO -- RESULTS SET TO ZERO  
 NOTE DIVIDE INTO ZERO -- RESULTS SET TO ZERO  
 NOTE MULTIPLY BY ONE -- IGNORED  
 NOTE DIVIDE BY ONE -- IGNORED  
 FATAL RESULTS OF CONSTANT USED WITH ..... OPERATOR OUT OF RANGE



FTN(TS) COMPILE-TIME ERROR MESSAGES

CONTINUE STATEMENT PROCESSOR.

NOTE CONTINUE WITH NO STATEMENT LABEL -- IGNORED

DATA STATEMENT PROCESSOR.

ANSI ARRAY ..... MUST HAVE IMPLIED LOOP  
 FATAL ..... IS IN // COMMON -- DATA IGNORED  
 NOTE EXCESS CONSTANTS IGNORED  
 WARNING TOO FEW CONSTANTS -- VARIABLE ..... AND FOLLOWING NOT INITIALIZD  
 ANSI PAREN REPEAT LIST IS NOT PERMITTED  
 FATAL DATA INTO ..... IS ILLEGAL  
 FATAL ILLEGAL REPEAT CONSTANT  
 ANSI HOLLERITH CONSTANT LONGER THAN ONE ITEM  
 FATAL SYNTAX ERROR IN DATA CONSTANT LIST  
 FATAL ILLEGAL CONSTANT FOLLOWING + OR -  
 FATAL REPEAT FACTOR IN DATA CONSTANT LIST MUST NOT BE NESTED  
 FATAL ILLEGAL SEPARATOR FOLLOWING DATA CONSTANT  
 WARNING NULL DATA STATEMENT IS IGNORED  
 ANSI ALTERNATE FORM OF DATA STATEMENT NOT PERMITTED  
 ANSI NON- ANSI CONSTRUCTS IN THIS DATA STATEMENT  
 FATAL SYNTAX ERROR IN DATA STATEMENT  
 WARNING ..... CONSTANT TOO LONG -- TRUNCATED  
 FATAL ILLEGAL SEPARATOR AFTER .....  
 FATAL SYNTAX ERROR IN IMPLIED DO NEST  
 FATAL IMPLIED DO INDEX MUST BE FOLLOWED BY EQUAL  
 FATAL IMPLIED DO LOWER LIMIT MUST BE NUMERIC  
 FATAL NO COMMA AFTER LOWER LIMIT  
 FATAL IMPLIED DO UPPER LIMIT MUST BE NUMERIC  
 FATAL IMPLIED DO INCREMENT MUST BE NUMERIC  
 FATAL MISSING RIGHT PAREN AFTER IMPLIED DO  
 CONTIN SYNTAX IN IMPLIED DO ON ARRAY .....  
 FATAL DATA VARIABLE LIST SYNTAX ERROR  
 FATAL NO MATCH OF LOOP INDEX AND SUBSCRIPT  
 FATAL ARRAY ..... HAS A VARIABLE SUBSCRIPT WITH NO IMPLIED LOOP  
 FATAL ..... SUBSCRIPT LESS THAN ONE OR EXCEEDS DIMENSION  
 WARNING VARIABLE ..... REFERENCED AS ARRAY  
 FATAL DATA SUBSCRIPT LIST SYNTAX ERROR  
 FATAL ..... SUBSCRIPT EXCEEDS 2\*\*17-1

CONVERSION OF CONSTANT SECTION.

FATAL CONSTANT CAN NOT BE CONVERTED -- CHECK SYNTAX  
 ANSI OCTAL DATA TYPE NOT DEFINED IN ANSI  
 FATAL MAGNITUDE OF EXPONENT EXCEEDS 512  
 FATAL ILLEGAL FORM OF EXPONENT .....  
 NOTE CONSTANT MISSING EXPONENT FIELD -- ZERO ASSUMED  
 WARNING OCTAL CONSTANT EXCEEDS 20 DIGITS -- TRUNCATED  
 WARNING NON- OCTAL DIGIT IN OCTAL CONSTANT -- IGNORED  
 ANSI COMPLEX CONSTANT MUST BE ( REAL , REAL )

FTN(TS) COMPILE-TIME ERROR MESSAGES

FATAL LOGICAL CONSTANT DECLARED PART OF COMPLEX CONSTANT  
 ANSI DOUBLE CONSTANT DECLARED PART OF COMPLEX CONSTANT -- TRUNCATED

DIMENSION PROCESSOR.

FATAL SYNTAX IN DIMENSION STATEMENT  
 FATAL ARRAY ..... DIMENSION INDICATOR NOT INTEGER  
 FATAL ARRAY ..... NULL OR ZERO DIMENSION INDICATOR  
 FATAL VARIABLE DIMENSION ARRAY ..... MUST BE DUMMY ARGUMENT  
 FATAL VARIABLE DIMENSION INDICATOR ..... MUST BE DUMMY ARGUMENT  
 WARNING VARIABLE ..... HAS NO DIMENSION INDICATOR -- IGNORED  
 FATAL ARRAY ..... EXCEEDS \*MAX.DIM\* DIMENSIONS  
 FATAL ARRAY ..... DIMENSION INDICATOR .....2 EXCEEDS 2\*17-1  
 WARNING DIMENSION OF ..... IGNORED. PRIOR DIMENSION RETAINED.  
 FATAL ARRAY ..... DIMENSION INDICATOR -- TERMINAL RIGHT PAREN MISSING  
 FATAL SYNTAX ERROR ON DIMENSION INDICATOR FOR .....  
 ANSI HOLLERITH DIMENSION FOR .....  
 FATAL NEGATIVE DIMENSION FOR ..... -- SET TO 1  
 FATAL VARIABLE DIMENSION INDICATOR ..... IS NOT INTEGER

DO PROCESSOR.

FATAL SYNTAX OF DO MUST BE I=M1,M2,M3 OR M1,M2  
 FATAL ..... INDEX PARAMETER MUST BE INTEGER OR OCTAL  
 WARNING LIMIT LESS THAN INITIAL -- ONE TRIP LOOP  
 FATAL ..... INDEX PARAMETER IS TOO LARGE  
 FATAL ..... INDEX PARAMETER MUST BE SIMPLE VARIABLE  
 FATAL ..... INDEX PARAMETER MUST BE POSITIVE  
 FATAL THIS STATEMENT REDEFINES A DO CONTROL INDEX  
 FATAL DO LOOP ..... NOT TERMINATED BEFORE END OF PROGRAM  
 FATAL INDEX OF OUTER DO REDEFINED BY CURRENT DO  
 FATAL IMPROPER NESTING OF DO LOOPS  
 NOTE DO CONCLUSION NOT COMPILED -- DO DEFINITION ERROR  
 ANSI ..... INDEX PARAMETER MUST BE SIMPLE INTEGER VARIABLE OR CONSTANT  
 -OOPS- DO CONTROL INDEX MUST BE SIMPLE INTEGER VARIABLE  
 NOTE DO COLLAPSES TO NOTHING -- IGNORED  
 WARNING THIS STATEMENT REDEFINES A DO INDEX PARAMETER  
 FATAL .....2 -- ILLEGAL TRANSFER TO INSIDE A CLOSED DO LOOP  
 FATAL TRANSFER PREVIOUSLY FLAGGED IS ILLEGAL  
 NOTE POSSIBLE ILLEGAL TRANSFER FROM OUTSIDE CURRENT DO  
 FATAL DO STATEMENT SYNTAX -- EXPECTED CONTROL INDEX -- FOUND E.O.S.  
 ANSI COMMA NOT PERMITTED AFTER DO LABEL  
 WARNING COMMA AFTER DO LABEL IGNORED

EQUIVALENCE PROCESSOR.

FATAL SYNTAX IN EQUIV. STATEMENT  
 FATAL MISSING BEGINNING LEFT PAREN AT .....

## FTN(ITS) COMPILE-TIME ERROR MESSAGES

FATAL SUBSCRIPT ..... EXCEEDS 2\*\*17-1  
 FATAL MISSING COMMA AT .....  
 FATAL MORE THAN #MAX.DIM# SUBSCRIPT  
 FATAL SUBSCRIPT ..... MUST BE NON- ZERO NUMERIC INTEGED CONSTANT  
 WARNING TRIVIAL EQUIV. GROUP WITH ONLY ONE MEMBER IS IGNORED

## CLOSE OF DECLARATIVES PROCESSING.

WARNING REDUNDANT EQUIV. SPEC.  
 FATAL CONFLICT IN EQUIV. SPEC.  
 FATAL NO DIMENSION FOUND FOR EQUIV. VARIABLE .....  
 FATAL EXCESS SUBSCRIPTS ON EQUIV. VARIABLE .....  
 WARNING MISSING SUBSCRIPTS SET TO ONE FOR EQUIV. VARIABLE .....

## EXTERNAL PROCESSOR.

NOTE ..... ALREADY EXTERNAL  
 FATAL SYNTAX -- EXPECTED COMMA FOUND .....  
 WARNING PREMATURE E.O.S. -- EXPECTED VARIABLE AT .....  
 WARNING MUST NOT DECLARE ENTRY ..... AS EXTERNAL -- IGNORED

## FORMAT PROCESSOR.

WARNING FORMAT MUST HAVE STATEMENT LABEL  
 FATAL TERMINAL RIGHT PAREN MISSING  
 FATAL ONLY 9 PAREN LEVELS ALLOWED  
 FATAL REPEAT COUNT IS NOT ALLOWED BEFORE THE FIELD DESCRIPTOR .....  
 WARNING T CODE RESETS COLUMN POINTER, OVERLAYING PREVIOUS LINE IMAGE  
 ANSI T EDIT IS NULL OR ZERO, COLUMN POINTER RESET AT ONE  
 FATAL SIGNED COUNT ALLOWED ONLY BEFORE P OR NX CODE  
 ANSI S CODE IS SPECIFIED  
 ANSI SKIP COUNT FOR X CODE IS PRECEDED BY .....  
 WARNING BACKSPACE ATTEMPTED BEYOND FIRST COLUMN -- COLUMN POINTER RESET TO FIRST COLUMN  
 ANSI X CODE PRECEDED BY NON- DIGIT -- 1X ASSUMED  
 ANSI X CODE PRECEDED BY ZERO -- X CODE IGNORED  
 ANSI ..... IS SPECIFIED AS CONVERSION CODE  
 FATAL ZERO IS SPECIFIED AS REPEAT COUNT  
 FATAL FIELD WIDTH OF THE CONVERSION CODE ..... IS ZERO OR NOT SPECIFIED  
 ANSI MINIMUM DIGITS IS SPECIFIED FOR THE CONVERSION CODE .....  
 ANSI DECIMAL POINT IS NOT SPECIFIED FOR THE CONVERSION CODE .....  
 ANSI EXPONENT LENGTH IS SPECIFIED FOR THE CONVERSION CODE .....  
 WARNING FIELD WIDTH OF CONVERSION CODE ..... IS LESS THAN THE MINIMUM REQUIRED  
 ANSI EQUAL SIGN = IS SPECIFIED FOR A DIGIT  
 FATAL COUNT FOR H CODE IS ZERO OR MISSING -- SCAN STOPS  
 FATAL REOCRD LENGTH EXCEEDS 131,071 COLUMNS  
 WARNING RECORD LENGTH EXCEEDS 137 COLUMNS -- MAY EXCEED I/O DEVICE  
 FATAL UNKNOWN FORMAT CODE ..... -- SCAN RESUMES AT NEXT SEPARATOR  
 FATAL CHARACTER ..... FOUND BEYOND TERMINAL RIGHT PAREN!

FTN(TS) COMPILE-TIME ERROR MESSAGES

FATAL TERMINAL RIGHT PAREN MISSING  
 FATAL FORMAT LABEL PREVIOUSLY REFERENCED AS DO STATEMENT LABEL  
 FATAL FORMAT LABEL PREVIOUSLY REFERENCED AS CONTROL STATEMENT LABEL  
 FATAL DUPLICATED DEFINITION OF CURRENT FORMAT NUMBER

GO TO STATEMENT.

FATAL SYNTAX IN GO TO STATEMENT  
 FATAL OBJECT OF GO TO MISSING  
 WARNING OBJECT OF GO TO NOT INTEGER VARIABLE  
 NOTE NULL TRANSFER STATEMENT -- TRANSFER IGNORED  
 NOTE IF RESULTS IN A NULL TRANSFER -- TEST IGNORED  
 FATAL EXPECTED LEFT PAREN -- FOUND .....

ANSI REQUIRES COMMA BEFORE VARIABLE NAME IN COMPUTED GO TO  
 ANSI REQUIRES COMMA AFTER VARIABLE NAME IN ASSIGNED GO TO  
 WARNING STATEMENT TRANSFERS TO ITSELF  
 NOTE STATEMENT CAN TRANSFER TO ITSELF  
 ANSI COMPUTED GO TO INDEX MUST BE SIMPLE VARIABLE  
 FATAL COMPUTED GO TO INDEX MUST NOT BE LOGICAL  
 ANSI COMPUTED GO TO INDEX MUST BE INTEGER

CONVERSION OF HOLLERITH CONSTANTS.

FATAL ZERO LENGTH SPECIFIED ON HOLLERITH CONSTANT  
 FATAL E.O.S. BEFORE HOLLERITH COUNT EXHAUSTED  
 FATAL TERMINAL DELIMITER ..... MISSING  
 FATAL NO CHARACTERS FOUND IN ..... DELIMITED HOLLERITH STRING  
 ANSI NON- ANSI HOLLERITH FORM

IF PROCESSOR.

FATAL ILLEGAL IF STATEMENT -- OBJECT MISSING  
 ANSI 2 BRANCH IF IS NON- ANSI  
 ANSI OBJECT OF IF IS ILLEGAL DO TERMINATOR  
 FATAL ILLEGAL OBJECT OF IF -- TROUBLE STARTED AT .....

WARNING THIS IF RESULTS IN A SIMPLE TRANSFER TO STATEMENT INDICATED  
 WARNING LAST IF RESULTS IN A NULL TRANSFER TO THIS STATEMENT  
 FATAL INVALID OBJECT OF LOGICAL IF  
 FATAL 3 BRANCH IF NOT DEFINED FOR LOGICAL RESULTS  
 ANSI COMPLEX EXPRESSION IN AN IF STATEMENT  
 FATAL ARITHMETIC IF HAS STATEMENT AS OBJECT  
 FATAL ONLY ONE LABEL IN IF STATEMENT  
 FATAL LOGICAL IF MUST NOT BE OBJECT OF LOGICAL IF  
 FATAL TOO MANY LABELS IN LOGICAL IF

## FTN(TS) COMPILE-TIME ERROR MESSAGES

## I/O PROCESSOR.

```

FATAL      UNIT DESIGNATOR ..... MUST BE SIMPLE INTEGER VARIABLE OR CONSTANT
FATAL      MISSING UNIT DESIGNATOR IN I/O STATEMENT
WARNING    EXTRANEIOUS COMMA IGNORED
FATAL      UNIT DESIGNATOR EXCEEDS TWO DIGITS
WARNING    EXTRA CHARACTERS ..... AFTER FILE NAME IGNORED
WARNING    EXTRA CHARACTER ..... AFTER FILE NAME IGNORED
FATAL      FORMAT DESIGNATOR MISSING
FATAL      MISSING RIGHT PAREN AFTER UNIT IS ASSUMED
WARNING    ASSUMED COMMA AFTER UNIT OR FORMAT -- FOUND .....
FATAL      PREMATURE E.O.S. IN I/O SUBSCRIPT
FATAL      EXCESS LEFT PAREN IN I/O SUBSCRIPT
WARNING    TERMINAL CHARACTER ..... CHANGED TO RIGHT PAREN
COMPLR     IMPLIED DO NOT TERMINATED
ANSI       ERR= IS NON- ANSI
NOTE       FRR= IS IGNORED
ANSI       END= IS NON- ANSI
WARNING    END= IS IGNORED ON WRITE STATEMENT
WARNING    FOUND ..... AFTER FORMAT -- ASSUMED RIGHT PAREN
FATAL      MISSING RIGHT PAREN AFTER FORMAT IS ASSUMED
WARNING    ..... IS NOT A LEGAL KEYWORD
WARNING    ERR= SPECIFIED TWICE
WARNING    END= SPECIFIED TWICE
NOTE       END= IS IGNORED
FATAL      UNFORMATED I/O NOT ALLOWED IN THIS STATEMENT
ANSI       ANSI REQUIRES AN I/O LIST
FATAL      THIS STATEMENT REQUIRES AN I/O LIST
ANSI       LIST DIRECTED I/O IS NON- ANSI
FATAL      ZERO IS AN ILLFGAL UNIT NUMBRER
FATAL      BUFFER DIRECTION INDICATOR MUST BE IN OR OUT
FATAL      BUFFER I/O PARITY INDICATOR MUST BE INTEGER CONSTANT OR VARIABLE
FATAL      SYNTAX -- BEGINNING LEFT PAREN MISSING
FATAL      SYNTAX -- EXPECTED COMMA AFTER UNIT DESIGNATOR -- FOUND .....
FATAL      SYNTAX -- EXPECTED RIGHT PAREN AFTER PARITY INDICATOR -- FOUND .....
FATAL      SYNTAX -- EXPECTED LEFT PAREN BEFORE FWA -- FOUND .....
FATAL      BUFFER I/O ADDRESS MUST BE VARIABLE
FATAL      BUFFER I/O PARITY INDICATOR VALUE MUST BE 0 OR 1
FATAL      BUFFER I/O FWA AND LWA MUST BE IN SAME ARRAY COMMON OR EQUIV CLASS
FATAL      BUFFER I/O LWA MUST BE GREATER THAN OR EQUAL TO FWA
FATAL      EXCESS LEFT PAREN IN I/O LIST
FATAL      SYNTAX IN I/O IMPLIED DO
FATAL      EXCESS RIGHT PAREN IN I/O LIST
FATAL      EXPRESSION IN INPUT LIST IS ILLEGAL
FATAL      CONSTANT IN INPUT LIST IS ILLEGAL
NOTE       IMPLIED LOOP IS REDUCED
FATAL      / NOT ALLOWED IN FORMATTED I/O OR UNFORMATED INPUT LIST
FATAL      FORMATTED I/O OR UNFORMATED INPUT LIST CANNOT END WITH COMMA

```

FTN(TS) COMPILE-TIME ERROR MESSAGES

FORMAT REFERENCED IN I/O.

ANSI           FORMAT INDICATOR ..... MUST BE ARRAY  
 FATAL         FORMAT INDICATOR MUST NOT BE EXPRESSION  
 WARNING      I/O LIST IGNORED WHEN USING NAMELIST  
 ANSI         NAMELIST I/O IS NON- ANSI  
 NOTE         REDUNDANT PAREN IN I/O LIST

ENCODE / DECODE.

FATAL         EXPECTED LEFT PAREN BEFORE COUNT -- FOUND .....  
 FATAL         EXPECTED COMMA AFTER COUNT -- FOUND .....  
 FATAL         FORMAT INDICATOR ..... IS NAMELIST NAME  
 FATAL         EXPECTED COMMA AFTER FORMAT INDICATOR -- FOUND .....  
 FATAL         EXPECTED RIGHT PAREN AFTER STRING ADDRESS -- FOUND .....  
 FATAL         PREMATURE E.O.S. IN ENCODE OR DECODE  
 FATAL         STRING ADDRESS MUST BE ARRAY ELEMENT OR SIMPLE VARIABLE  
 FATAL         ILLEGAL CHARACTER COUNT  
 FATAL         ILLEGAL FORMAT INDICATOR .....

NAMELIST PROCESSING.

FATAL         SYNTAX ERROR IN NAMELIST  
 FATAL         MISSING SLASH ON GROUP NAME  
 FATAL         EXPECTED NAME -- FOUND .....  
 FATAL         GROUP NAME ..... PREVIOUSLY DEFINED  
 FATAL         VARIABLE DIMENSION NOT PERMITTED IN NAMELIST

PARENTHESIS MIS-MATCH.

FATAL         TOO FEW RIGHT PAREN  
 FATAL         TOO FEW LEFT PAREN  
 FATAL         TOO FEW RIGHT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT .....  
 FATAL         TOO FEW LEFT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT .....

LEVEL PROCESSING

FATAL         INTEGER 1,2 OR 3 MUST FOLLOW LEVEL  
 FATAL         COMMA MUST FOLLOW LEVEL NUMBER  
 FATAL         MISSING VARIABLE OR ARRAY NAME IN LEVEL LIST  
 WARNING      MULTIPLY DEFINED LEVEL FOR NAME ..... -- IGNORED  
 FATAL         COMMA OR E.O.S. MUST FOLLOW LEVEL LIST NAME  
 FATAL         NAME ..... IS LEVEL AND MUST BE COMMON OR DUMMY ARGUMENT  
 FATAL         LEVEL CONFLICT IN COMMON BLOCK .....  
 NOTE         NOT ALL NAMES IN COMMON BLOCK ..... ARE IN LEVEL STATEMENT  
 FATAL         NAME ..... IS IN EQUIV. GROUP THAT IS LEVEL AND MUST BE COMMON  
 FATAL         NAME ..... IS IN EQUIV. GROUP THAT HAS LEVEL CONFLICT  
 FATAL         LEVEL 3 NAME ..... MAY NOT OCCUR IN THIS STATEMENT

## FTN(ITS) COMPILE-TIME ERROR MESSAGES

## MASTER LOOP.

```

FATAL      THIS IS NOT A FORTRAN STATEMENT
FATAL      STATEMENT FUNCTION DEFINITION MUST OCCUR BEFORE FIRST EXECUTABLE
FATAL      EXECUTABLE STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM
ANSI       ONLY #ANS.CONT# CONTINUATN CARDS ARE PERMITTED
WARNING    INITIAL LINE IS CONTINUATN
FATAL      ..... STATEMENT MUST OCCUR BEFORE FIRST STATEMENT FUNCTION DEFINITION
FATAL      THIS STATEMENT MAY NOT BE A DO TERMINAL
-----
FATAL      END LINE ABSENT
FATAL      HEADER CARD NOT FIRST STATEMENT -- IGNORED
FATAL      IMPLICIT STATEMENT MUST OCCUR BEFORE ANY DECLARATIV
ANSI       ..... BLANK STATEMENTS WERE IGNORED
ANSI       MULTIPLE STATEMENT PER CARD NOT PERMITTED
-----
WARNING    MULTIPLE STATEMENT IGNORED AFTER END
-----
WARNING    MULTIPLE STATEMENT IGNORED AFTER LOADER DIRECTIVE
-----
WARNING    MISPELLED KEYWORD -- ..... 2 ASSUMED
FATAL      NULL STATEMENT WITH LABEL -- CONTINUE ASSUMED
FATAL      PROGRAM LENGTH EXCEEDS 2**17-1
FATAL      TABLE OVERFLOW -- INCREASE F.L. AND RERUN
-----
FATAL      NO COMPILE TO CORE -- NOT ENOUGH CORE
-----
FATAL      SCRATCH FILE SPILLS TO DISK
-----
FATAL      REFERENCE FILE SPILLS TO DISK
FATAL      PREMATURE E.O.S OR MISSING RIGHT PAREN
WARNING    PREMATURE E.O.S. OR EXTRA TRAILING SEPARATOR .....
FATAL      PREMATURE E. O. S.
FATAL      RETURNS PARAMETER ..... NOT ALLOWED IN THIS STATEMENT
FATAL      STATEMENT LABEL ..... REFERENCED BUT NOT DEFINED
-----
WARNING    TRIVIAL PROGRAM UNIT IGNORED
WARNING    FOLLOWING STATEMENT DOES NOT BEGIN WITH SEQUENCE NUMBER -- COMMENT ASSUMED
WARNING    NO PATH TO THIS STATEMENT
WARNING    NO PATH TO THE ENTIRE RANGE OF DO
WARNING    NULL LOADER DIRECTIVE IS IGNORED
FATAL      LOADER DIRECTIVE MUST BEGIN WITH LEFT PAREN
-----
FATAL      NO COMPILE TO CORE -- LOADER DIRECTIVE
-----
FATAL      NO COMPILE TO CORE -- INTERMIXED COMPASS
NOTE       LOADER DIRECTIVE SHOULD BE CONTAINED ON ONE CARD
FATAL      SYNTAX ERROR IN PROGRAM UNIT NAME
NOTE       MISSING PROGRAM STATEMENT -- PROGRAM START. ASSUMED

```

## ENTRY PROCESSOR.

```

WARNING    NAME ..... PREVIOUSLY DEFINED -- ENTRY STATEMENT IGNORED
WARNING    ENTRY INSIDE DO LOOP IS IGNORED
WARNING    MISSING NAME -- ENTRY STATEMENT IGNORED
WARNING    ENTRY STATEMENT IGNORED IN MAIN PROGRAM
WARNING    ILLEGAL NAME -- ENTRY STATEMENT IGNORED

```

FTN(TS) COMPILE-TIME ERROR MESSAGES

SUBSCRIPT PROCESSOR.

NOTE MISSING SUBSCRIPTS ON ..... ARE ASSIGNED VALUE OF ONE  
 ANSI FORM OF SUBSCRIPT .....2 ON ..... NOT DEFINED IN ANSI  
 NOTE SUBSCRIPT .....2 FOR ..... NOT INTEGER -- TRUNCATED  
 FATAL SUBSCRIPT .....2 ON ..... MUST NOT BE LOGICAL  
 ANSI A TERM IN SUBSCRIPT .....2 ON ..... IS NOT INTEGER  
 FATAL TOO MANY SUBSCRIPTS ON .....

STATEMENT LABEL PROCESSOR.

FATAL MULTIPLY DEFINED STATEMENT LABEL .....  
 FATAL PREVIOUS REFERENCE TO THIS DO LABEL IS ILLEGAL  
 FATAL DO LOOP .....2 PREVIOUSLY DEFINED -- ILLEGAL NESTING  
 FATAL USAGE CONFLICT -- .....2 PREVIOUSLY DEFINED AS DO TERMINAL  
 FATAL USAGE CONFLICT -- .....2 PREVIOUSLY DEFINED AS FORMAT  
 FATAL ILLEGAL TRANSFER TO .....2 FORMAT  
 FATAL ILLEGAL REFERENCE TO STATEMENT LABEL ..... AS A FORMAT  
 FATAL PREVIOUS REFERENCE TO DO LABEL .....2 IS ILLEGAL  
 FATAL PREVIOUS REFERENCE TO FORMAT LABEL .....2 IS ILLEGAL  
 FATAL ILLEGAL TRANSFER TO DO .....2 TERMINATOR  
 FATAL ILLEGAL REFERENCE TO FORMAT STATEMENT LABEL .....2  
 FATAL PREVIOUS TRANSFER TO ..... IS FROM OUTSIDE CURRENT DO  
 FATAL USAGE CONFLICT -- ..... PREVIOUSLY USED AS A FORMAT LABEL  
 FATAL STATEMENT LABEL ..... EXCEEDS FIVE DIGITS  
 FATAL STATEMENT LABEL ..... CONTAINS NON- DIGIT  
 WARNING STATEMENT LABEL ON NON- EXECUTABLE IGNORED  
 FATAL THE TERMINAL STATEMENT OF DO .....2 PRECEDES THE DO DEFINITION  
 FATAL STATEMENT LABEL EXPECTED BUT NOT FOUND  
 FATAL STATEMENT LABEL ZERO IS ILLEGAL  
 NOTE STATEMENT LABEL ZERO IGNORED

SUBROUTINE/FUNCTION REFERENCE PROCESSING.

WARNING NUMBER OF ARGUMENTS IN REFERENCE TO ..... IS NOT CONSISTENT  
 FATAL ARGUMENT MODE MUST AGREE WITH TYPE DEFINED FOR LIBRARY FUNCTION .....  
 ANSI ..... IS DEFINED BY C.D.C. TO BE INTRINSIC  
 ANSI ..... IS DEFINED BY C.D.C. TO BE B.E.F.  
 FATAL ARGUMENT COUNT ON ..... EXCEEDS #MAX.SARG#  
 FATAL ARGUMENT COUNT ON .....3 MUST BE MORE THAN ONE  
 FATAL ARGUMENT COUNT DIFFERS FROM DEFINED FOR INTRINSIC .....3  
 WARNING FUNCTION NAME IS NEVER ASSIGNED A VALUE

STOP / PAUSE PROCESSING

WARNING UNKNOWN FORM -- BLANK ASSUMED  
 NOTE HOLLERITH ARGUMENT MUST NOT EXCEED 70 CHARACTERS  
 NOTE CONSTANT EXCEEDS 5 DIGITS -- TRUNCATED  
 ANSI PAUSE MAY NOT BE A DO TERMINAL



## FTN(TS) COMPILE-TIME ERROR MESSAGES

ANSI HOLLERITH ARGUMENT IS NON-ANSI

## PROGRAM STATEMENT FILE DECLARATION.

WARNING FILE ..... PREVIOUSLY DEFINED -- IGNORED  
 FATAL EXPECTED RIGHT PAREN OR COMMA -- FOUND .....  
 WARNING ILLEGAL BUFFER LENGTH FOR FILE ..... -- DEFAULT USED  
 WARNING BUFFER LENGTH FOR FILE ..... EXCEEDS 3600008 -- DEFAULT USED  
 WARNING ONLY #MAX.PARG# FILES ARE PERMITTED -- EXCESS IGNORED  
 WARNING FILE .....2 NOT DEFINED -- EQUIV. IGNORED  
 FATAL EQUAL SIGN MUST BE FOLLOWED BY NAME OR NUMBER OR SLASH  
 WARNING ILLEGAL RECORD LENGTH FOR FILE ..... -- DEFAULT USED  
 WARNING RECORD LENGTH FOR FILE ..... EXCEEDS 2\*\*17-1 -- DEFAULT USED  
 FATAL FILE NAME .....2 EXCEEDS 6 CHARACTERS  
 FATAL SYNTAX ERROR IN PROGRAM CARD -- SCAN STOPPED AT .....

## TRANSLATION OF VARIABLE SECTION.

FATAL REFERENCE TO VARIABLE ..... AS A FUNCTION OR ARRAY  
 FATAL REFERENCE TO FUNCTION ..... REQUIRES AN ARGUMENT LIST  
 FATAL REFERENCE TO INTRINSIC ..... REQUIRES AN ARGUMENT LIST  
 FATAL SUBROUTINE ..... REFERENCE AS A FUNCTION  
 ANSI ARRAY ..... MISSING SUBSCRIPT -- FIRST ELEMENT ASSUMED  
 FATAL SYNTAX ERROR -- EXPECTED SYMBOL BUT FOUND ..... -- SCAN OF CARD STOPPED  
 -----  
 FATAL NAME EXCEEDS 7 CHARACTERS -- TRUNCATED TO .....  
 FATAL LEFT SIDE OF EQUAL SIGN IS ILLEGAL

## TYPE PROCESSOR.

WARNING TYPING OF ..... IGNORED -- PRIOR TYPING RETAINED  
 FATAL SYNTAX -- EXPECTED COMMA FOUND .....  
 WARNING CONFLICT IN RANGE INDICATOR -- FIRST HOLDS  
 NOTE \*SIZE SPEC ON ..... IS IGNORED  
 ANSI \*SIZE WAS USED ..... TIMES  
 ANSI THE WORD TYPE IS NOT PERMITTED  
 FATAL TYPE MUST BE FOLLOWED BY A TYPE INDICATOR  
 FATAL ..... IS NOT A LEGAL TYPE  
 FATAL EXPECTED LEFT PAREN -- FOUND .....  
 FATAL RANGE INDICATOR ..... MUST BE A LETTER  
 WARNING RANGE INDICATOR ..... NOT SINGLE LETTER -- TRUNCATED TO .....3  
 FATAL RANGE BACKWARD -- .....2 NOT LESS THAN ..... -- TRUNCATED  
 WARNING TRIVIAL RANGE -- .....2 SAME AS .....  
 FATAL EXPECTED RIGHT PAREN -- FOUND .....  
 FATAL EXPECTED E.O.S. -- FOUND AND IGNORED .....

## FTN(ITS) COMPILE-TIME ERROR MESSAGES

## I/O UNIT DESIGNATOR.

FATAL I/O UNIT DESIGNATOR MUST BE INTEGER  
FATAL I/O UNIT DESIGNATOR MUST BE SIMPLE VARIABLE  
WARNING I/O FILE ..... NOT DEFINED  
FATAL CHARACTER ..... NOT DEFINED IN STANDARD FORTRAN -- SCAN OF CARD STOPPED  
FATAL USAGE CONFLICT -- ..... PREVIOUSLY USED AS .....?  
ANSI DOES NOT ALLOW SHORT FORMS OF LOGICAL OPERATORS OR CONSTANTS  
FATAL USAGE CONFLICT -- ..... IS .....2 AND CANNOT BE .....3  
WARNING SYNTAX -- EXPECTED E.O.S. -- FOUND AND IGNORED .....  
WARNING SYNTAX -- EXPECTED E.O.S. -- FOUND AND IGNORED .....

## EXP PROCESSOR

FATAL OPERAND TO \*\* OPERATOR MUST NOT BE LOGICAL  
FATAL COMPLEX MUST ONLY BE RAISED TO INTEGER POWER  
NOTE ZERO \*\* ZERO -- RESULTS INDEFINITE  
NOTE INTEGER \*\* NEGATIVE CONSTANT -- RESULTS ZERO  
WARNING EVALUATION OF CONSTANTS WILL RESULT IN OUT OF RANGE OR INDEFINITE RESULTS

\*SYSTEM\* ERROR -- COMPILER MALFUNCTION

COMPLR \*\*\*\* COMPILE ERROR \*\*\*\*

## EXECUTION DIAGNOSTICS

Execution diagnostics are printed on the source listing in the following format:

ERROR NUMBER x DETECTED BY routine AT ADDRESS y

or

ERROR NUMBER x DETECTED BY routine

followed by

CALLED FROM routine AT ADDRESS z

or

CALLED FROM routine AT LINE d

y and z are octal addresses, x is a decimal error number, and d is a decimal line number as printed on the source listing.

Example:

```
PROGRAM EXERR          74/74  OPT=1
```

```
1          PROGRAM EXERR(INPUT,OUTPUT)
           N=5
           GO TO (1,2,3),N
           1  N=N+1
           5  2  N=N+2
           3  STOP
           END
```

```
CARD NR. SEVERITY  DETAILS      DIAGNOSIS OF PROBLEM
```

```
3      I          AN IF STATEMENT MAY BE MORE EFFICIENT
                    THAN A 2 OR 3 BRANCH COMPUTED GO TO
                    STATEMENT.
```

```
ERROR IN COMPUTED GOTO STATEMENT- INDEX VALUE INVALID
```

```
ERROR NUMBER 1      DETECTED BY GOTOER= AT ADDRESS 000004
CALLED FROM EXERR  AT LINE 3
```

In the following list of execution diagnostics under class, the letters are interpreted as follows:

F = Fatal	D = Debug
I = Informative, non-fatal	T = Trace
A = Always	

The severity level (fatal or non-fatal) of any error can be changed by a call to SYSTEMC (see section III-3).

Error No.	Class	Message	Routine
1	F A	ERROR IN COMPUTED GO TO STATEMENT INDEX VALUE INVALID	GOTOER=
2	I A	ARGUMENT ABS VALUE.GT.1 ARGUMENT INFINITE ARGUMENT INDEFINITE	ACOSIN=(ACOS)
3	I A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	ALOG
4	I A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	ALOG10
5	I A	ARGUMENT ABS VALUE.GT.1 ARGUMENT INFINITE ARGUMENT INDEFINITE	ACOSIN=(ASIN)
6	I A	ARGUMENT INDEFINITE	ATAN
7	I A	ARGUMENT VECTOR ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE	ATAN2
8	I A	ARGUMENT TOO LARGE ARGUMENT INFINITE ARGUMENT INDEFINITE	CABS
9	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ZTOI
10	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CCOS
11	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CEXP
12	I T	ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	CLOG

Error No.	Class	Message	Routine
13	I A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	COS (in SINCOS=)
14	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	CSIN
15	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	CSQRT
16	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOX (D**X)
17	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	DATAN
18	I T	X=Y=0.0 <sup>†</sup> INFINITE ARGUMENT INDEFINITE ARGUMENT	DATAN2
19	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOD (D**D)
20	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOI (D**I)
21	I T	FLOATING OVERFLOW IN D**REAL(Z) <sup>†</sup> ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(D) <sup>†</sup> TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	DTOZ (D**Z)

---

<sup>†</sup>X and Y=real; Z=complex; D=double precision

Error No.	Class	Message	Routine
22	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	DCOS
23	I T	ARGUMENT TOO LARGE, FLOATING OVERFLOW INFINITE ARGUMENT INDEFINITE ARGUMENT	DEXP
24	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DLOG
25	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DLOG10
26	I T	DOUBLE PRECISION INTEGER EXCEEDS 96 BITS 2ND ARGUMENT ZERO INFINITE ARGUMENT INDEFINITE ARGUMENT	DMOD
28	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	DSIN
29	I T	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	DSQRT
30	I A	ARGUMENT TOO LARGE, FLOATING OVERFLOW ARGUMENT INFINITE ARGUMENT INDEFINITE	EXP
31	I T	INTEGER OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER	IT0J

Error No.	Class	Message	Routine
33	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOD (X**D)
34	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOI (X**I)
35	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	XTOY (X**Y)
36	I A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	SIN (in SINCOS=)
37	I T	ILLEGAL SENSE LITE NUMBER	SLITE
38	I T	ILLEGAL SENSE LITE NUMBER	SLITET
39	I A	ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	SQRT
40	I T	ILLEGAL SENSE SWTICH NUMBER	SSWTCH
41	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	TAN
42	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	TANH
44	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ITOD (I**D)

Error No.	Class	Message	Routine
45	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	ITOX (I**X)
46	I T	FLOATING OVERFLOW IN I**REAL(Z) <sup>†</sup> ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(I) <sup>†</sup> TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	ITOZ (I**Z)
47	I T	FLOATING OVERFLOW IN X**REAL(Z) <sup>†</sup> ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(X) <sup>†</sup> TOO LARGE INFINITE ARGUMENT	XTOZ
48	F D	FATAL ERROR ENCOUNTERED DURING PROGRAM EXECUTION DUE TO COMPILATION ERROR	FTNERR=
49	I A I A I A	COMMA MISSING AT END OF RECORD – COMMA ASSUMED NAMELIST DATA TERMINATED BY EOF NOT \$ CONSTANTS MISSING AT END OF RECORD – NEXT RECORD READ	NAMIN=
50	F A	FATAL ERROR IN LOADER	OVERLA=
51	I A	Set by user via subroutine SYSTEM or SYSTEMC.	
52	F A	Set by user via subroutine SYSTEM or SYSTEMC. Error numbers larger than those listed in this table become error 52.	
55	F A	END-OF-FILE ENCOUNTERED, FILENAME- - - -xxxxxxx	BUFIN=
56	F A	WRITE FOLLOWED BY READ, FILENAME- - - -xxxxxxx	BUFIN=
57	F A	AREA SPECIFICATION SPANS SCM/LCM	BUFIO=
58	F A	BUFFER DESIGNATION BAD- - FWA.GT.LWA	BUFIO=
59	F A	BUFFER SPECIFICATION BAD- -FWA.GT.LWA	BUFOUT=
62	F A	FILENAME NOT DECLARED-xxxxxxx	GETFIT=

<sup>†</sup>Z=complex; I=integer, X=real



Error No.	Class	Message	Routine
63	F A	END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx	INPB=
65	F A	END-OF-FILE ENCOUNTERED, FILENAME-xxxxxxx	INPC=
66	F A	NAMELIST NAME NOT FOUND-xxxxxxx	NAMIN=
	F A	INCORRECT SUBSCRIPT	
	F A	TOO MANY CONSTANTS	
	F A	, ( \$ OR = EXPECTED, MISSING	
	F A	VARIABLE NAME NOT FOUND-xxxxxxx	
	F A	CONSTANT MISSING	
67	F A	DECODE RECORD LENGTH .LE. 0 DECODE LCM RECORD .GT. 150 CHARACTERS	DECODE=
68	F A	*ILL-PLACED NUMBER OR SIGN	FMTAP=
	F A	*ILLEGAL FUNCTIONAL LETTER	
69	F A	*IMPROPER PARENTHESIS NESTING	FMTAP=
70	F A	*EXCEEDED RECORD SIZE	FMTAP=
71	F A	*SPECIFIED FIELD WIDTH ZERO	FMTAP=
	F A	*BAD VALUE FOR = OR V	
72	F A	*FIELD WIDTH .LE. DECIMAL WIDTH	FMTAP=
73	F A	*HOLLERITH FORMAT WITH LIST	FMTAP=
78	F A	*ILLEGAL DATA IN FIELD .↑.	INCOM=
79	F A	*DATA OVERFLOW .↑.	INCOM=
83	F A	OUTPUT FILE LINE LIMIT EXCEEDED	OUTC= NAMOUT=
85	F A	ENCODE CHARACTER/RECORD .LE. 0 ENCODE LCM RECORD .GT. 150 CHARACTERS	ENCODE=
88	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx	INPB=
89	F A	LIST EXCEEDS DATA, FILENAME-xxxxxxx	INPB=
90	F A	PARITY ERROR READING (BINARY) FILE-xxxxxxx	INPB=
91	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx	INPC=
92	F A	PARITY ERROR READING (CODED) FILE-xxxxxxx	INPC= NAMIN=

Error No.	Class	Message	Routine
93	F A	PARITY ERROR ON LAST READ ON FILE-xxxxxxx	OUTB=
94	F A	PARITY ERROR ON LAST READ ON FILE-xxxxxxx	OUTC=
97	F A	INDEX NUMBER ERROR	RANMS=
98	F A	FILE ORGANIZATION OR RECORD TYPE ERR	RANMS=
99	F A	WRONG INDEX TYPE	RANMS=
100	F A	INDEX IS FULL	RANMS=
101	F A	DEFECTIVE INDEX CONTROL WORD	RANMS=
102	F A	RECORD LENGTH EXCEEDS SPACE ALLOCATED	RANMS=
103	F A	6RM/7DM I/O ERR NUMBER 000	RANMS=
104	F A	INDEX KEY UNKNOWN	RANMS=
112	F A	ECS UNIT HAS LOST POWER OR IS IN MAINTENANCE MODE	WRITEC
113	F A	ECS READ PARITY ERROR	READEC
114	F A	CONNEX CHARACTER CODE CONVERSION IS OUT OF RANGE	CONDIS
115	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO SMALL	EXP
116	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	HYP=(COSH)
117	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	HYP=(SINH)

---

FORSYS= is a multiple entry routine which handles program initialization, error tracing, diagnostic printing, and termination of output buffers. FORSYS= has the following entry points:

Q8NTRY.	Initializes input/output buffer parameters.
STOP.	Enters STOP in dayfile and begins END processing.
EXIT	Enters EXIT in dayfile and begins END processing.
END.	Terminates all output buffers and prints an error summary. If in a secondary overlay, returns control to the calling primary overlay; if in a primary overlay, returns control to the main overlay; otherwise, advances to the next job step.
ABNORM.	Issues an error message to the dayfile and aborts the job step.
SYSEND.	Closes all files.
SYSERR.	Handles error tracing and diagnostic printing.
IOERR.	Issues a fatal record manager diagnostic to output file.

A routine SYS=AID interfaces, through its entry point SYSAID., between FORSYS= and math library routines.

### ERROR PROCESSING

All routines which perform error checking call the routine FORSYS= at the entry point SYSERR. to issue diagnostic and traceback information. For a non-fatal (informative) error, this information is printed, and control is returned to the routine that called the routine detecting the error. For a fatal error, diagnostic and traceback information is printed, all output buffers are flushed, and the job is terminated.

### EXTENDED ERROR PROCESSING

#### SYSTEM

The subroutine SYSTEM enables the user to issue an execution-time error message.

CALL SYSTEM(errnum, mesg)

errnum	Error number decimal: an integer constant or integer variable with a value of 0 to 9999. The error numbers listed in section III-2 retain the severity association indicated there. Error numbers 51 (non-fatal) and 52 (fatal) are reserved for the user.
mesg	Error message: entered as a Hollerith constant with the first character used as a carriage control character and not printed.

If error number zero is entered, the message is ignored, the output buffers are flushed, and control is returned to the calling program.

The file OUTPUT should be declared before SYSTEM is called. Otherwise, no errors are printed; and a message to this effect is entered in the dayfile.

Each line is printed unless the line limit of the OUTPUT buffer is exceeded, in which case the job is terminated.

**Example:**

CALL SYSTEM (3, ≠ CHECK DATA ≠)

**SYSTEMC**

SYSTEMC enables the user to alter the contents of the error table, which contains specifications that regulate error processing.

In the error table, the first entry corresponds to error number 1, the second to error number 2, and so on. Each entry has the following format:

59	51	43	31	20	17	0
print frequency	frequency increment	print limit	detection total	F/A N/N F/A	user-specified recovery address	

**print frequency** Ordinarily, print frequency value is 0. If the value is changed to n by a call to SYSTEMC, diagnostic and traceback information is listed every nth time until the print limit is reached.

**frequency increment** Ordinarily, frequency increment value is 1. This specification can be changed by a call to SYSTEMC if the call specifies print frequency as 0. When frequency increment is 0, diagnostic and traceback information is not listed; when it is 1, such information is listed until the print limit is reached; when the frequency increment is n>1, such information is listed only the first n times unless the print limit is reached first.

**print limit** Ordinarily, print limit value is 7777 octal. It can be changed by a call to SYSTEMC.

**detection total** Detection total is a running count of the number of times an error occurs. The final value is reported in the error summary issued at end of job if SYSTEMC is called during execution.

**F/NF** This bit specifies the severity of the error: 1 indicates a fatal error; 0, non-fatal. The severities of system defined errors are given in section III-2. All errors defined by the user with these numbers in a call to SYSTEM retain the specified severity. The severity of any error can be changed by a call to SYSTEMC, however.

A/NA The A/NA bit is ignored unless a non-standard recovery address is specified; it can be set only during assembly of SYSTEMC. When this bit is set, the address in an auxiliary table is passed in the third word of the secondary argument list to the recovery routine. Each word in the auxiliary table must have the error number in its upper 10 bits, so that the address of the first error number match is passed to the recovery routine. An entry in the auxiliary table for an error is not limited to any specific number of words.

user-specified recovery address This address is specified in a call to SYSTEMC.

SYSTEMC is called by the following statement:

CALL SYSTEMC (errnum,speclist)

errnum	Error number for which non-standard recovery is to be implemented.
speclist	Integer array containing error processing specifications in consecutive locations: word 1 F/NF (1 = fatal, 0 = non-fatal). word 2 Print frequency word 3 Frequency increment word 4 Print limit word 5 User-specified error recovery routine address word 6 Maximum traceback limit applicable to all errors; this limit is 20 unless changed by a call to SYSTEMC

A negative value for any word in the speclist indicates that the current value of that specification is not to be changed. A user-specified error recovery routine activated by a call to SYSTEMC can be canceled by a subsequent call with word 5 of the speclist set to zero.

If SYSTEMC has been called, an error summary is issued at job termination indicating the number of times each error occurred since the first call to SYSTEMC.

For an error detected by a routine in the math library, a user-supplied error recovery routine should be a function subprogram of the same type as the FORTRAN function detecting the error. For any other error, a user-supplied error recovery routine should be a subroutine subprogram.

When an error previously referenced by a SYSTEMC call is detected, the following sequence of operations is initiated:

1. Diagnostic and traceback information is printed in accordance with the specification in the pertinent error table entry. The traceback information is terminated for any of the following conditions:
  - Calling routine is a program.
  - Maximum traceback limit is reached.
  - No traceback information is supplied.

2. If the SYSTEMC call references a user-specified error recovery routine address, SYSTEMC, FORSYS=, and the routine detecting the error are delinked from the calling chain, and the user-supplied error recovery routine is entered.
3. If the error is non-fatal, control returns to the routine that called the routine detecting the error. An error summary is printed at job termination.
4. If the error is fatal, all output buffers are flushed, an error summary is printed, and the job is terminated.

If a non-standard recovery address is specified in the SYSTEMC call, the following information is available to the user recovery routine:

**Register    Contents**

A1	<p>Address of argument list passed to routine detecting the error for errors detected by a math library routine.</p> <p>Address of the FIT for error 103.</p> <p>Undefined for all other errors.</p>
X1	<p>Address of the first argument in the list for errors detected by a math library routine.</p> <p>Undefined for all other errors.</p>
A0	<p>Address of argument list of routine that called the routine detecting the error.</p>
B1	<p>Address of a secondary argument list containing, in successive words:</p> <p style="padding-left: 40px;">Error number associated with this error.</p> <p style="padding-left: 40px;">Address of message associated with this error.</p> <p style="padding-left: 40px;">Address within auxiliary table if A/NA bit set; otherwise 0.</p> <p style="padding-left: 40px;">In upper 30 bits, instruction consisting of RJ to SYSERR.j; in lower 30 bits, address of traceback information for routine detecting the error.</p> <p>Information in the secondary argument list is not available to user supplied error recovery routines coded in FORTRAN.</p>
A2	<p>Address of error table entry for this error.</p>
X2	<p>Contents of error table entry for this error.</p>

Example 2:

```
PROGRAM EXPECT(OUTPUT)
DIMENSION IRAY(6)
DATA IRAY /6 * (-0)/
C      SET PRINT LIMIT TO ZERO
      IRAY(4)=0

      X = EXP(800.0)
      X = EXP(-800.0)

C      CALL SYSTEMC TO INHIBIT PRINTING OF ERROR 115
C      AND START ERROR SUMMARY ACCUMULATION
      CALL SYSTEMC (115,IRAY)
      PRINT *, # #
      PRINT *,#*****SYSTEMC IS CALLED TO SUPPRESS PRINTING#
X      # OF ERROR 115#

      X = EXP(800.0)
      X = EXP(-800.0)

      PRINT *,# #
      PRINT *,#*****ERROR 115 DETECTED BUT NOT PRINTED#
      END
```

```
ARGUMENT TOO LARGE, FLOATING OVERFLOW
ERROR NUMBER  30  DETECTED BY EXP
```

```
ARGUMENT TOO SMALL
ERROR NUMBER  115 DETECTED BY EXP
```

```
*****SYSTEMC IS CALLED TO SUPPRESS PRINTING OF ERROR 115
```

```
ARGUMENT TOO LARGE, FLOATING OVERFLOW
ERROR NUMBER  30  DETECTED BY EXP
```

```
*****ERROR 115 DETECTED BUT NOT PRINTED
```

```
ERROR SUMMARY
ERROR      TIMES
0030      0001
0115      0001
```

Program EXPECT illustrates a non-standard error recovery in a math library routine and how to suppress the printing of error message 115.

Example:

```
PROGRAM EXAMPL(TAPE1,OUTPUT)
EXTERNAL ITSOK
DIMENSION NARRAY(6)
DATA NARRAY/6*(-1)/
NARRAY(1) = 0
NARRAY(5) = LOCF(ITSOK)
NARRAY(6) = 1
CALL SYSTEMC(66,NARRAY)
NAMelist/DATA1/A,B
READ (1, DATA1)
REWIND 1
NAMelist/DATA2/A,B
READ (1, DATA2)
NAMelist/DATAOUT/A,B
PRINT DATAOUT
STOP
END
SUBROUTINE ITSOK
PRINT 10
10 FORMAT (*0DATA SET NAMED ABOVE NOT USED*)
RETURN
END
```

Input:

```
$DATA2
A = 3.,
B = 4.,
$
```

Output:

```
NAMelist NAME NOT FOUND - DATA1
ERROR NUMBER 0066 DETECTED BY NAMIN= AT ADDRESS 000435
```

```
DATA SET NAMED ABOVE NOT USED
```

```
$DATAOUT
```

```
A = .3E+01,
```

```
B = .4E+01,
```

```
$END
```

```
ERROR SUMMARY
```

```
ERROR    TIMES
0066     0001
```



## ERRSET

The subroutine ERRSET enables the user to input data without the risk of termination when improper data is encountered.

CALL ERRSET(num,lim)

num Integer variable; returns the current total of accumulated errors.

lim Integer constant or integer variable; the program will not terminate when data errors are encountered until the value of lim has been exceeded. The maximum permissible value of lim is  $2^{59}-1$ .

ERRSET can be used to inhibit job termination when data is being input with a formatted, NAMELIST, or list directed read. It operates only when data is encountered that would ordinarily cause job termination under error number 78 ("ILLEGAL DATA IN FIELD") or error number 79 ("DATA OVERFLOW").

CALL ERRSET initializes an error count location (num) and specifies a maximum limit (lim) on the number of data errors allowed before termination. ERRSET continues in effect for all subsequent READ statements until the limit is reached. ERRSET can be reactivated with another call, which will reinitialize the error count location and reset the limit. A CALL ERRSET with lim specified as zero nullifies a previous call; improper data will then result in job termination as usual.

When improper data is encountered in a formatted or NAMELIST read with ERRSET in effect, the bad data field is bypassed, and processing continues at the next field. When improper data is encountered in a list directed read, control moves to the statement immediately following the READ statement.

### Example:

The following example illustrates the use of ERRSET to suppress normal fatal termination when large sets of data are being processed.

```
      .  
      .  
      .  
      CALL ERRSET(KOUNT,200)  
      READ(1,125)(ARAY(I),I=1,1500)  
125  FORMAT(3F10.5,E10.1)  
      IF (KOUNT.GT.0) GO TO 500  
      .  
      .  
      .  
500  CALL ERRSET(KOUNT,200)  
      READ(1,125)(BRAY(I),I=1,1500)  
      IF (KOUNT.GT.0) GO TO 600  
      .  
      .  
      .
```

```

600 CALL ERRSET(KOUNT,100)
    READ(1,230)(LARRAY(I),I=1,500)
    PRINT 99, KOUNT
    READ(4,127)(MARRAY(I),I=1,500)
    PRINT 99, KOUNT
    READ(4,225)(NARRAY(I),I=1,50)
    .
    .
    .
    .
    .
    IF (KOUNT.GT.0) GO TO 700
    .
    .
    .
700 CALL EXIT
    END

```

When ERRSET is called, a limit of 200 errors is established. The number of errors will be stored in KOUNT. After ARAY is read, KOUNT is checked. If errors occur, the following statements are not processed and a branch is made to statement 500. Had ERRSET not been called, fatal errors would have terminated the program before the branch to statement 500. At statement 500, ERRSET once more initializes the error count, and execution continues.

Example:

```

PROGRAM EXAMPL(TAPE1,OUTPUT)
DIMENSION ACARD(5)
CALL ERRSET(NUM,2)
READ(1,10) (ACARD(I),I=1,5)
10 FORMAT (F4.1)
PRINT 20, NUM
20 FORMAT (1H0, I1, * DATA ERRORS FOUND*//)
PRINT 30, (ACARD(I),I=1,5)
30 FORMAT (1X, F4.1)
STOP
END

```

Input:

```

47.1
25./
48.3
24.6
91.2

```

Output:

```
RECORD NO. 2          25./  
                   ...↑.....123456789012345678901234567890
```

```
* ERROR DATA INPUT * ILLEGAL DATA IN FIELD *↑*  
ERROR NUMBER 0078 DETECTED BY INCOM= AT ADDRESS 000200  
CALLED FROM KRAKER= AT ADDRESS 000353  
CALLED FROM INPC= AT ADDRESS 000074  
CALLED FROM EXAMPL AT LINE 0004
```

```
RECORD NO. 4          24.6  
                   ..↑.....123456789012345678901234567890
```

```
* ERROR DATA INPUT * ILLEGAL DATA IN FIELD *↑*  
ERROR NUMBER 0078 DETECTED BY INCOM= AT ADDRESS 000200  
CALLED FROM KRAKER= AT ADDRESS 000353  
CALLED FROM INPC= AT ADDRESS 000074  
CALLED FROM EXAMPL AT LINE 0004
```

2 DATA ERRORS FOUND

```
47.1  
 0.0  
48.3  
 0.0  
91.2
```

## EXECUTION TIME OPTIONS

### FILE NAME HANDLING

The file names in the PROGRAM statement are placed in RA+2 and the locations immediately following by FORSYS= (entry point Q8NTRY.). RA is the reference address, the absolute address where the user's field length begins. The file name is left justified, and the file's file information table (FIT) address is right justified in the word.

The logical file name (LFN) which appears in the first word of the file information table is determined in one of three ways:

1. If no file names are specified on the LGO or EXECUTE control card, the logical file name is the file name in the PROGRAM statement.

Example:

```
FTN.  
LGO.  
:  
:  
PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)
```

Contents of RA + 2 before execution of Q8NTRY:

000 ... 000  
000 ... 000

Contents of RA + 2 after execution of Q8NTRY:

INPUT ... fit address  
OUTPUT .. fit address  
TAPE1 ... fit address  
TAPE2 ... fit address

The logical file names in the file information table will be:

INPUT  
OUTPUT  
TAPE1  
TAPE2

2. If file names are specified on the LGO or EXECUTE control card, the logical file name is the name specified there. A one-to-one correspondence exists between parameters on the LGO or EXECUTE card and parameters in the PROGRAM statement.

Example:

```
FTN.  
LGO( , , DATA, ANSW )  
.  
.  
.  
PROGRAM TEST2( INPUT, OUTPUT, TAPE1, TAPE2, TAPE3=TAPE1 )
```

Contents of RA + 2 before execution of Q8NTRY:

000 ... 000  
000 ... 000  
DATA .. 000  
ANSW .. 000

Contents of RA + 2 after execution of Q8NTRY:

INPUT ... fit address  
OUTPUT .. fit address  
TAPE1 ... fit address  
TAPE2 ... fit address  
TAPE3 ... fit address of TAPE1

The logical file names in the file information table will be:

INPUT  
OUTPUT  
DATA  
ANSW  
uses TAPE1 file information table

3. If a file name in the PROGRAM statement is equivalenced, the logical file name is the file to the right of the equals sign. A corresponding file name in the LGO or EXECUTE control card is ignored.

Example:

```
FTN.  
LGO( , , DATA, ANSW )  
.  
.  
PROGRAM TEST3( INPUT, OUTPUT, TAPE1=OUTPUT, TAPE2, TAPE3 )
```

Contents of RA+2 before execution of Q8NTRY:

000 ... 000  
000 ... 000  
DATA .. 000  
ANSW .. 000

Contents of RA+2 after execution of Q8NTRY:

INPUT ... fit address  
OUTPUT .. fit address  
TAPE1 ... fit address of OUTPUT  
TAPE2 ... fit address  
TAPE3 ... fit address

The logical file names in the file information table will be:

INPUT  
OUTPUT  
uses OUTPUT file information table  
ANSW  
TAPE3

## PRINT LIMIT SPECIFICATION

A parameter can be specified on the LGO or EXECUTE control card to regulate the maximum number of records that can be written at execution-time on the file OUTPUT. This parameter has the same form as the PL parameter specified at compilation-time on the FTN control card. If specified on the LGO or EXECUTE card, it overrides the value specified either explicitly or by default at compilation-time (see section I-11). This parameter may appear anywhere in the parameter list of the LGO or EXECUTE card; it does not affect the correspondence of file names between the LGO or EXECUTE card and the FTN card.

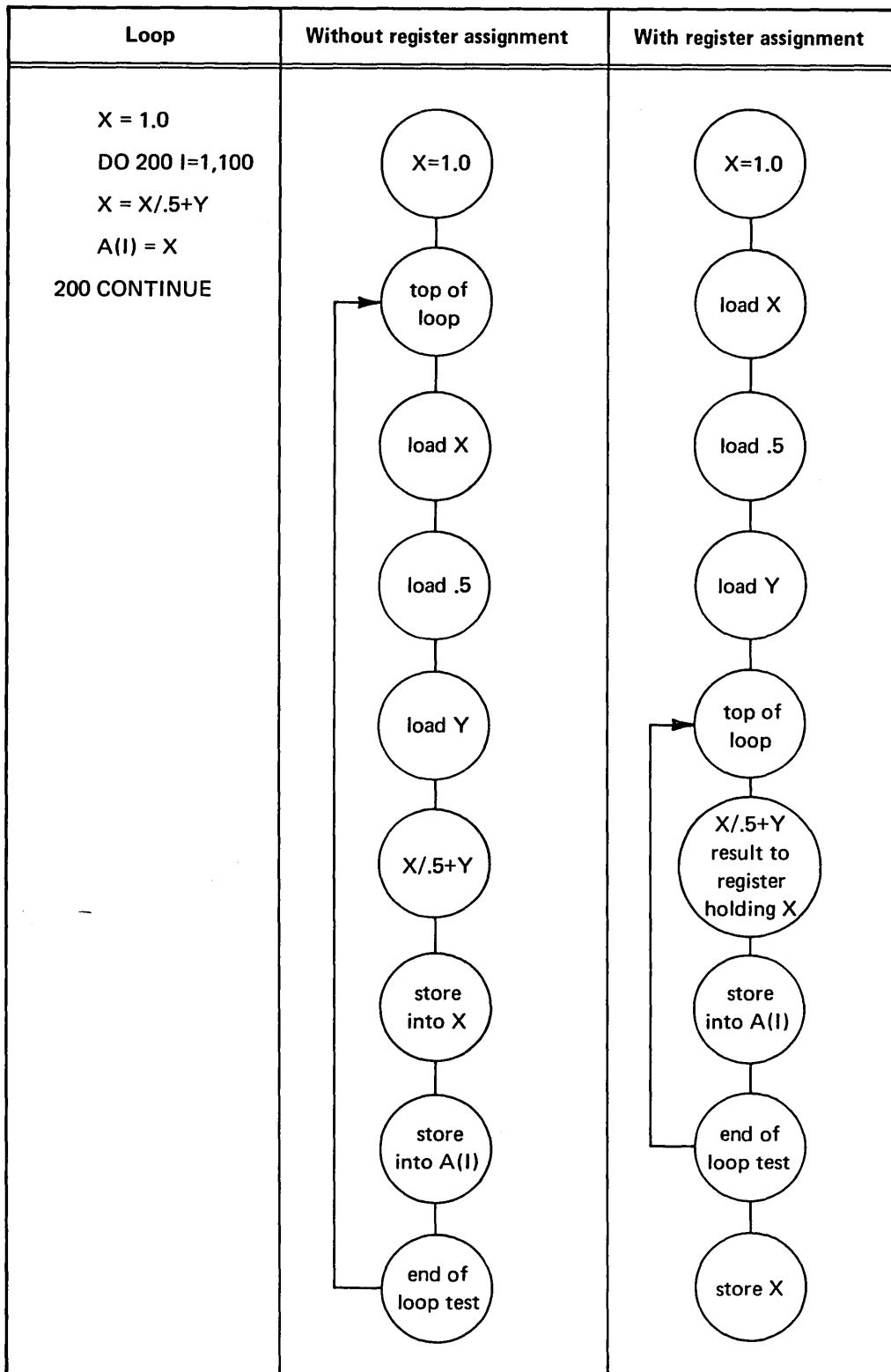
The print limit parameter (specified either at compilation-time or at execution-time) is operative only on files with the name OUTPUT in the first word of its corresponding file information table. Thus, if a file name declared in the PROGRAM statement is superseded at execution-time by the file name OUTPUT as described previously, the print limit parameter will be operative on the original file name. Conversely, if the file name OUTPUT is superseded at execution-time by another file name, the effect of the print limit parameter is nullified.

### Examples:

LGO(PL=2000)

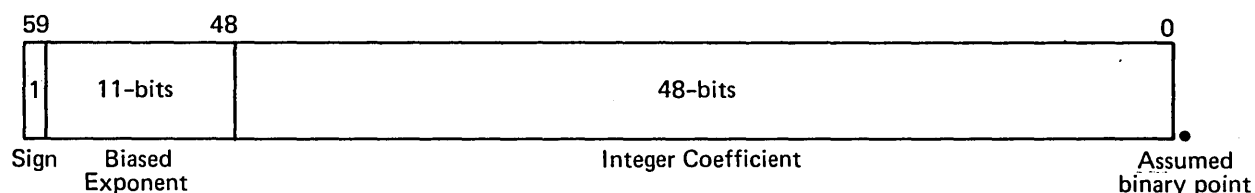
EXECUTE(FILE1,OUTPUT,PL=1000,FILE2)

Example:



**FLOATING POINT ARITHMETIC**

Floating point arithmetic is carried out in the functional units of the central processor.



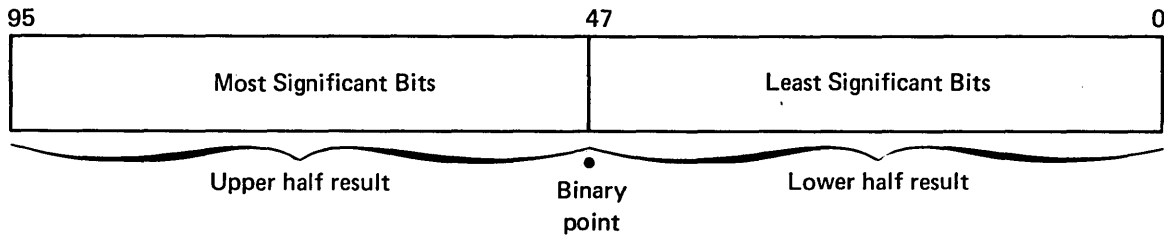
In the 60-bit floating point format shown above, the binary point is considered to be to the right of the coefficient. The lower 48 bits express the integer coefficient, which is the equivalent of approximately 14 decimal digits. The sign of the number is the highest order bit of the packed word. Negative numbers are represented by the one's complement of the 60-bit number.

The exponent portion of the floating point format is biased by 2000 octal. This particular format for floating point numbers was chosen so that the packed form may be treated as a 60-bit integer for sign, equality and zero tests. (Refer to 6400/6500/6600 Computer Systems Reference Manual or 7600 Computer System Reference Manual for details of the hardware pack instruction.)

The following table summarizes the configurations of bits 58 and 59 and the signs of the possible combinations. The number is negative if bit 59 is 1 and positive if bit 59 is 0.

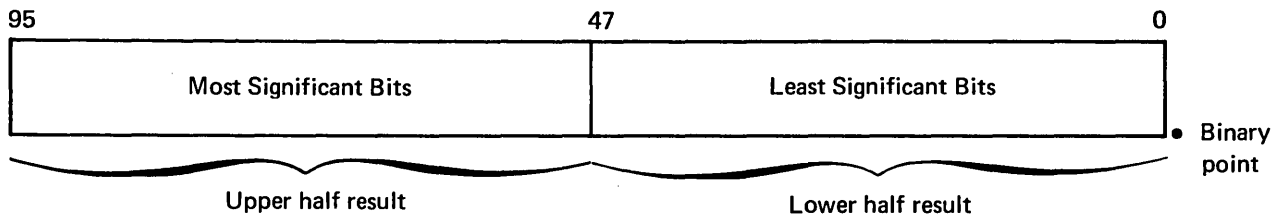
Bit 59	Coefficient Sign	Bit 58	Exponent Sign
0	Positive	1	Positive
0	Positive	0	Negative
1	Negative	0	Positive
1	Negative	1	Negative

To add or subtract two floating point numbers, the floating point ADD unit enters the coefficient with the smaller exponent into the upper half of an accumulator and shifts it right by the difference of the exponents. Then it adds the other coefficient into the upper half of the accumulator. The result is a double length register with the following format:



If single precision is selected, the result is the upper 48 bits of the 96-bit result and the larger exponent. Selecting double precision causes the lower 48 bits of the 96-bit result and the larger exponent minus 60 octal (or 48) to be returned as the result. The subtraction of 60 octal (or 48) is necessary because effectively, the binary point is moved from the right of bit 48 to the right of bit 0.

The multiply units generate 96-bit products from two 48-bit coefficients. The result of a multiply operation is a double length register with the following format:



When unrounded instructions are used, the upper and lower half results with proper exponents may be recovered separately; when rounded instructions are used, only upper half results may be obtained.

If single precision is selected, the upper 48 bits of the product and the sum of the exponents plus 60 octal (or 48) are returned as the result. The addition of 60 octal (or 48) is necessary because, effectively, the binary point is moved from the right of bit 0 to the right of bit 48 when the upper half of the 96-bit result is selected. If double precision is selected, the lower 48 bits of the product and the sum of the exponents is the result.

Some examples of floating point numbers are shown below in octal notation.

Normalized floating point + 1	= 1720 4000 0000 0000 0000
Normalized floating point + 100	= 1726 6200 0000 0000 0000
Normalized floating point -100	= 6051 1577 7777 7777 7777
Normalized floating point $10^{+64}$	= 2245 6047 4037 2237 7733
Normalized floating point $-10^{-64}$	= 6404 2570 0025 6605 5317



## OVERFLOW ( $+\infty$ or $-\infty$ )

Overflow of the floating point range is indicated by an exponent of 3777 for a positive result and 4000 for a negative result. These are the largest exponent values that can be represented in floating point format, as shown in the floating point table. If the computed value of an exponent is exactly 3777 or 4000, a partial overflow condition exists. The error mode 2 flag is not set by a partial overflow. However, any further computation in floating point functional units with this exponent will set an error mode 2 flag. A complete overflow occurs when a floating point functional unit computes a result that requires an exponent larger than 3777 or 4000.

‡ { In this case the result is given a 3777 or 4000 exponent and a zero coefficient. The sign of the coefficient remains the same, as if the result had not exceeded the floating point range. Any further computation in floating point functional units with this result sets an error mode 2 flag.

§ { In this case, the result is given a 3777 or 4000 exponent and a zero coefficient. The sign of the coefficient remains the same, as if the result had not exceeded the floating point range. The coefficient calculation is ignored, and the overflow condition flag is set in the Program Status Designator (PSD) register. When the overflow condition occurs, the overflow flag in the PSD register causes an overflow condition message to be printed and the program to abort. Alternative actions (SCOPE 2.1 Reference Manual) can be specified by the user.

## UNDERFLOW ( +0 or -0 )

Underflow of the floating point range is indicated by an exponent of 0000 for positive numbers and 7777 for negative numbers, the smallest exponent values that can be represented in floating point format. If these exponent values happen to be the exact representation of a result, a partial underflow condition exists; and the underflow condition flag is not set. However, further computation in floating point functional units with these exponents may set the underflow condition flag.

A complete underflow occurs when a floating point functional unit computes a result that requires an exponent smaller than 0000 or 7777. In this case the result is given a 0000 or 7777 exponent and zero coefficient. The sign of the coefficient will be the same as that generated if the result had not fallen short of the floating point range. Thus, the complete underflow indicator is a word of all zero bits, or all one bits, depending on the sign. It is the same as a zero word in integer format.

‡ No underflow indicator is set and no error message is printed.

§ { A complete underflow occurs for this instruction whenever the exponent computation results in less than -1776 octal. This situation is sensed as a special case, and a complete zero word with proper sign results; the coefficient calculation is ignored, and the underflow condition flag is set in the PSD register. When the underflow condition occurs, the underflow flag in the PSD register causes an underflow condition message to be printed and the program to abort. Alternative actions (SCOPE 2.1 Reference Manual) can be specified by the user.

---

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

## INDEFINITE RESULT

An indefinite result indicator is generated by a floating point functional unit when a calculation cannot be resolved; such as a division operation where the divisor and the dividend are both zero. Another case is multiplication of an overflow number times zero. An indefinite result is a value which cannot occur in normal floating point calculations. An indefinite result is represented by a minus zero exponent and a zero coefficient (17770 --- 0).

‡ Any floating point functional unit receiving an indefinite indicator as an operand will generate an indefinite result regardless of the other operand value, and sets an error mode 4 flag.

§ When the indefinite result is generated, a flag is set in the PSD register, an indefinite condition message is printed, and the program aborts. Alternative actions (SCOPE 2.1 Reference Manual) can be specified by the user.

FLOATING POINT REPRESENTATION TABLE

Positive Coefficient			Negative Coefficient		
OVERFLOW	Complete Overflow	= 3777 0----- 0	Complete Overflow	= 4000 7----- 7	
	Partial Overflow	= 3777 X----- X	Partial Overflow	= 4000 X----- X	
INTEGERS	Largest: $7----- 7. \times 2^{+1776}$	= 3776 7----- 7	*Largest: $-7----- 7. \times 2^{-1776}$	= 4001 0----- 0	
	Smallest: $1. \times 2^0$	= 2000 0--- 01	*Smallest: $-1. \times 2^0$	= 5777 7--- 76	
*** ZERO	Positive Zero	= 2000 0----- 0	Negative Zero	= 5777 7----- 7	
INDEFINITE OPERANDS	Indefinite Operand	= 1777 0----- 0	**Indefinite Operand	= 6000 7----- 7	
FRACTIONS	Largest: $7----- 7. \times 2^{-60}$	= 1717 7----- 7	*Largest: $-7----- 7. \times 2^{-60}$	= 6060 0----- 0	
	Smallest: $1. \times 2^{-1777}$	= 0000 0--- 01	*Smallest: $-1. \times 2^{-1777}$	= 7777 7--- 76	
UNDERFLOW	Complete Underflow	= 0000 0----- 0	Complete Underflow	= 7777 7----- 7	
	Partial Underflow	= 0000 X----- X	Partial Underflow	= 7777 X----- X	
<p>* In absolute value.  ** An indefinite operand with a negative sign can occur only from packing or Boolean operations.  *** FORTRAN represents positive zero by 0000 0-- -0 and negative zero by 7777 7-- -7.</p>					

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

## NONSTANDARD FLOATING POINT ARITHMETIC

Non-standard floating point representation:

0000X----- X is treated by multiply and divide as positive zero (+0)  
 7777X----- X is treated by multiply and divide as negative zero (-0)  
 3777X-----X is treated as positive infinity ( $+\infty$ )  
 4000X-----X is treated as negative infinity ( $-\infty$ )  
 1777X-----X is treated as positive indefinite (+IND)  
 6000X-----X is treated as negative indefinite (-IND)

where X is an unspecified octal digit.

If the correct result of an operation coincides with any of the above exponents, no error flag is set.

When a floating point arithmetic unit uses one of these six special forms as an operand, however, only the following octal words can occur as results and the associated error mode flag is set.

37770-----0	positive infinity ( $+\infty$ )	Overflow condition flag
40007-----7	negative infinity ( $-\infty$ )	Overflow condition flag
17770-----0	positive indefinite (+IND)	Indefinite condition flag
00000-----0	positive zero (+0)	Underflow condition flag

The following tabulations show results of the add, subtract, multiply and divide operations using various combinations of infinite, indefinite, and zero quantities as operands. The designations w and n are defined as follows:

w = any word except  $\pm\infty$ , IND  
 n = any word except  $\pm\infty$ , IND, or  $\pm 0$

**ADD**  
 $X1 = X2 + X3$

		X3			
		W	$+\infty$	$-\infty$	$\pm$ IND
X2	W	-	$+\infty$	$-\infty$	IND
	$+\infty$	$+\infty$	$+\infty$	IND	IND
	$-\infty$	$-\infty$	IND	$-\infty$	IND
	$\pm$ IND	IND	IND	IND	IND

**SUBTRACT**  
 $X1 = X2 - X3$

X3

		W	$+\infty$	$-\infty$	$\pm\text{IND}$
X2	W	-	$-\infty$	$+\infty$	IND
	$+\infty$	$+\infty$	IND	$+\infty$	IND
	$-\infty$	$-\infty$	$-\infty$	IND	IND
	$\pm\text{IND}$	IND	IND	IND	IND

**MULTIPLY**  
 $X1 = X2 * X3$

X3

		+N	-N	+0	-0	$+\infty$	$-\infty$	$\pm\text{IND}$
X2	+N	-	-	0	0	$+\infty$	$-\infty$	IND
	-N	-	-	0	0	$-\infty$	$+\infty$	IND
	+0	0	0	0	0	IND	IND	IND
	-0	0	0	0	0	IND	IND	IND
	$+\infty$	$+\infty$	$-\infty$	IND	IND	$+\infty$	$-\infty$	IND
	$-\infty$	$-\infty$	$+\infty$	IND	IND	$-\infty$	$+\infty$	IND
	$\pm\text{IND}$	IND	IND	IND	IND	IND	IND	IND

**DIVIDE**  
 $X1 = X2 / X3$

X3

		+N	-N	+0	-0	$+\infty$	$-\infty$	$\pm$ IND
X2	+N	-	-	$+\infty$	$-\infty$	0	0	IND
	-N	-	-	$-\infty$	$+\infty$	0	0	IND
	+0	0	0	IND	IND	0	0	IND
	-0	0	0	IND	IND	0	0	IND
	$+\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	IND	IND	IND
	$-\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	IND	IND	IND
	$\pm$ IND	IND	IND	IND	IND	IND	IND	IND

## INTEGER ARITHMETIC

Central processor has no 60-bit integer multiply or divide instructions. Integer multiplication and division are performed with 48-bit arguments. The exponent of the result is set to zero. 48-bit integer multiplication is performed with an integer multiply instruction, but integer division must be performed in the floating divide unit. Integer arithmetic is accomplished by putting the integers into unnormalized floating point format using the pack instruction with a zero exponent value.

In integer division, the exponent of the resulting quotient is removed and the result is shifted to compensate for the fact that the result was normalized. In FORTRAN Extended, integer results of multiplication or division are expressed within 48 bits. Full 60-bit one's complement integer sums and differences are possible internally as the central processor has integer addition and subtraction instructions. However, because the binary-to-decimal conversion routines use multiplication and division, the range of integer values output is limited to those which can be expressed with 48 bits.

## DOUBLE PRECISION

Although complete arithmetic instructions using double precision arguments are not provided by the hardware, the FORTRAN compiler generates code for true double precision by using instructions which give upper and lower half results with single precision arguments.

## COMPLEX

Complex arithmetic instructions are not provided by hardware. The FORTRAN compiler generates code for complex arithmetic by using single precision floating point instructions.

## LOGICAL AND MASKING

Logical and masking operations are provided by hardware logical instructions which operate on the entire 60-bit word (refer to section 2, part I). Positive values are considered false; negative values are true. The constant TRUE. generates -1; the constant FALSE. generates zero.

## ARITHMETIC ERRORS

Arithmetic errors are classified at execution time as mode 1 - 7:

Mode	Error
1	Address out of range
	§ { Reference to LCM or SCM outside established limits. LCM or SCM block range
2	Operand is an infinite number
3	Address out of range or operand is infinite number
4	Indefinite operand
5	Address out of range or indefinite operand
6	Operand is infinite or indefinite number
7	Operand is infinite, indefinite or address is out of range

---

§Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

Mode 1 Address out of range. A non-existent storage location has been referenced. Mode 1 errors may be caused by:

- calling a non-existent subprogram during execution
- using an incorrect number of arguments when calling a subprogram
- a subscript assuming an illegal value
- no dimensions specified for an array name

Mode 2 Infinite operand. One of the operands in a real operation is infinite. Infinity is the result whenever the true result of a real operation would be too large for the computer, or when division by zero is attempted. A value of infinity may be returned when some functions are referenced. For example, EXP(999.) would be infinity.

In the following example, Z would be given the value infinity. When the addition  $Z + 56$ . is attempted execution terminates with a mode 2 error.

```

1 FORMAT (F12.3)
Y = 0.
Z = 23.2/Y
PRINT 1, Z
CAT = Z + 56.

```

When the print statement is executed, an R is printed to indicate an out of range value.

Mode 3 Address is out of range or operand is infinite number.

Mode 4 Indefinite operand. One of the operands in a real operation is indefinite. An indefinite result is produced by dividing 0. by 0. or multiplying an infinite operand by 0. An illegal library function reference may return an indefinite value. For example, SQRT (-2.) would produce an indefinite result. An attempt to print an indefinite value produces the letter I.

Mode 5 Address is out of range or indefinite operand.

Mode 6 Operand is infinite or indefinite. A mode 6 arithmetic error occurs when a real operation is performed with one operand infinite and the other operand indefinite.

Mode 7 Operand is infinite, indefinite, or address is out of range.

‡ When an arithmetic error occurs the following type of message appears in the dayfile and execution is terminated:

```

14.39.06.ERROR MODE = 2. ADDRESS = 002135

```

---

‡Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

When an arithmetic error occurs, the following type of message appears in the dayfile under the headings shown below:

14.30.36\*00012.059\*SYS.                    SC006 -                    SCM DIRECT RANGE

CODE xxnnn

xx SC or JM                    SC indicates System Control; JM, Job Management. System Control provides system overlay loaders and some communication between operating system overlays. Job Management controls user program input/output, and prepares user programs for execution.

nnn                    Index number of the message.

MESSAGE AND MEANING                    The message and an interpretation (if necessary) are printed.

LEVEL                    Indicates the level of severity of the error as follows:

- X                    Job terminates. No EXIT processing occurs.
- F                    Job terminates. EXIT processing occurs.
- W                    Warning is printed, and error is ignored. Processing continues, although the portion of the program containing the error may not be executed.
- I                    Informative message is printed.

CODE	MESSAGE AND MEANING	LEVEL
SC001	LCM PARITY	F
SC002	SCM PARITY	F
SC003	LCM BLOCK RANGE	F
SC004	SCM BLOCK RANGE	F
SC005	LCM DIRECT RANGE	F
SC006	SCM DIRECT RANGE	F
SC007	PROGRAM RANGE	F
SC008	BREAKPOINT	F
SC009	STEP CONDITION	F
SC010	INDEFINITE CONDITION	F
SC011	OVERFLOW CONDITION	F
SC012	UNDERFLOW CONDITION	F
SC040	JOB MAKING 6000 REQUEST IN RAS+1; RAS+1 of user area is non-zero.	F

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.



The following example outlines a method for detecting the location of an arithmetic error. When the following program is executed:

```

PROGRAM ERR (OUTPUT,TAPE1=OUTPUT)
  NAMELIST /OUT/T,E
  DATA T,E/0.,1./
  1 WRITE (1,OUT)
  5 E = E/T + 1.
    T = T - 1.
    GO TO 1
  END

```

this message appears in the dayfile:

**15.12.32.ERROR MODE = 2. ADDRESS =002154**

2154 is one plus the address at which the error was detected. The error was detected at address 2153. To locate this address in the program, turn to the Load Map and read the entries under PROGRAM AND BLOCK ASSIGNMENTS.

BLOCK	ADDRESS	LENGTH	FILE
ERR	101	2071	LGO
/Q8.IO./	2172	134	
FORSYS=	2326	643	SL-FORTRAN
GETFIT=	3171	34	SL-FORTRAN
/IO.BUF./	3225	227	
NAMOUT=	3454	600	SL-FORTRAN
SYSID=	4254	1	SL-FORTRAN
/JMPS.RM/	4255	11	
LBUF.SQ	4266	133	SL-SYSIO
/CON.RM/	4421	6	

The user program ERR occupies storage locations 101 through 2171. Location 2153 lies between 101 and 2171 and is therefore in the main program ERR. It is location 2052 relative to the beginning of ERR (all locations are relative to the first word address of the program load);  $2153 - 101 = 2052$  (octal).

‡Applies only to CONTROL DATA CYBER 70/Models 72, 73, 74, CYBER 170, and 6000 Series computers.

## USER EXCHANGE PACKAGE

P	00	000136	A0	045000	AP	000000	SC(A0)=										SC(P)=	5475	0040	0000	1314	6000		
RAS	00	017320	A1	001342	B1	000321	SC(A1)=	0000	0000	0000	0004	5000					SC(B1)=	3353	0516	0400	0000	0000		
FLS	00	045000	A2	002014	B2	000003	SC(A2)=	1717	0631	4631	4631	4632					SC(B2)=	2401	2005	3400	0000	0100		
PSD	00	060003	A3	000153	B3	000000	SC(A3)=	1720	4000	0000	0000	0000					SC(B3)=	0000	0000	0000	0000	0000		
RAL	00	000000	A4	000146	B4	000027	SC(A4)=	6057	3777	7777	7777	7777					SC(B4)=	0000	0000	0000	0000	0000		
FLL	00	000000	A5	000147	B5	000006	SC(A5)=	1721	5335	6735	6735	6735					SC(B5)=	0317	1520	2300	0000	0277		
NEA	40	016140	A6	000146	B6	000001	SC(A6)=	6057	3777	7777	7777	7777					SC(B6)=	0000	0000	0000	0000	0000		
EEA	00	010460	A7	000321	B7	000001	SC(A7)=	3353	0516	0400	0000	0000					SC(B7)=	0000	0000	0000	0000	0000		
X0		3777	0000	0000	0000	0000	SC(X0)=	0000	0000	0000	0000	0000					LC(X0)=							
X1		0000	0000	0000	0004	5000	SC(X1)=											LC(X1)=						
X2		3777	0000	0000	0000	0000	SC(X2)=	0000	0000	0000	0000	0000						LC(X2)=						
X3		1720	4000	0000	0000	0000	SC(X3)=	0000	0000	0000	0000	0000						LC(X3)=						
X4		0000	0000	0000	0000	0000	SC(X4)=	0000	0000	0000	0000	0000						LC(X4)=						
X5		1721	5335	6735	6735	6735	SC(X5)=												LC(X5)=					
X6		6057	3777	7777	7777	7777	SC(X6)=	0000	0000	0000	0000	0000							LC(X6)=					
X7		3777	0000	0000	0000	0000	SC(X7)=	0000	0000	0000	0000	0000							LC(X7)=					

---

All input and output between a file referenced in a FORTRAN Extended program and the file storage device is under control of a Record Manager. When running on SCOPE 2, 7000 Record Manager is used; when running on other systems, the CYBER Record Manager is used. These Record Managers normally appear the same to FORTRAN users; however, they do offer substantially different capabilities. Standard file organizations and record formats are defined to facilitate file interchange and access through different products.

Record Manager can be called directly, as described in III-6, to use the extended file structure and processing available. This section deals only with Record Manager processing that results from standard language use.

File processing is governed by values compiled into the file information table (FIT) for each file.

**If a file or its FIT is changed by other than standard FORTRAN I/O statements, subsequent FORTRAN I/O to that file may not function correctly. Thus, it is recommended that the user not try to use both standard FORTRAN and non-standard I/O on the same file within a program.**

### FILE AND RECORD DEFINITIONS

A file is a collection of records referenced by its logical file name. It begins at beginning-of-information and ends with end-of-information.

A record is data created or processed by:

One unformatted READ or WRITE.

One card image or a print line defined within a formatted, list directed, or NAMELIST READ or WRITE.

One READMS or WRITMS.

One BUFFER IN or BUFFER OUT.

On storage, a file may have records in one of 8 formats defined to Record Manager. Only 4 of these are part of standard processing:

Z Record is terminated by a 12-bit zero byte in the low order byte position of a 60-bit word.

W Record length is contained in a control word prefixed to the record by Record Manager.

U Record length is defined by the user.

S SCOPE logical record.

The remaining types can be formatted within a program under user control and written to a device using a WRITE statement if the FILE card is used to specify another record type. Similarly, these types can be read by a READ.

The user is responsible for supplying record length information appropriate to each type before a write and for determining record end for a read. For example, a D type record requires a field within the record to specify record length.

Unformatted READ and WRITE are implemented through the GETP and PUTP macros of Record Manager; consequently, record operations must conform to macro restrictions. Specifically, RT=R cannot be performed for unformatted operations.

## STRUCTURE OF INPUT/OUTPUT FILES

FORTTRAN Extended sets certain values in the file information table depending on the nature of the input/output operation and its associated file structure. Table III-5-1 lists these values for their respective FIT fields; all except those marked with an asterisk (\*) can be overridden at execution-time by a FILE card. (Numbers in parentheses and asterisks refer to notes listed following the table.)

## SEQUENTIAL FILES

‡ { With READ and WRITE statements, the record type (RT) depends on whether the access is formatted or unformatted. A formatted WRITE produces RT=Z records, with each record terminated by a system-supplied zero byte in the low order bits of the last word in the record. An unformatted WRITE produces RT=W records, in which each record is prefixed by a system-supplied control word. Blocking is type C or I for formatted and unformatted records, respectively.

§ { With READ and WRITE statements, the record type is W for all file types; blocking is I for unformatted tape files, and unblocked for all other files.

PRINT and PUNCH statements produce Z‡ type records with C type blocks or W§ type records unblocked for processing on unit record equipment.

BUFFER IN and BUFFER OUT assume S‡-type or W§-type records. Formatting is determined by the parity designator in each BUFFER statement. An unformatted operation does not convert character codes (CM=NO), while a formatted operation does.

---

‡Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§Applies only to SCOPE 2.1.

TABLE III-5-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED

FIT Fields		Formatted, NAMELIST, & List-Directed READ/WRITE	Unformatted READ/WRITE	BUFFER IN/ BUFFER OUT	READMS/WRTMS
Meaning	Mnemonic				
CIO buffer size (words)	(1) BFS‡	2002B	2002B	2002B	2002B
Block type	BT	C‡	I	C‡	n/a
Close flag (positioning of file at CLOSEM time)	CF	N	N	N	N‡/R§ *
Length in characters of record trailer count field (T type records only)	CL	0	0	0	n/a
Conversion mode	CM	YES	NO	(2)	n/a
Beginning character position of trailer count field, numbered from zero (T type records only)	CP	0	0	0	n/a
Length field (D type records ) or trailer count field (T type records) is binary	C1	NO	NO	NO	n/a
Error options	EO	AD	AD	AD	AD
Trivial error limit	ERL	0	0	0	0
Extended diagnostic flag	EXD	NO	NO	NO	NO
Length in character of an F or Z type record	FL	150 (5)	0	0	0
File organization	FO	SQ *	SQ *	SQ *	WA *
Character length of fixed header for T type records	HL	0	0	0	n/a
Length of user's label area (number of characters)	(7) LBL	0 *	0 *	0 *	n/a

‡Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§Applies only to SCOPE 2.1.

TABLE III-5-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED (continued)

FIT Fields		Formatted, NAMELIST, & List-Directed READ/WRITE	Unformatted READ/WRITE	BUFFER IN/ BUFFER OUT	READMS/WRITMS
Meaning	Mnemonic				
Logical file name	LFN	(3)	(3)	(3)	(3)
Length in characters of record length field (D type records)	LL	0	0	0	n/a
Beginning character position of record length, numbered from zero (D type records)	LP	0	0	0	n/a
Label type	(7) LT	U	U	U	n/a
Maximum block length (MBL is set as the number of characters but is converted and maintained as the number of words)	MBL	(4)	(4)	(4)	n/a
Minimum block length in characters	MNB	0	0	0	n/a
Minimum character length of R type records	MNR	0	0	0	n/a
Maximum record length in characters	(5) MRL	150	$2^{23}-1$	(8) *	n/a
Multiple of characters per K, E type block	MUL	2	2	2	n/a
Open flag (positioning of file at OPENM time)	(7) OF	N	N	N	$N^{\ddagger}/R^{\S}$ *
Padding character for sequential file blocks	PC	76B	76B	76B	n/a
Processing direction	PD	IO	IO	IO	IO
Number of records per K type block	RB	1	1	1	n/a

$\ddagger$  Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

$\S$  Applies only to SCOPE 2.1.

TABLE III-5-1. DEFAULTS FOR FIT FIELDS UNDER FORTRAN EXTENDED (continued)

FIT Fields		Formatted, NAMELIST, & List-Directed READ/WRITE	Unformatted READ/WRITE	BUFFER IN/ BUFFER OUT	READMS/WRTIMS
Meaning	Mnemonic				
Record mark character	RMK	62B	n/a	62B	n/a
Record type	RT	Z <sup>‡</sup> /W <sup>§</sup>	W (6)	S <sup>‡</sup> /W <sup>§</sup>	U
Length field (D type records) or trailer count field (T type records) has sign overpunch	SB	NO	NO	NO	n/a
Error message disposition	SDS	YES	YES	YES	YES
Suppress read ahead	SPR	NO	NO	NO	n/a
Character length of trailer portion of T type records	TL	0	0	0	n/a
User label processing	(7) ULP	NO	NO	NO	NO
End of volume flag (positioning of file at volume CLOSEM time)	VF	U	U	U	U

Notes:

- n/a FIT field not applicable to this I/O mode.
- \* Default cannot be overridden by a FILE card.
- (1) Default can be changed on PROGRAM statement; if the FILE card resets BFS, a different buffer will be used.
- (2) Set by parity designator in BUFFER IN or BUFFER OUT statement.
- (3) Set by PROGRAM statement or EXECUTE control card.
- (4) Set by Record Manager.
- (5) Default can be changed on PROGRAM statement.
- (6) Default can be overridden by a FILE card only if RT≠R and RT≠Z.
- (7) Use of the LABEL subroutine sets LBL=80, LT=ST, OF=R, and ULP=F.
- (8) Maximum record length equal to length of record specified in BUFFER IN or BUFFER OUT statement.

<sup>‡</sup>Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

<sup>§</sup>Applies only to SCOPE 2.1.

The ENDFILE statement writes a boundary condition known as an end-of-partition. When this boundary is encountered during a read, the EOF function returns end-of-file status. An end-of-partition may not necessarily coincide with end-of-information, however, and reading can continue on the same file until end-of-information on the file has been encountered.

End-of-information is written as the file is closed during program termination. A third boundary for sequential files, a section, is not recognized during reading except for the special case of the file INPUT.

## RANDOM FILES

Files created by the random mass storage routines OPENMS, WRITMS, STINDEX, and CLOSMS (described in section III-7) create an indexed word addressable file. The master index, which is the last record in the file, is created and maintained by FORTRAN routines rather than Record Manager routines.

One WRITMS call creates one U† format record; one READMS call reads one U format record. If the length specified for a READMS is longer than the actual record, the excess locations in the user area are not changed by the read. If the record is longer than the length specified for a READMS, the excess words in the record are skipped.

## FILE CONTROL CARD

The FILE card provides a means to override FIT field values compiled into a program and consequently a means to change processing normally supplied for standard input/output. In particular, the FILE card can be used to read or create a file with a structure that does not conform to the assumptions of default processing.

A FILE card also can be used to supplement standard processing. For example, setting EXD=YES produces a full error message for Record Manager errors, instead of an error number only.

At execution time, FILE card values are placed in the FIT when the referenced file is opened. FILE card values have no effect if the execution routines do not use the fields referenced. Furthermore, FORTRAN routines may, in some cases, reset FIT fields after the FILE card is processed. These fields are noted in Table III-5-1.

Format of the FILE card is:

FILE(lfn,field=value, . . . )

lfn	File name as it appears on the EXECUTE or LGO control card; if file name does not appear there, then lfn is file name as it appears in the PROGRAM statement.
field	FIT field mnemonic
value	Symbolic or integer value

---

†Record type W was written through Version 4.2. Existing files with RT=W are recognized and processed correctly under subsequent versions of FORTRAN Extended without user action.



‡An LDSET loader control card must appear in the load set of a program using a FILE card. An acceptable format is:

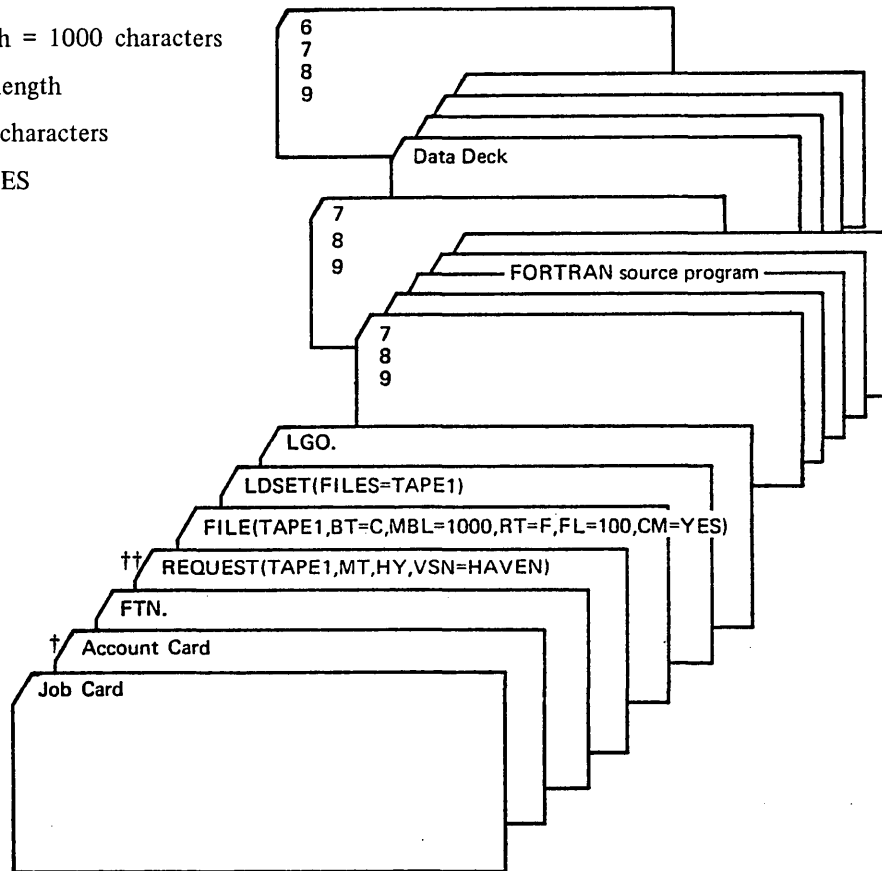
LDSET(FILE=lfm)

lfm            File name appearing on FILE card.

The FILE card itself may appear anywhere in the control cards prior to program execution, but it must not interrupt a load set.

This deck illustrates the use of the FILE card to override default values supplied by the FORTRAN compiler. Assuming the source program is using formatted WRITES and 100-character records are always written, the file is written on magnetic tape in 1000-character blocks (except possibly the last block) with even parity, at 800 bpi. No labels are recorded, and no information is written except that supplied by the user. Records are blocked 10 to a block. The following values are used:

- Block type = character count
- Maximum block length = 1000 characters
- Record type = fixed length
- Record length = 100 characters
- Conversion mode = YES



‡Does not apply to SCOPE 2.1.  
†If required by the operating system.  
‡‡Format applicable to SCOPE 3.4 only.

## SEQUENTIAL FILE OPERATIONS

### BACKSPACE/REWIND

Backspacing on FORTRAN files repositions them so that the last logical record becomes the next logical record.

§ BACKSPACE is permitted only for files with F, S, or W record format or tape files with one record per block.

The user should remember that formatted input/output operations can read/write more than one record; unformatted input/output and BUFFER IN/OUT read/write only one record.

The rewind operation positions a magnetic tape file so that the next FORTRAN input/output operation references the first record. A mass storage file is positioned to the beginning of information.

The following table details the actions performed prior to positioning.

Condition	Device Type	Action
‡ Last operation was WRITE or BUFFER OUT	Mass Storage	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written.
	Unlabeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written. Two file marks are written.
	Labeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted record is written. A file mark is written. A single EOF label is written. Two file marks are written.

§ Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

‡ Applies only to SCOPE 2.1.

Condition	Device Type	Action
Last operation was WRITE. BUFFER OUT or ENDFILE	Mass storage (no blocking)	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written.
	Unlabeled Magnetic Tape or Blocked Mass Storage	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written. Two file marks are written (on tape).
	Labeled Magnetic Tape or Labeled Blocked Mass Storage	Any unwritten blocks for the file are written. If record format is S, a zero length level 17 block is written. A file mark is written. A single EOF label is written. Two file marks are written.
Last operation was READ, BUFFER IN or BACKSPACE	Mass Storage	None
	Unlabeled Magnetic Tape	None
	Labeled Magnetic Tape	If the end of information has been reached, labels are processed.
No previous operation	Magnetic Tape	If the file is assigned to on-line magnetic tape, a REWIND request is executed. § If the file is staged, the REWIND request has no effect. The file is staged and rewound when it is first referenced.
	Mass Storage	REWIND request causes the file to be rewound when first referenced.
Previous operation was REWIND		Current REWIND is ignored.

§ Applies only to SCOPE 2.1.

## ENDFILE

The ENDFILE operation introduces a delimiter into an input/output file. ENDFILE writes an end-of-partition for W record types. ENDFILE terminates the current block for a magnetic tape file and writes a level 17 zero length block for record types W, D, R, T, F, and U, and record type Z with and without C blocking.

A WRITE/BUFFER OUT can follow an ENDFILE operation. If the file has records of the format W, S, or Z with C blocking or it is a mass storage file with any other block/record formats, no special action is performed. However, if the file is assigned to magnetic tape and has a record format other than W, S, or Z with C blocking, a tape mark is written preceding the requested record.

Meaningful results are not guaranteed if an ENDFILE is written on a random access file, and subsequently a random file subroutine, such as READMS, is called.

## BUFFER INPUT/OUTPUT

‡ The maximum lengths for physical records on tape can be exceeded using the BUFFER input/output statement if the L parameter on the REQUEST control card is specified.

§ { BUFFER IN/OUT statements can be used to achieve some degree of overlap between the user program and input/output with an external device (mass storage or tape); however the memory area specified in the BUFFER IN/OUT statement will not be used as the physical record buffer. These buffers are maintained within an operating system buffer area in LCM. The execution of a BUFFER IN/OUT statement, therefore, involves movement of a record between system buffers in LCM and the memory area specified in the BUFFER IN/OUT statement. Correspondence between individual BUFFER statements and physical records on a device depends upon the block specification. For example, K blocking with a record count of one ensures that each BUFFER IN/OUT corresponds to a block.

## BUFFER IN

1. Only one record is read each time a BUFFER IN is performed. If the length specified by the BUFFER statement is longer than the record read, excess locations are not changed by the read. If the record read is longer than the length specified by the BUFFER statement, the excess words in the record are ignored. The number of central memory words transferred to the program block can be obtained by referencing the function LENGTH or the subroutine LENGTHX (section 8, part I).
2. When records do not terminate on a word boundary (such as might occur on a file not created by BUFFER statements), and if the number of words requested in a BUFFER IN is greater than or equal to the number of words in the record, the exact length of the record can be determined by using the LENGTHX library subroutine. LENGTHX returns the number of unused bits in the last word of the data transfer as well as the number of central memory words transferred (section 8, part I).

---

‡ Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§ Applies only to SCOPE 2.1.

3. After using a BUFFER IN/OUT statement on unit *u*, and prior to referencing unit *u* or the contents of storage locations *a* through *b*, the status of the BUFFER operation must be checked by a reference to the UNIT function (section 8, part I). This status check ensures that the data has actually been transferred, and the buffer parameters for the file have been restored.
4. If an attempt is made to BUFFER IN past an end-of-file without testing for the condition by referencing the UNIT function, the program terminates with the diagnostic: END OF FILE ENCOUNTERED file name
5. If the last operation on the file was a write operation, no data is available to read. If a read is attempted, the program terminates with the diagnostic: WRITE FOLLOWED BY READ ON FILE
6. If the starting address for the block is greater than the terminal address, the program terminates with the diagnostic: BUFFER DESIGNATION BAD FWA.GT.WA, file name
7. If an attempt is made to BUFFER IN from an undefined file (a file not declared on the PROGRAM card), the program terminates with the diagnostic: UNASSIGNED MEDIUM, file name

## BUFFER OUT

1. One record is written each time a BUFFER OUT is performed. The length of the record is the terminal address of the record (LWA) – starting address (FWA) + 1.
2. As with BUFFER IN, a BUFFER OUT operation must be followed by a reference to the UNIT function. This reference must occur prior to any other reference to the file.
3. If the terminal address is less than the first word address, the program terminates and the following diagnostic is issued:  

BUFFER SPECIFICATION BAD FWA.GT.LWA, file name
4. The UNASSIGNED MEDIUM diagnostic is similar to that issued from a BUFFER IN.

## LABELED FILE PROCESSING

FORTRAN Extended has the capability of input/output processing of labeled files on magnetic tape. The following subroutine is provided to pass label information to the operating system.†

```
CALL LABEL(u,labinfo)
```

*u*            Logical unit number.

*labinfo*     Name of 4-word array containing label information in the format given for words 9–12 of the file environment table (FET) in the operating system reference manual.

The control card that requests the tape for the job must have specified that the tape has labels before the CALL LABEL statement can be used.

---

†The CALL LABEL subroutine has no effect under SCOPE 2.1.

On input, the specified file's label is compared with the indicated information in labinfo (unless it was so checked when an earlier LABEL control card was executed). If any of the relevant fields were filled with binary zeros by CALL LABEL, these fields are set to the values contained in the label read. If there is a mismatch between the label read and any field not zero-filled, a request is sent to the operator for a GO or DROP response.

On output, the appropriate information from labinfo is written as a label at the beginning of the specified file. If any of the relevant fields are filled with binary zeros, the corresponding label field will be set to an appropriate default value.

CALL LABEL should not be used with files accessed with direct Record Manager input/output routines.

## PROGRAMMING NOTES

Meaningful results are not guaranteed in the following circumstances:

1. Mixed formatted and unformatted read/write statements on the same file (without an intervening REWIND).
2. Mixed buffer input/output statements and read/write statements on the same file.
3. Requesting a LENGTH function of LENGTHX call on a buffer unit before requesting a UNIT function.
4. Two consecutive buffer input/output statements on the same file without the intervening execution of a UNIT function call.
5. Violating any of the restrictions specified in Table III-5-1.
6. Failing to close a mass storage input/output file with an explicit CLOSMS in an overlay program.
7. Writing formatted records on a seven-track S or L tape without specifying CM=NO on a FILE card.
8. Using the first variable of an input/output list after encountering end-of-file in a formatted read.
9. Issuing an ENDFILE as the first operation on a file.

The FORTRAN user can access Record Manager<sup>‡</sup> facilities by calling external subprograms that use the COMPASS Record Manager macros. The subprograms described here allow limited access to the Record Manager macros without requiring the user to write his own subprograms in COMPASS. Subprograms are provided to create, access, position, and process the files and to modify the file information tables. The Record Manager Reference Manual includes a complete description of each macro and its parameters.

## FILE INFORMATION TABLE CALLS

To place values in the file information table the user can call one of the following subroutines:

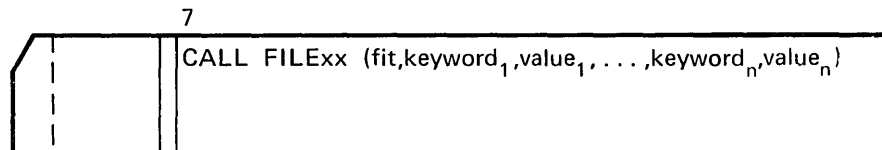
FILESQ for sequential files

FILEWA for word addressable files

FILEIS for indexed sequential files

FILEDA for direct access files

FILEAK for actual key files



All parameters, with the exception of fit, are paired; the first parameter is the keyword which indicates the field in the file information table, the second parameter is the value to be placed in the field. Only the pertinent parameters need be specified, and they may appear in any order. Since a FORTRAN call can contain a maximum of 63 parameters, 31 file information table fields can be specified with a FILExx call.

xx            SQ, WA, IS, DA, or AK

fit            Name of an array. Record Manager resides in the user's field length, and the array must be large enough to contain both the file information table (FIT) and the file environment table (FET). 35 words should be allocated; 20 words for the file information table and 15 words for the file environment table. The FIT is created by the subroutine FILExx, beginning in the first word of the array. Record Manager supplies the information which is placed in the user's array after the FIT.

---

<sup>†</sup>Record Manager implies CYBER Record Manager throughout this section.

<sup>‡</sup>This information applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

INFORMATION ON THIS PAGE DOES NOT APPLY TO SCOPE 2.1.

**Keyword** Specifies a file information table field. An FIT field mnemonic is passed as an L format Hollerith constant. FIT mnemonics are described in the Record Manager Reference Manual.

**Example:**

```
3LFWB
3LLFN
2LKL
```

**Value** Value to be placed in the FIT field specified by keyword. The following three types of values are allowed:

Names of arrays or external subroutines.

**Example:** Specify the array RCD as the user's record area.

```
... , 3LWSA,RCD, ...
```

Integer constants or integer variables.

**Example:** Set the key length field to ten characters.

```
... , 2LKL,10, ...
```

Symbolic option keywords. The value of some FIT fields must be supplied symbolically (see Record Manager Reference Manual). Symbolic option keywords are passed as L format Hollerith constants.

**Example:** Select the duplicate key processing option.

```
... , 3LKDI,3LYES, ...
```

To insure that the routines required for processing various record types are loaded, the subroutines FILEDA, FILEIS, and FILEAK force the loading of entry points GET.D, GET.T, and GET.R. Subroutine FILESQ forces the loading of entry points GET.W, GET.F, GET.S, GET.U, GET.D, GET.T, GET.R, GET.Z, and GET.SP. For the record types not needed in a particular run, loading of those routines can be suppressed with LDSET(OMIT= . . .).

#### ACCESSING FILE INFORMATION TABLE FIELDS

Contents of the FIT can be accessed by using the integer function IFETCH.

IFETCH (fit,keyword)

**fit** Names of the array containing file information table.

**keyword** Character name of the field.



If the keyword specifies a one-bit field, negative result is returned if the bit is on and can be sensed by a positive-negative check; otherwise it is returned as an integer value.

Example:

```
M=IFETCH(FILE,2LRL)
```

The record length is returned to the function IFETCH and replaces the value of M.

## FILE COMMANDS

After the file information table is created using CALL FILExx file accessing commands can be issued. The first command must be OPENM, and the last CLOSEM.

In file commands, the parameters are identified strictly by their position; thus, parameters can be omitted only from the right. In FORTRAN, unlike COMPASS macros, adjacent commas are illegal in a subroutine call. When parameters are omitted the current value of the corresponding FIT fields remain unchanged. If the same subroutine is called twice, each with a different number of parameters, the compiler issues an informative diagnostic.

In some file commands a parameter position can have two meanings, for example  $\left\{ \begin{matrix} ka \\ wa \end{matrix} \right\}$  in CALL PUT, the top parameter always applies to index sequential or direct access files, and the bottom to word addressable files.

In the following description of the file commands, fit is the name of the array containing the file information table.

Example:

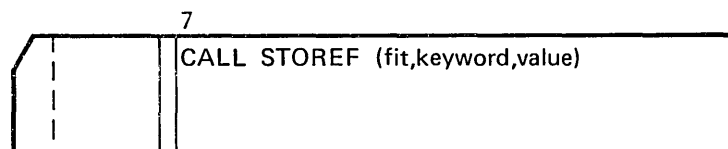
The following call sets up the FIT for a direct access file:

```
CALL FILEDA ( FILE, 3LLFN, 7LSDAFILE, 3LFWB, BUFFER, 3LBFS, 400, 3LBCK, 3LYES )
```

The FIT and the FET are to be constructed in the array named FILE. The file name (LFN) is SDAFILE. The buffer is to be placed in the 400-word array BUFFER. 3LBCK,3LYES selects the block checksumming option.

## UPDATING FILE INFORMATION TABLE

After the file information table is created, it can be updated by calls to the subroutine STOREF.

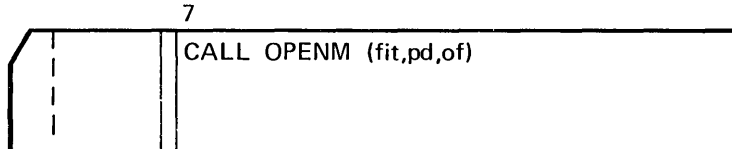


fit	Array where the file information table was created.
keyword	File information table field.
value	Value to be placed in the field.

Example:

```
CALL STOREF (FILE,2LRL,250)
```

Sets record length in the FIT, in the array FILE, to 250 characters.



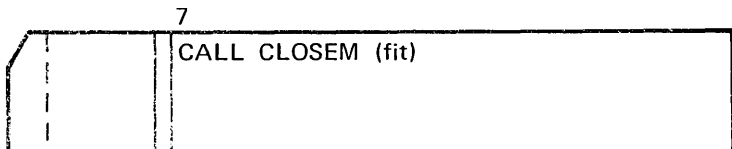
OPENM prepares a file for processing. Each file must be opened before processing.

pd Processing direction established when file is opened:

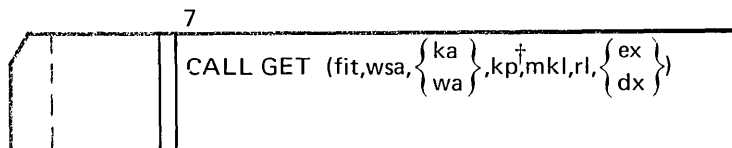
5LINPUT	Read only
6LOUTPUT	Write only
3LI-O	Read and write
3LNEW	Indexed sequential or direct access file to be created (write only)

of Open flag specifies position of file when it is opened:

1LR	Rewind; file is rewound before any other open procedures are performed.
1LN	No file positioning is done before other open procedures.
1LE	File is positioned immediately before end of information to allow extensions to a mass storage file.



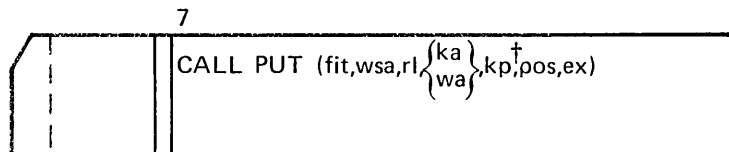
Terminates processing.



†kp is not applicable to AK files.

GET reads a record from an input/output device and delivers it to the user's record area.

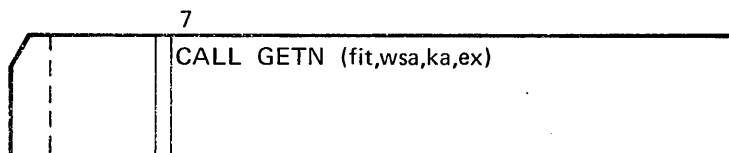
- wsa            Address of user's record area.
- ka            Address of user's key area for direct access or indexed sequential record to be read.
- wa            Word address on file where reading is to start.
- kp<sup>†</sup>        Beginning character position of key within ka. Key positions are ordered from left to right (0-9).
- mkl         Major key length on indexed sequential files.
- rl            Record length in characters.
- ex            Address of exit subroutine to be entered when an error occurs (word addressable, indexed sequential or direct access files). The value of ex must not be zero.
- dx            Address of end of the external subroutine to be entered at end of data for sequential files.



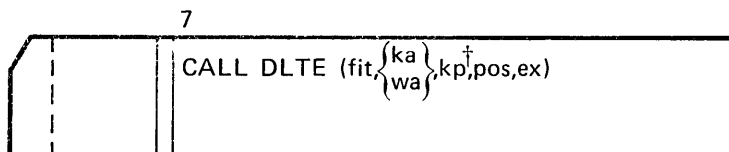
PUT places a record in a file.

- pos            For duplicate key processing, value may be 1LP to precede the current record or 1LN to make it the next record.

wsa,rl,ka,wa,kp,ex are the same as for GET.



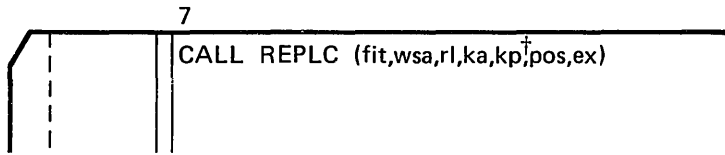
GETN accesses the next record on the file.



<sup>†</sup>kp is not applicable to AK files.

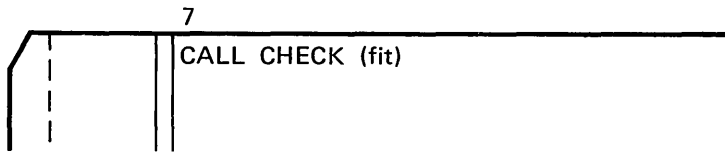
DLTE deletes a record from the file.

- ka                    Key address of record to be deleted.
- wa                    Word address of record to be deleted.
- pos                   Value may be 1LC to specify the current (last referenced) record to be deleted, or zero to delete the first record in a duplicate key chain.

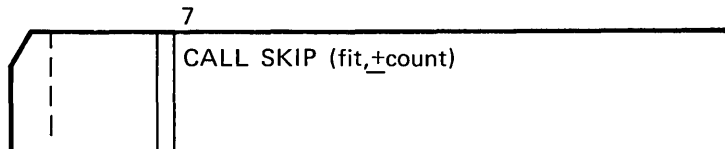


REPLC replaces an existing record with a record from the user's record area.

- pos                   Value may be 1LC to specify the current (last referenced) record to be replaced, or zero which will replace the first record in a duplicate key chain.

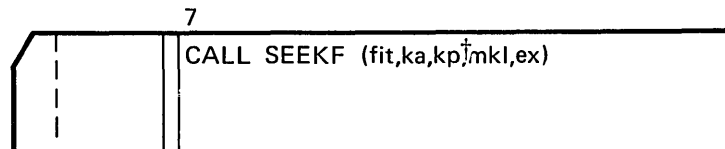


CHECK determines whether input/output operations on a file are complete and upon completion returns control.



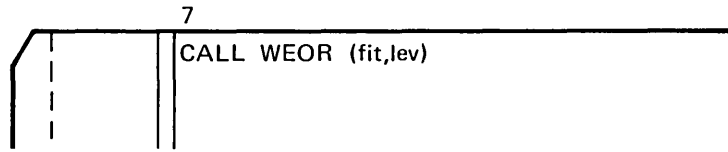
Repositions a file.

- count                Number of logical records to be skipped; positive for a forward skip, negative for a backward skip.



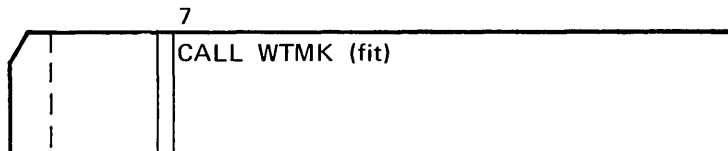
SEEKF allows central memory processing to overlap input/output operations.

†kp is not applicable to AK files.

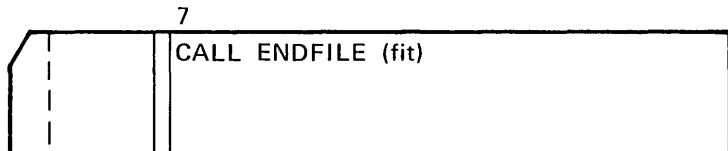


WEOR terminates a section, and an S type record.

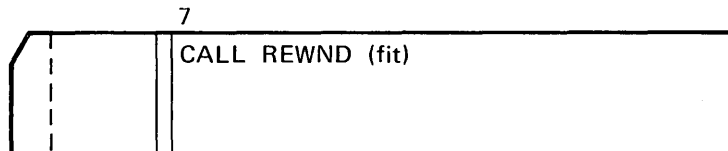
lev                    Level number (any value 0 to 16B) to be appended if record type is S; default is zero.



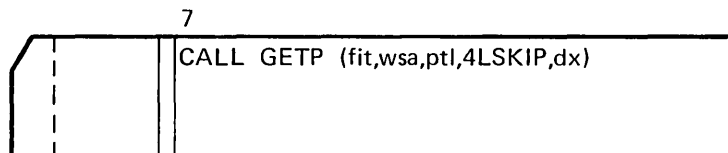
Writes a tape-mark.



Writes an end of partition.



REWND positions a tape file to the beginning of the current volume. It positions a mass storage file to the beginning of information.



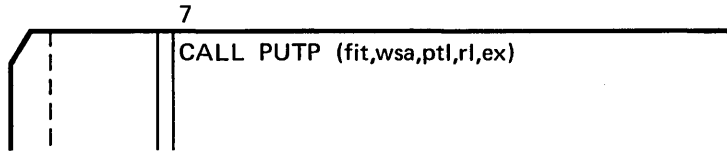
GETP retrieves partial records; it may be used to retrieve an arbitrary amount of data from a record.

wsa                    Name of user's record area to receive the record.

ptl                    Partial transfer length. Number of characters to be transferred.

skip                   Causes Record Manager to advance to next record before getting data if the value is 4LSKIP. Otherwise zero should be used.

dx                     Name of end-of-data routine.



Writes a portion of a record.

- wsa            Address of user's record area from which the record portion will be taken.
- ptl            Partial transfer length specifies the number of characters to be transferred.
- rl             Record length in characters (required only for U, W, and R type records).
- ex             Address of error subroutine.

#### KEY -- HASHING SUBROUTINE FOR DIRECT ACCESS FILE

A hashing subroutine is used to generate, from the key, an integer value for locating the record.

A user-coded randomizing subroutine may be specified for a DA file instead of the system-supplied default hash subroutine. A key analysis utility is available to help the user decide if his hash subroutine is more suitable for the file than the default subroutine. This subroutine should be added to a user library, as it must be supplied each time the file is processed.

In the user's main program the entry address of the hash subroutine must be declared external and set into the HRL field of the FIT prior to the first open of the file. During processing of the file the hash subroutine is called by DA with the following argument list:

- Key length, in characters
- Key, left justified and zero filled
- Number of home blocks
- Returned result

All arguments are integer, and the returned result must be non-negative. The value used is the returned result mod (number of home blocks minus one).

The following example illustrates how subroutine MYHASH is specified for file MYFILE. The hash result is the product of the words of the key.

```

PROGRAMS
INTEGER FIT(35)
EXTERNAL MYHASH
CALL FILEDA(FIT,3LLFN,6LMYFILE,3LHRL,MYHASH, . . .)

.
.
.

END
SUBROUTINE MYHASH(KL,KEY,HMB,RESULT)
INTEGER KEY(1),HMB,RESULT
KW=(KL+9)/10
DO 20 I=1,KW
20  RESULT=RESULT*KEY(I)
RETURN
END
    
```

## ERROR CHECKING

FORTRAN/Record Manager routines perform limited error checking to determine whether the call can be interpreted, but actual parameter values are not checked.

The following error conditions are detected, and a message appears in the dayfile:

FIT ADDRESS NOT SPECIFIED	Array name was not specified.
FORMAT ERROR	Parameters were not paired (FILExx), or required parameters were not specified (STOREF, IFETCH or SKIP).
UNDEFINED SYMBOL	A file information table field mnemonic or symbolic option was specified incorrectly; for example, an incorrect spelling, or the <u>of</u> parameter in OPENM was not specified as R, N or E.

Example of error message:

```

ERROR IN STOREF CALL
UNDEFINED SYMBOL  INPUT
    
```

## MULTIPLE INDEX PROCESSING

FORTRAN Extended provides the capability of multiple indexing for IS, DA, and AK files via the CYBER Record Manager.

Each multiple-indexed file has an associated alternate key index file. An alternate key index is a cross-reference table of alternate key values and IS, DA, or AK primary key values. The key-field position identifies each table, which consists of all the different alternate key values that occur in the records of the file. Associated with each alternate key value is a list of primary keys, each of which identifies a record containing the alternate key value.

To utilize the capabilities, the following statement should be used:

```
CALL FILExx(fit,2LXN,indxln, . . . )
```

xx IS, DA, or AK.

fit Name of an array containing the FIT.

indxln Name of the alternate key index file, specified as L format Hollerith constant.

XN may also be specified on a FILE control card.

To open the file, the following statement should be used:

```
CALL RMOPNX(fit,pd,of)
```

The parameters are the same as those of CALL OPENM. The file may be opened by a CALL OPENM instead of CALL RMOPNX if XN was specified on a FILE control card rather than by a CALL FILExx.

The following subroutine should be called to describe a key field to Record Manager when creating a new IS, DA, or AK file. It must be called once for each key field in the record.

```
CALL RMKDEF(fit,KW,KP,KL,KI,KT,KS,KG,KC)
```

fit Name of an array containing the FIT.

KW Word of record in which key starts (0 = first word)

KP Starting character position of key (0-9)

KL Key length in characters (1-255)

KI Summary index reserved (0)

KT Key type 0 = symbolic; 1 = signed integer; 2 = unsigned

KS Substructure for each primary key list in the index: I = index-sequential; F = FIFO; U (default) = unique; can be specified as L format Hollerith constant.

KG Size of repeating group in which key resides (default = 0).

KC Occurrences of group (default = 0).



To position the index file to the first primary key for a given alternate key value (KA), a CALL GET with  $rl = 1$  should be used.

To access the record for the next primary key in the record, a CALL GETN should be used.

For updating, the logical file name of the alternate key index file must be specified in the FILE card.

To retrieve the number of records containing an alternate key value (KA), a CALL GET with  $rl = 1$  may be used.

Before performing alternate key operations, the FIT parameters RKW (word displacement of alternate key in record), RKP (character position of key in record), and KL (key length) should be set with calls to STOREF. **I**



---

Mass storage input/output (MSIO) subroutines allow the user to create, access, and modify multi-record files on a random basis without regard for their physical position or internal structure. Each record in the file can be read or written at random without logically affecting the remaining file contents. The length and content of each record are determined by the user. A random file can reside on any mass storage device for which Record Manager word addressable file organization is defined. (The Record Manager Reference Manual and 7000 SCOPE Reference Manual contain details.)

## RANDOM FILE ACCESS

Random file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written, and can normally be read back only in the same order. This can be slow and inconvenient in applications where the order of writing and retrieving records differ and, in addition, it requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, a randomly accessible file capability is provided by the mass storage input/output subroutines.

In a random file, any record may be read, written or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random-access rotating mass storage device that can be positioned to any portion of a file. Thus, the entire concept of file position does not apply to a random file. The notion of rewinding a random file is, for instance, without meaning.

To permit random accessing, each record in a random file is uniquely and permanently identified by a record key. A key is an 18- or 60-bit quantity, selected by the user and included as a READMS or WRITMS call parameter. When a record is first written, the key in the WRITMS call becomes the permanent identifier for that record. The record can be retrieved later by a READMS call that includes the same key, and it can be updated by a WRITMS call with the same key.

When a random file is in active use, the record key information is kept in an array in the user's field length. The user is responsible for allocating the array space by a DIMENSION, type or similar array declaration statement, but must not attempt to manipulate the array contents. The array becomes the directory or index to the file contents. In addition to the key data, it contains the word address and length of each record in the file. The index is the logical link that enables the mass storage subroutines, in conjunction with Record Manager, to associate a user call key with the hardware address of the required record.

The index is maintained automatically by the mass storage subroutines. The user must not alter the contents of the array containing the index in any manner; to do so may result in destruction of the file contents. (In the case of a sub-index, the user must clear the array before using it as a sub-index; and read the sub-index into the array if an existing file is being reopened and manipulated. However, individual index entries should not be altered.)

Under SCOPE, when a permanent file that was created by mass storage input/output routines is to be modified, the EXTEND control card should be used to ensure that the new index is made permanent.

In response to an OPENMS call, the mass storage subroutines automatically clear the assigned index array. If an existing file is being reopened, the mass storage subroutines will locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to the mass storage device. When the file is reopened, by the same job or another job, the index is again read into the index array space provided, so that file manipulation may continue.

## **INDEX KEY TYPES**

There are two types of index key, name and number. A name key may be any 60-bit quantity except +0 or -0. A number key must be a simple positive integer, greater than 0 and less than or equal to (lngh - 1). The user selects the type of key by the (t) parameter. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. If the user should inadvertently attempt to reopen an existing file with an incorrect index type parameter, the job will be aborted. (This does not apply to sub-indexes chosen by STINDEX calls; proper index type specification is the sole responsibility of the user.) In addition, key types cannot be mixed within a file. Violation of this restriction may result in destruction of a file.

The choice between name and number keys is left entirely to the user. The nature of the application may clearly dictate one type or the other. However, where possible, the number key type is preferable. Job execution will be faster and less central memory space will be required. Faster execution occurs because it is not necessary to search the index for a matching key entry (as is necessary when a name key is used). Space is saved due to the smaller index array length requirement.

Example 1:

```
PROGRAM MS1 (TAPE3)

C CREATE RANDOM FILE WITH NUMBER INDEX.

    DIMENSION INDEX(11), DATA(25)
    CALL OPENMS (3,INDEX,11,0)

    DO 99 NRKEY=1,10
C          .
C          .
C (GENERATE RECORD IN ARRAY NAMED DATA.)
C          .
C          .
    99 CALL WRITMS (3,DATA,25,NRKEY)

    STOP
    END

PROGRAM MS2 (TAPE3)

C MODIFY RANDOM FILE CREATED BY PROGRAM MS1.
C NOTE LARGER INDEX BUFFER TO ACCOMMODATE TWO NEW
C RECORDS.

    DIMENSION INDEX(13), DATA(25), DATAMOR(40)
    CALL OPENMS (3,INDEX,13,0)

C READ 8TH RECORD FROM FILE TAPE3.
    CALL READMS (3,DATA,25,8)
C          .
C          .
C (MODIFY ARRAY NAMED DATA.)
C          .
C          .

C WRITE MODIFIED ARRAY AS RECORD 8 AT END OF
C INFORMATION IN THE FILE
    CALL WRITMS (3,DATA,25,8)

C READ 6TH RECORD.
    CALL READMS (3,DATA,25,6)
C          .
C          .
C (MODIFY ARRAY.)
C          .
```

```

C          .
C REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6.
  CALL WRITMS (3,DATA,25,6,1)
C READ 2ND RECORD INTO LONGER ARRAY AREA.
  CALL READMS (3,DATAMOR,25,2)
C          .
C          .
C (ADD 15 NEW WORDS TO THE ARRAY NAMED DATAMOR.)
C          .
C          .
C CALL FOR IN-PLACE REWRITE OF RECORD 2. IT WILL
C DEFAULT TO A NORMAL WRITE AT END-OF-INFORMATION
C SINCE THE NEW RECORD IS LONGER THAN THE OLD ONE,
C AND FILE SPACE IS THEREFORE UNAVAILABLE.
  CALL WRITMS (3,DATAMOR,40,2,-1)
C READ THE 4TH AND 5TH RECORDS.
  CALL READMS (3,DATA,25,4)
  CALL READMS (3,DATAMOR,25,5)
C          .
C          .
C (MODIFY THE ARRAYS NAMED DATA AND DATAMOR.)
C          .
C          .
C WRITE THE ARRAYS TO THE FILE AS TWO NEW RECORDS.
  CALL WRITMS (3,DATA,25,11)
  CALL WRITMS (3,DATAMOR,25,12)
C
C STOP
C END

```

This example creates and modifies a random file using a number index.  
 Example 2:

```

PROGRAM MS3 (TAPE7)
C CREATE A RANDOM FILE WITH NAME INDEX.
C
C DIMENSION INDEX(9), ARRAY(15,4)
C DATA REC1,REC2/7HRECORD1,≠RECORD2≠/
C          .
C          .
C (GENERATE DATA IN ARRAY AREA.)
C          .
C          .

```

```

C WRITE FOUR RECORDS TO THE FILE. NOTE THAT
C KEY NAMES ARE RECORD(N).
    CALL WRITMS (7,ARRAY(1,1),15,REC1)
    CALL WRITMS (7,ARRAY(1,2),15,REC2)
    CALL WRITMS (7,ARRAY(1,3),15,7RRECORD3)
    CALL WRITMS (7,ARRAY(1,4),15,≠RECORD4≠)

C CLOSE THE FILE.

    CALL CLOSMS (7)

    STOP
    END

```

This example uses a name index for a random file.

## MULTI-LEVEL FILE INDEXING

When a file is opened by an OPENMS call, the mass storage routines clear the array specified as the index area, and if the call is to an existing file, locates the file index and reads it into the array. This creates the initial or master index.

The user can create additional indexes (sub-indexes) by allocating additional index array areas, preparing the area for use as described below, and calling the STINDEX subroutine to indicate to the mass storage routine the location, length and type of the sub-index array. This process may be chained as many times as required, limited only by the amount of central memory space available. (Each active sub-index requires an index array area.) The mass storage routine uses the sub-index just as it uses the master index; no distinction is made.

A separate array space must be declared for each sub-index that will be in active use. Inactive sub-indexes may, of course, be stored in the random file as additional data records.

The sub-index is read from and written to the file by the standard READMS and WRITMS calls, since it is indistinguishable from any other data record. Although the master index array area is cleared by OPENMS when the file is opened, STINDEX does not clear the sub-index array area. The user must clear the sub-index array to zeros. If an existing file is being manipulated and the sub-index already exists on the file, the user must read the sub-index from the file into the sub-index array by a call to READMS before STINDEX is called. STINDEX then informs the mass storage routine to use this sub-index as the current index. The first WRITMS to an existing file using a sub-index must be preceded by a call to STINDEX to inform the mass storage routine where to place the index control word entry before the write takes place.

If the user wishes to retain the sub-index, it must be written to the file after the current index designation has been changed back to the master index, or a higher level sub-index by a call to STINDEX.

## I MASTER INDEX

The master index type for a given file is selected by the t parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created; attempts to do so by reopening the file with the opposite type index are treated as fatal errors.

## SUB-INDEX

The sub-index type can be specified independently for each sub-index. A different sub-index name/number type can be specified by including the t parameter in the STINDEX call. If t is omitted, the index type remains the same as the current index. Intervening calls which omit the t parameter do not change the most recent explicit type specification. The type remains in effect until changed by another STINDEX call.

STINDEX cannot change the type of an index which already exists on a file. The user must ensure that the t parameter in a call to an existing index agrees with the type of the index in the file. Correct sub-index type specification is the responsibility of the user; no error message is issued.

Example:

```
PROGRAM MS4 (TAPE2)

C GENERATE SUBINDEXED FILE WITH NUMBER INDEX. FOUR
C SUBINDEXES WILL BE USED, WITH NINE DATA RECORDS
C PER SUBINDEX, FOR A TOTAL OF 36 RECORDS.

DIMENSION MASTER(5), SUBIX(10), RECORD(50)
CALL OPENMS (2,MASTER,5,0)

DO 99 MAJOR=1,4

C CLEAR THE SUBINDEX AREA.
DO 77 I=1,10
77 SUBIX(I)=0

C CHANGE THE INDEX IN CURRENT USE TO SUBIX.
CALL STINDEX (2,SUBIX,10)

C GENERATE AND WRITE NINE RECORDS.
DO 88 MINOR=1,9
C .
C .
```



```

C WRITE A RECORD.
88 CALL WRITMS (2,RECORD,50,MINOR)

C CHANGE BACK TO THE MASTER INDEX.
CALL STINDX (2,MASTER,5)

C WRITE THE SUBINDEX TO THE FILE.
CALL WRITMS (2,SUBIX,10,MAJOR,0,1)

99 CONTINUE

C READ THE 5TH RECORD INDEXED UNDER THE 2ND SUBINDEX.
CALL READMS (2,SUBIX,10,2)
CALL STINDX (2,SUBIX,10)
CALL READMS (2,RECORD,50,5)

C      .
C      .
C (MANIPULATE THE SELECTED RECORD AS DESIRED.)
C      .
C      .

STOP
END

```

PROGRAM MS5 (INPUT,OUTPUT,TAPE9)

```

C CREATE FILE WITH NAME INDEX AND TWO LEVELS OF SUBINDEX.

DIMENSION STATE(101), COUNTY(501), CITY(501), ZIP(100)
INTEGER STATE, COUNTY, CITY, ZIP
10 FORMAT (A10,I10)
11 FORMAT (I10)
12 FORMAT (5X,8I15)

CALL OPENMS (9,STATE,101,1)

C READ MASTER DECK CONTAINING STATES, COUNTIES, CITIES
AND ZIP CODES.
DO 99 NRSTATE=1,50
READ 10,STATNAM, NRCNTYS

C CLEAR THE COUNTY SUBINDEX.
DO 21 I=1,501
21 COUNTY(I)=0

```

DO 98 NRCN=1,NRCNTYS  
READ 10, CNTYNAM, NRCITYS

C CLEAR THE CITY SUBINDEX.

DO 31 I=1,501

31 CITY(I)=0

CALL STINDX (9,CITY,501)

DO 97 NRCY=1,NRCITYS

READ 10, CITYNAM, NRZIP

DO 96 NRZ=1,NRZIP

96 READ 11,ZIP(NRZ)

97 CALL WRITMS (9,ZIP,100,CITYNAM)

CALL STINDX (9,COUNTY,501)

98 CALL WRITMS (9,CITY,501,CNTYNAM)

CALL STINDX (9,STATE,101)

99 CALL WRITMS (9,COUNTY,501,STATNAM,

C FILE IS GENERATED. NOW PRINT OUT LOCAL ZIP CODES.

CALL STINDX (9,STATE,101)

CALL READMS (9,COUNTY,501,≠CALIFORNIA≠)

CALL STINDX (9,COUNTY,501)

CALL READMS (9,CITY,501,≠SANTA CLARA≠)

CALL STINDX (9,CITY,501)

CALL READMS (9,ZIP,100,≠SUNNYVALE≠)

PRINT 12, ZIP

CALL STINDX (9,STATE,101)

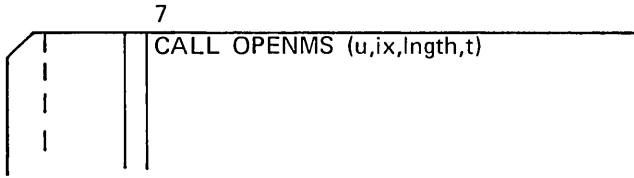
STOP

END

## MASS STORAGE SUBROUTINES

Object time input/output subroutines control the transfer of records between central memory and mass storage. These routines use the word addressable feature available through Record Manager.

### OPENMS



u Unit designator.

ix Name of the array containing the master index.

lngth Length of master index

for a number index:  $\text{lngth} \geq (\text{number of entries in master index}) + 1$

for a name index:  $\text{lngth} \geq 2 * (\text{number of entries in master index}) + 1$

t Type of index.

t = 0 file has a number master index

t = 1 file has a name master index

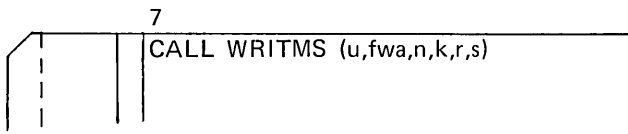
OPENMS opens the mass storage file and informs Record Manager that it is a random (word addressable) file. The array specified in the call is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

Example:

```
DIMENSION I(11)
CALL OPENMS (5,I,11,0)
```

These statements prepare for random input/output on unit 5 using an 11-word master index of the number type. If the file already exists, the master index is read into memory starting at address I.

### WRITMS



u Unit designator.

fwa Name of the array in central memory (address of first word).

- n      Number of 60-bit words to be transferred.
- k      Record key.  
          for number index:  $k = 1 \leq k \leq \text{length} - 1$   
          for name index     $k = \text{any 60-bit quantity except } \pm 0$
- r      Rewrite.  
           $r = 1$     Rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length.  
           $r = -1$  Rewrite in place if space available, otherwise write at end of information.  
           $r = 0$     No rewrite; write normally at end-of-information (default value).
- s      Sub-index flag.  
           $s = 1$     Write sub-index marker flag in index control word for this record.  
           $s = 0$     Do not write sub-index marker flag in index control word (default value).

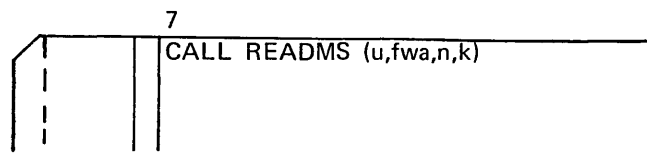
WRITMS transmits data from central memory to the selected mass storage device. Except under SCOPE 2.1, Record Manager operates more efficiently if n is always a multiple of 64. The r parameter can be omitted if the s parameter is also omitted. The s parameter is for future file editing routines. Current routines do not test the flag, but the user should include this parameter in new programs (when appropriate) to facilitate transition to a future edit capability.

Example:

```
CALL WRITMS (3,DATA,25,6,1)
```

This statement unconditionally rewrites in place on file TAPE3, starting at the address of the array named DATA, a 25-word record with an index number key of 6. The default value is taken for the s parameter.

### READMS



- u      Unit designator
- fwa    Name of the array in central memory (address of first word)
- n      Number of 60-bit words to be transferred.

- k      Record key  
 for number index:  $k = 1 \leq k \leq \text{length} - 1$   
 for name index:  $k = \text{any 60-bit quantity except } \pm 0$

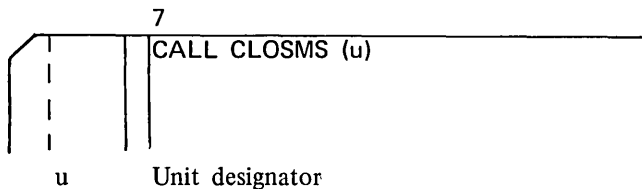
READMS transmits data from mass storage to central memory. Except under SCOPE 2.1, Record Manager operates more efficiently if n is always a multiple of 64.

Example:

```
CALL READMS (3,DATAMOR,25,2)
```

This statement reads the first 25 words of record 2 from unit 3 (TAPE3) into central memory starting at the address of the array DATAMOR.

### CLOSMS



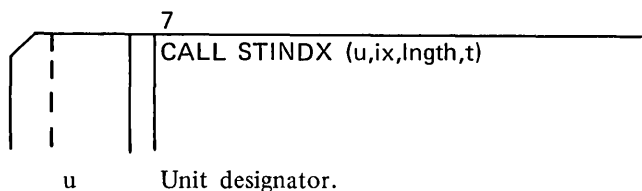
CLOSMS writes the master index from central memory to the file and closes the file. CLOSMS is provided to close a file so that it can be returned to the operating system before the end of a FORTRAN run, to preserve a file created by an experimental job that might subsequently abort, or for other special purposes.

Example:

```
CALL CLOSMS (2)
```

This statement closes the file TAPE2.

### STINDEX



- u      Unit designator.
- ix      Name of the array in central memory containing the sub-index (first word address).
- length      Length of sub-index  
 for a number index:  $\text{length} (\text{number of entries in sub-index}) + 1$   
 for a name index:  $\text{length } 2 * (\text{number of entries in sub index}) + 1$

t        Type of index. If omitted, t is the same as the current index.

t = 0     File has a number sub-index

t = 1     File has a name sub-index

STINDEX selects a different array to be used as the current index to the file. The call permits a file to be manipulated with more than one index. For example, when the user wishes to use a sub-index instead of the master index, STINDEX is called to select the sub-index as the current index. The STINDEX call does not cause the sub-index to be read or written; that task must be carried out by explicit READMS or WRITMS calls. It merely updates the internal description of the current index to the file.

Example 1:

```
DIMENSION SUBIX (10)
CALL STINDEX (3,SUBIX,10,0)
```

These statements select a new index, SUBIX, for file TAPE3 with an index length of 10. The records referenced via this sub-index use a number index.

Example 2:

```
DIMENSION MASTER (5)
CALL STINDEX (2,MASTER,5)
```

These statements select a new index, MASTER, from file TAPE2 with an index length of 5 and index type unchanged from the last index used.

## COMPATABILITY WITH PREVIOUS MASS STORAGE ROUTINES

FORTRAN Extended mass storage routines and the files they create are not compatible with mass storage routines and files created under versions of FORTRAN Extended (before version 4) Major internal differences in the file structure were necessitated by adding the Record Manager interface. However, source programs are fully compatible. Any source program that compiled and executed successfully under earlier versions will do so under this version, provided that all file manipulations were and continue to be executed by mass storage routines.

## ERROR MESSAGES

Random file processing errors are fatal; the job terminates and one of the following error messages is printed:

### 97 INDEX NUMBER ERR

The index number key is negative, zero, or greater than the index buffer length minus one.

### 98 FILE ORGANIZATION ERR

During the initial OPENMS call, mass storage routines set the file organization as word addressable (FO=WA) and the record type to U (RT=U). A conflicting file organization was specified in an external subroutine call or FILE control card.

### 99 WRONG INDEX TYPE

An attempt was made to open an existing file with the wrong index type parameter. File index type is permanently determined when a file is created.

### 100 INDEX IS FULL

WRITMS was called with a name index key, and the end of the index buffer occurred before a match was found. Either the name key is in error, or the buffer must be lengthened.

### 101 DEFECTIVE INDEX CONTROL WORD

This message may occur for either of two reasons:

1. An OPENMS for an existing file found the master index control word has been destroyed. Since this word was properly set when the file was last closed, the user should check for an external cause of file destruction.
2. A READMS or WRITMS call has encountered a defective index control word. Check for an improperly cleared sub-index array, for a program sequence that writes into an index array (other than the required initial zeroing) or for an external cause of file destruction.

### 102 RECORD LENGTH EXCEEDS SPACE AVAILABLE

1. During an OPENMS call, not enough index buffer space was provided for the master index of an existing file.
2. During a WRITMS call with in-place rewrite requested ( $r = +1$ ), the new record length exceeded the old record length.

### 103 6RM/7DM I/O ERR NUMBER 000

Record Manager has detected an error; the actual error number appears in the message. Refer to Record Manager Reference Manual to identify the source of the error.

### 104 INDEX KEY UNKNOWN

No data record exists for the user's index key. This error may be diagnosed for a READMS call or for a WRITMS call with rewrite requested ( $r = +1$ ).





---

The following information will be useful only to the assembly language programmer.

## REGISTER NAMES

The compiler changes some legal FORTRAN names so that FORTRAN object code can be used as COMPASS input. When a two-character name begins with A, B, or X and the last character is 0 to 7, the compiler adds a currency symbol (\$) to the name for the object code listing. (A0-A7, B0-B7, and X0-X7 represent registers to the COMPASS assembler which may be used by the FORTRAN Extended compiler).

## EXTERNAL PROCEDURE NAMES (PROCESSOR SUPPLIED)

### CALL-BY-VALUE

The name of a system supplied external procedure called by value is suffixed with a decimal point. The entry point is the symbolic name of the external procedure and a decimal point suffix. For example, EXP, COS, CSQRT.

The names of all external procedures called by value are listed in table 8-2 Basic External Functions, section 8, part 1. A procedure will not be called by value and the name will not be suffixed with a decimal point if it appears either in an EXTERNAL statement or an overriding type statement, or if option T, D, or OPT=0 is specified on the FTN control card.

### CALL-BY-NAME

The call-by-name entry point is the symbolic name of the external procedure with no suffix.

External procedures called by name appear in section 8, part 1 under the heading Additional Utility Subprograms. Any name which appears in table 8-1 Intrinsic Functions or table 8-2 Basic External Functions are called by name if it appears in an EXTERNAL statement or in an overriding type statement; those listed as Basic External Functions are also called by name if option T, D, or OPT=0 is specified on the FTN control card.



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

STATEMENTS	
1	OVERLAY
2	PROGRAM* FUNCTION* SUBROUTINE* BLOCK DATA
3	IMPLICIT
4	type COMMON DIMENSION EQUIVALENCE EXTERNAL* LEVEL
5	Statement function* definitions
6	ENTRY* Executable statements*
7	END

\* F O R M A T  
 † N A M E L I S T  
 \* D A T A

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

The following description of the arrangement of code and data within PROGRAM, SUBROUTINE and FUNCTION program units does not include the arrangement of data within common blocks because this arrangement is specified by the programmer. However, the diagram of a typical memory layout at the end of this section illustrates the position of blank common and labeled common blocks.

## SUBROUTINE AND FUNCTION STRUCTURE

The code within subprograms is arranged in the following blocks (relocation bases) in the order given.

START.	Code for the primary entry and for saving A0
VARDIM.	Address substitution code and any variable dimension initialization code
ENTRY.	Either a full word of NO's or nothing
CODE.	Code generated by compiling: Executable statements Parameter lists for external procedure references within the current procedure Storage statements DO loops and optimizing temporary use
DATA.	Storage for simple variables, FORMAT statements, and program constants
DATA..	Storage for arrays other than those in common
HOL.	Storage for Hollerith constants
FORMAL PARAMETERS.	One local block for each dummy argument in the same order as they appear in the subroutine statement, to hold tables used in address substitution for processing references to dummy arguments

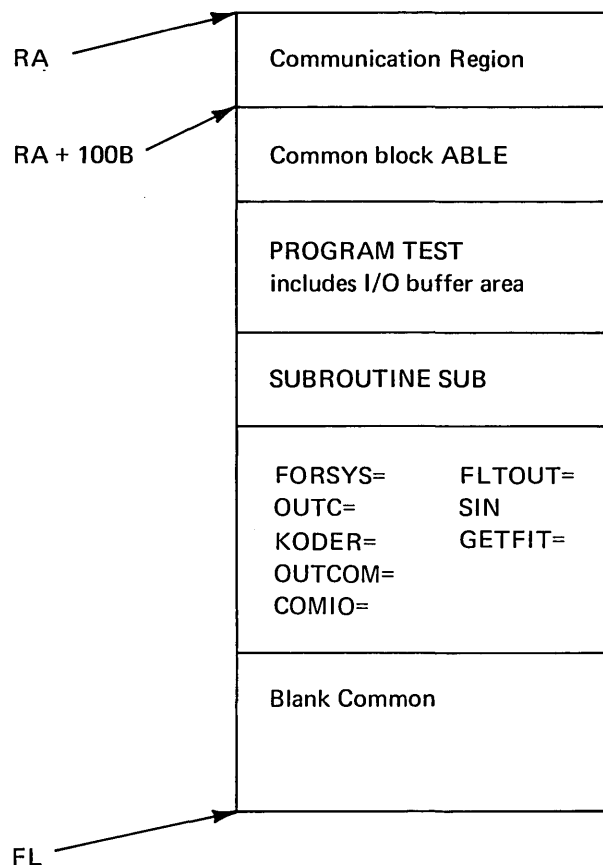
## MAIN PROGRAM STRUCTURE

START.	Input/output file buffers and a table of file names specified in the program statement
CODE.	Transfer address code plus the code specified for the subroutine and function CODE. block
DATA. DATA.. HOL.	Same as SUBROUTINE and FUNCTION structure

## MEMORY STRUCTURE

Memory is not cleared, and subprograms are loaded as they appear in the input file starting at the program's reference address (RA) + 100B, toward the user's field length (FL). RA to RA + 100B is the communication region used by the operating system. Labeled common blocks are loaded prior to the subprogram in which they are first referenced. Library routines are loaded immediately after the last subprogram and are followed by blank common.

Typical memory layout:





Both SUBROUTINES and FUNCTIONS may be written in COMPASS Assembly language and called from a FORTRAN source program. For either, register A0 is the only register that must be restored to its initial condition when the subprogram returns control to the calling routine.

When a FORTRAN generated subprogram is called, the calling routine must not depend on values being preserved in any registers other than A0.

### CALL BY NAME AND CALL BY VALUE

To increase speed, arguments to library functions are normally passed to subprograms by placing their values in the registers. This method is call by value. For user defined subprograms, the address of the arguments are passed to the subprogram. This method is call by name.

### CALL BY NAME SEQUENCE

The FORTRAN compiler uses the call by name sequence when a subroutine or function name differs from any of those listed in table I-8-1 and I-8-2. Call by name is also used when a listed subroutine or function also appears in an EXTERNAL or overriding type statement, or (except in the case of intrinsic functions) the program unit specifies D, T, or OPT=0 on the FTN control card.

The call by name sequence generated is shown below:

SAI	Address of the argument list (if parameters appear)	
+RJ	Subprogram name	
-VFD	12/line number, 18/trace word address	
	line number	Source line number of statement containing the reference
	trace word address	Address of the trace word for the calling routine

Arguments in the call must correspond with the argument usage in the called routine, and they must reside in the same level.

The argument list consists of consecutive words in the following form followed by a zero word. The sign bit will be set in the argument list for any argument entry address that is LCM§ or ECS‡.

VFD 60/address of argument

§ Applies only to CONTROL DATA CYBER 70/Model 76 and 7600 computers.

‡ Applies only to CONTROL DATA CYBER 70/Models 72, 73, and 74, CYBER 170, and 6000 Series computers.

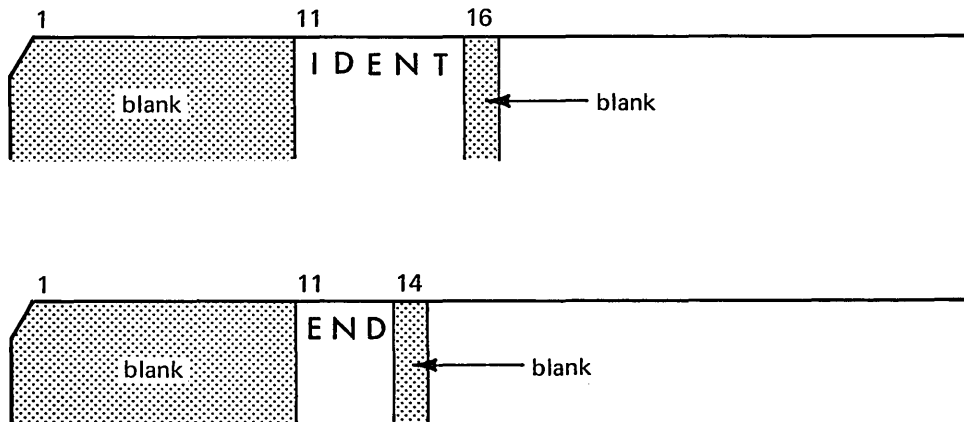
## CALL BY VALUE SEQUENCE

For increased efficiency the compiler generates a call by value code sequence for references to library functions if the function name does not appear in an EXTERNAL or overriding type statement and (in the case of external functions only) the D, T, or OPT=0 options on the FTN control card are not specified. The name of any library function called by value or generated in line must appear in an EXTERNAL statement in the calling routine if the call by name calling sequence is required (section 8, part 1 lists the library functions called by value and generated in-line).

The call by value code sequence consists of code to load the arguments into X1 through X4, followed by an RJ instruction to the function. The second register loaded for a double precision or complex argument contains the least significant or imaginary part of the argument.

## COMPASS SUBPROGRAMS

Subprograms in COMPASS assembly language can be intermixed with FORTRAN coded subprograms in the source deck. COMPASS subprograms must begin with a card containing the word IDENTb, in columns 11-16, and terminate with a card containing the word ENDb, in columns 11-14 (b denotes a blank). Columns 1-10 of the IDENT and END cards must be blank.





If the COMPASS subprogram changes the value of A0, it must restore the initial contents of A0 upon returning control to the calling subprogram. When the COMPASS subprogram is entered by a function reference, the subprogram must return the function result in X6 or X6 and X7 with the least significant or imaginary part of the double precision or complex result appearing in X7.

The COMPASS assembler normally requires the system text SYSTEXT, which is the default for the S parameter. The amount of storage available depends on installation options. Insufficient storage for SYSTEXT causes an error. The user may need to specify a larger field length for compilation or a different option for S. See the appropriate operating system reference manual and section I-11 of this manual for more details on systems texts.

Example:

The following page contains an example of a simple COMPASS Function and the calling FORTRAN main program. The parity function, PF, returns an integer value; therefore it must be declared integer in the calling program. The argument to PF may be either real or integer.

The title and comments are unnecessary; they are included to encourage good programming practice. The following is a recommended convention.

```
PF      EQ      *+1S17  ENTRY/EXIT
```

This statement causes a jump to 400 000<sub>8</sub> plus the location of the entry point of the routine if the function is not entered with a return jump. This results in a mode error that can quickly be identified. Since A0 is not used in this subprogram, it need not be restored.

**SOURCE DECK**

*job card*

MAP(OFF)  
FTN(R=0)

LGO.

7/8/9 in column 1.

```
PROGRAM NPSAMP(OUTPUT)
INTEGER PF, PVAL(24)
001I=1,24
1 PVAL(I)=PF(I)
PRINT2,(I,I=1,24),PVAL
2 FORMAT(32H0INTEGERS AND THEIR PARITY BELOW/(24I3))
STOP
END
```

} main program

```
IDENT PF
ENTRY PF
PF TITLE PF - COMPUTE PARITY OF WORD.
COMMENT COMPUTE PARITY OF WORD.
PF SPACE 4,11
*** PF - COMPUTE PARITY OF WORD.
```

```
*
* FORTRAN SOURCE CALL --
*
* PARITY = PF (ARG)
*
* RESULT = 1. IFF ARG HAS ODD NUMBER OF BITS SET.
* = 0. OTHERWISE.
**
** ENTRY (X1) = ADDRESS OF ARGUMENT.
* EXIT (X6) = RESULT.
```

```
PF EQ *+1S17 ENTRY/EXIT...
SA2 X1 ← get the argument value
CX3 X2 ← count the 1 bits in X2 and leave result in X3
MX0 -1 ← form a mask in X0
RX6 -X0*X3 ISOLATE LOWEST BIT ← put result into X6
EQ PF EXIT..
```

END

6/7/8/9 in column 1.

**OUTPUT**

INTEGERS AND THEIR PARITY BELOW

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	1	0	1	0	0	1	1	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	0

## ENTRY POINT

For subprograms written in FORTRAN, the compiler uses the following conventions in generating code:

The entry point of the subprogram (for reference by an RJ instruction) is preceded by two words. The first is a trace word for the subprogram; it contains the subprogram name in left justified display code (blank filled) in the upper 42 bits and the subprogram entry address in the lower 18 bits. The second word is used to save the contents of A0 upon entry to the subprogram. The subprogram restores A0 upon exit.

Trace word:	VFD	42/name, 18/entry address
A0 word:	DATA	0
Entry point:	DATA	0

## RESTRICTIONS ON USING LIBRARY FUNCTION NAMES

Functions written in FORTRAN that have library function names listed in tables 8-1 or 8-2, such as AMAX1 or SQRT, must be declared EXTERNAL in the calling program unit. This declaration is necessary because the compiler produced functions always use the call by name calling sequence.

Functions written in COMPASS that have basic external library names listed in table 8-2, such as SQRT, should be written using the call by name sequence when they are declared EXTERNAL in the calling routine; or they should use the call by value rules if they are not declared EXTERNAL.

Functions written in COMPASS that have intrinsic library names listed in table 8-1, such as AMAX1, must be declared EXTERNAL in the calling routine; otherwise in-line coding is generated for them (the COMPASS coding is ignored). Furthermore, the call by name sequence must be used.

If a library function, called by value, is to be overridden by a routine coded in COMPASS, the COMPASS routine must use the library function name with a period appended as the entry point name (e.g., SIN.) to use the call by value calling sequence.

The following sample illustrates the code generated for: a library function call, SQRT; an external function call, ZEUS; and a reference to an intrinsic (in-line) function, AMAX1.

The coding generated for the external function, ZEUS, is illustrated also.

```
MAP(OFF)
FTN(R=0,OL)
7/8/9 in column 1
  PROGRAM SURLNK
  X=SQRT(7.0)
  Y=ZEUS(X,1.0)
  END
  FUNCTION ZEUS(ARG1,ARG2)
  ZEUS=AMAX1(ARG1,ARG2,0.)
  RETURN
  END
6/7/8/9 in column 1
```

```

PROGRAM      SUBLNK
              PROGRAM SUBLNK
              X=SQRT(7.0)
              Y=ZEUS(X,1.0)
              END

```

```

PROGRAM      SUBLNK      OPT=1

```

```

IDENT      SUBLNK
USEBLK
LDSET      LIB=FORTRAN
LDSET      LIB=SYSIO
USE START.

```

```

000000 000002 START. LOCAL
000002 000000 VARDIM. LOCAL
000002 000000 ENTRY. LOCAL
000002 000011 CODE. LOCAL
000013 000004 DATA. LOCAL
000017 000000 DATA.. LOCAL
000017 000000 HOL. LOCAL

```

```

EXTERNALS
END.      ZEUS      SQRT.      Q8NTRY.

```

```

000000 START.      777777777777777766167
000001 START.      23250214161355000002

000002 CODE.      5110000000      START.
                   0100000000      <EXT>

000013 DATA.
000013 DATA.      1722700000000000000000
000014 DATA.      1720400000000000000000

000015 DATA.
000016 DATA.

```

```

FILES. BSS 0B
DATA      777777777777777766167B
TRACE     SUBLNK,SUBLNK
USE CODE.
PENTRY   SUBLNK
SA1      FILES.
RJ       Q8NTRY.
USE DATA.
USE DATA..
USE DATA.
CON. BSS 0B
DATA 172270000000000000000000B } constant table
DATA 172040000000000000000000B }
EXT      END.
EXT      ZEUS
EXT      SQRT.
EXT      Q8NTRY.
X        BSS 1B
Y        BSS 1B

```

```

*
000003 CODE.      5110000013      DATA.
                   0100000000      <EXT>
000004 CODE.      5160000015      DATA.
                   5110000010      CODE.
000005 CCDE.      0100000000      <EXT>
                   0003000001
000006 CCDE.      5160000016      DATA.
                   5110000001      START.
000007 CCDE.      0400000000      <EXT>
000010 CODE.
000010 CODE.      00000000000000000015 DATA.
000011 CCDE.      00000000000000000014 DATA.
000012 CCDE.      00000000000000000000

X        BSS 1B
Y        BSS 1B
USE      CODE.
SA1      CON. ← get actual parameter into X1
RJ       SQRT.
SA6      X
SA1      [AP1 ← get address of parameter list into X1
RJ       ZEUS,3B
SA6      Y
SA1      TRACE.
EQ       END.
[AP1     BSS 0B
        APL X
        APL CON.+1B } parameter address list
        APL
END      SUBLNK

```

FUNCTION ZEUS

FUNCTION ZEUS(ARG1,ARG2)
ZEUS=AMAX1(ARG1,ARG2,0.)
RETURN
END

FUNCTION ZEUS OPT=1

IDENT ZEUS
USEBLK
LOSET LIB=FORTRAN
LOSET LIB=SYSIO
USE START.

000000 000006 START. LOCAL
000006 000000 VARDIM. LOCAL
000006 000000 ENTRY. LOCAL
000006 000005 CODE. LOCAL
000013 000001 DATA. LOCAL
000014 000000 DATA.. LOCAL
000014 000000 HOL. LOCAL
000014 000000 ARG1 LOCAL
000014 000000 ARG2 LOCAL

000000 START. 3205252355555000004 000000 TRACE ZEUS,ZEUS,26 name of program unit and entry point address
000001 START. 00000000000000000000 cell to save A0 in
000002 START. 51400000131064446000 PENTRY ZEUS,ENTRY.,1 restores A0 on exit
000003 START. 51300000015203000000
000004 START. 04004000026100046000 entry point
000005 START. 74000540105160000001 saves A0 and sets A0 to the new A1

FORPAR ARG1
FORPAR ARG2
USE DATA.
USE DATA..
USE DATA.
VALUE. BSS 1B
USE CODE.

000013 DATA.

000006 CCDE.

54500
5040000001
53350

000007 CODE.

53240
31032
21073
11702

000010 CODE.

15630
36067
22500
21573

000011 CCDE.

15705
5170000013

000012 CCDE.

0400000002

DATA.
START.

LINE 2
SA5 A0
SA4 A0+1B
SA3 X5
SA2 X4
FX0 X3-X2
AX0 73B
BX7 X0\*X2
BX6 -X0\*X3
IX0 X6+X7
LX5 B0,X0
AX5 73B
BX7 -X5\*X0
SA7 VALUE.
EQ EXIT.
ENC



If a FORTRAN program to be run under SCOPE's INTERCOM or under the KRONOS or NOS Time-Sharing System calls for input/output operations through the user's remote terminal, all files to be accessed through the terminal must be formally associated with the terminal at the time of execution.

In particular, the file INPUT must be connected to the terminal if data is to be entered there and an alternate logical unit is not designated in the READ statement. The file OUTPUT must be connected to the terminal if execution diagnostics are to be displayed or printed at the terminal, or if data is to be displayed or printed there and an alternate logical unit is not designated in the WRITE or PRINT statement. These files are automatically connected to the terminal when the program is executed under either KRONOS or NOS or under the RUN command of the EDITOR utility of INTERCOM.

For a FORTRAN program run under INTERCOM, any file (including INPUT and OUTPUT) can be connected to the terminal by the CONNECT command. In addition, the user can connect any file from within the program by using either of the statements:

CALL CONNEC (fd,cs)

CALL CONNEC (fd)

- fd     file designator: fd can be a logical unit number u, a Hollerith constant nLfilename, or a simple integer variable with a value of u or nLfilename. u is an integer constant from 1 to 99 (associated by the compiler with the file name TAPEu); filename is a file name of 1 to 6 letters or digits beginning with a letter.
  
- cs     character set designator: cs should be an integer constant or an integer variable with a value of 0 to 2, in accordance with the character code set to be used for the data entered or displayed at the terminal:
  - 0     display code
  - 1     ASCII-95 code
  - 2     ASCII-256 code

---

<sup>†</sup>Applies only to INTERCOM, KRONOS Time-Sharing System, or NOS Time-Sharing System. More information about INTERCOM is in the INTERCOM Reference Manual and the INTERCOM Interactive Guide for Users of FORTRAN Extended. More information about KRONOS is in the KRONOS 2.1 Reference Manual and the KRONOS 2.1 Time-Sharing User's Reference Manual. More information about NOS is in the NOS 1.0 Reference Manual and the NOS 1.0 Time-Sharing User's Reference Manual.

If `cs` is not specified, it is set to 0. If display code is selected, input/output operations should be formatted, list-directed, NAMELIST, or buffered. If either of the ASCII codes is selected, input/output operations should be either formatted or buffered. When a CALL CONNEC specifies a file already connected with the character set specified, the call is ignored. If the file specified is already connected with a character set other than that specified, `cs` is reset accordingly.

Data input or output through a terminal under INTERCOM is represented ordinarily in a CDC 64-character, ASCII 64-character, or CDC 63-character set, depending on installation option. For these sets, ten characters in 6-bit display code are stored in each central memory word. As described above, a terminal user can specify from within a FORTRAN program that data represented in an ASCII 95-character set (providing the capability for recognizing lowercase letters) or an ASCII 256-character set (providing the capability for recognizing lowercase letters, control codes, and parity) be input or output through the terminal. For the ASCII 95-character and 256-character sets, characters are stored in five 12-bit bytes in each central memory word. Characters in the ASCII 95-character set are represented in 7-bit ASCII code right justified in each byte with binary zero fill; characters in the ASCII 256-character set are represented in 8-bit ASCII code right justified in each byte with binary zero fill. When data represented in either ASCII character set code is transferred with a formatted I/O statement, the maximum record length should be specified in the PROGRAM statement as twice the number of characters to be transferred (see section I-7).

When the ASCII 95-character or 256-character set has been specified for terminal input/output under INTERCOM, blanks following the end of data on each line are not translated into ASCII code but are retained in display code (as 55g). Unless the user eliminates them, these blanks will appear on output as lowercase `m` characters (two blanks in display code translates to one `m` in ASCII code). For formatted input, the user can identify the end of data on a line by scanning data entered in nR2 format until the Hollerith constant 2Rbb (`b` = blank) is found. For buffered input, the end can be determined by reading the data into an array, manipulating it with a DECODE statement, and then scanning as with formatted input.

For a FORTRAN program run under KRONOS or NOS, any file can be connected to the terminal by the ASSIGN command. In addition, the user can connect any file from within the program by using the statement:

```
CALL CONNEC (fd)
```

`fd`, the file designator, should be specified as described above for programs run under INTERCOM.

Data input or output through a terminal under KRONOS or NOS is represented ordinarily in a standard 61-character set. However, the user can elect to have data represented in an ASCII 128-character set (which provides the capability for recognizing control codes and lowercase, as well as uppercase, letters) by entering the ASCII command. Characters contained in the standard set are stored internally in 6-bit display code, whether or not the ASCII command has been entered. The additional characters which complete the ASCII 128-character set are stored internally in 12-bit display code if the ASCII command has been entered; otherwise, they are mapped into the standard 61-character set and stored internally in 6-bit display code.

Under SCOPE, KRONOS, and NOS, if a file specified in a CALL CONNEC exists as a local file but is not connected at the time of the call, the file's buffer is flushed before the file is connected to the terminal. Any file can be disconnected from within a FORTRAN program by the statement:

```
CALL DISCON (fd)
```



This request is ignored if the specified file is not connected. After execution of this statement, the specified file remains local to the terminal. In addition, if the file existed prior to connection, the file name is re-associated with the information contained on the device where the file resided prior to connection. Data written to a connected file is not contained in the file after it is disconnected.

All files to be connected or disconnected during program execution must be declared in the PROGRAM statement. An attempt to connect or disconnect an undeclared file results in a diagnostic fatal to execution.

Calls to CONNEC and DISCON are ignored when programs are not executed under INTERCOM or interactively under KRONOS or NOS.

Examples:

```
CALL CONNEC (6)

K = 4LAGES
CALL CONNEC (K)

CALL CONNEC (6,2)

CALL CONNEC (4LDATA,1)

CALL DISCON (6)
```



During a typical compilation and execution, source program, reference map, and core map listings are produced (unless LIST,NONE is in effect).

A header line at the top of each page of compiler output contains the program unit type and name, the machine used and the target machine for which the compiler was assembled, control card options, compiler version and mod-level, date, time, and page number.

The source program is listed 60 lines per page (including headers); every fifth source line is numbered. These numbers are used in the error messages and in the cross reference map.

The compiler produces a reference map for each routine compiled. The compiler generated addresses assume loading of program units starts at location 0. A description of the reference map is described in section III-1.

A map is produced by the loader at load time. In this map, the user program starts at relative address 101g. (The first 101 words, 0-100, serve as the communication region between the operating system and the user program.) Refer to the Loader Reference Manual for details of the load map.

To find the address of a variable, the address of the program unit, which appears in the load map, is added to the address of the variable which appears in the reference map. All locations and addresses in the reference map and the core map are in octal.

For example:

VARIABLES	SN	TYPE
0 A		REAL
17 AVG		REAL
20 I		INTEGER
0 J		INTEGER

PROGRAM AND BLOCK ASSIGNMENTS.			
BLOCK	ADDRESS	LENGTH	FILE
VARDIM2	101	2141	LGO
SET	2242	34	LGO
IOTA	2276	15	LGO
PVAL	2313	33	LGO
AVG	2346	21	LGO
MULT	2367	20	LGO

the address of the location generated for the variable I would be:

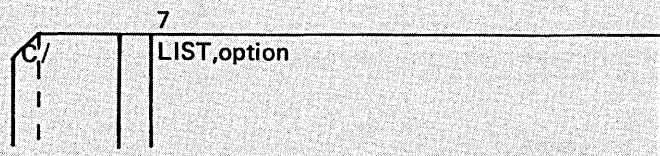
$$\begin{array}{r} 2346 \\ +20 \\ \hline 2366 \end{array}$$

## FORTRAN LISTING CONTROL

LIST directives permit a source program listing to be stopped or restarted at any line. They also provide control of the non-fatal diagnostic summary, reference map, and object code listing on a program unit basis. LIST directives can only suppress a listing that would otherwise appear; they cannot add a listing if it has not been selected on the FTN control card.

LIST directives have no effect if TS mode is selected.

The format of the LIST directive is:



C/ must appear in columns 1 and 2 with columns 3 through 6 blank

option NONE stops source program listing and can suppress the other listings

ALL resumes source program listing

LIST, option appears anywhere within columns 7 through 72. Leading, trailing, and embedded blanks are allowed; continuation is not permitted.

Lines that do not conform to the LIST directive format are processed as comment lines, assuming column 1 contains a C. No diagnostic is issued.

LIST directives can be placed anywhere in a source program or C\$ debug deck, except they must not interrupt statement continuation. A list statement within a continuation sequence causes a fatal-to-execution diagnostic to be issued.

LIST directives control only the listings for the program unit in which they appear.

LIST,NONE stops source program listing. The directive itself is listed but subsequent source lines, including additional LIST,NONE directives, will not be listed. However, when LIST,NONE is the first physical line of a program unit, neither it nor the page header is listed.

LIST,ALL resumes source program listing beginning with the directive itself. The listing will be restarted immediately regardless of the number of preceding LIST,NONE directives.

If LIST,NONE is active when an END statement is encountered, no reference map or object listing is output. No diagnostic summary appears unless the program unit contained fatal errors. If fatal source program errors were detected, output includes the incorrect statements and a complete diagnostic summary containing errors of all levels requested by the EL parameter on the FTN control card.

Partial maps, diagnostic summaries, and object listings cannot be obtained. The LIST directives have no effect on the process of accumulating information to be output; they only control whether to list or not to list information. A LIST directive cannot override a control card parameter that inhibits the accumulation of information for a listing.

Example:

If the R=3 (long reference map) control card option is chosen and LIST,NONE is active for 90 lines of the 150-line source program, 60 lines of the source program are listed but map information is accumulated for all 150 lines. The complete map is listed unless LIST,NONE is active when the END statement is encountered.

## DMPX.

When a program does not compile or execute successfully, a partial dump is produced. A DMPX includes the contents of the registers, the first 101 words of the user's field length (the communication region), and the contents of the 101 (octal) words immediately preceding and immediately following the addresses where the job terminated.

1. P Address of program step to be executed next if job had not terminated.
2. RA Reference address: absolute address where user's field begins. All other addresses are relative to this address.
3. FL Field length of job.
4. EM Default exit mode.
5. RE Extended core storage reference address.
6. FE Field length assigned to job in extended core storage.
7. MA Address used for linkage between the operating system and user program.
8. Address registers.
9. Contents of address registers.
10. Index registers.
11. Contents of index registers.
12. Operand registers.
13. Contents of operand registers.
14. Contents of locations specified in the A register. For example, items 8 and 9 show register A2 contains the address 002155, and item 14 shows location 002155 contains 1725 2420 2524 0000 0133.

15. Address of 60-bit word in central memory, followed by contents of that word (in octal).
16. Indicates that contents of previous locations are repeated up to but not including this location

DMPX.

		⑧	⑨	⑩	⑪
① P	013552	A0	002133	B0	000000
② RA	312100	A1	000001	B1	000001
③ FL	065000	A2	002155	B2	000001
④ EM	070000	A3	002140	B3	000040
⑤ RE	000000	A4	004474	B4	000130
⑥ FE	000000	A5	002135	B5	000001
⑦ MA	001400	A6	000001	B6	004636
⑫		A7	002140	B7	002206
X0	7777	7777	7777	7777	7776
X1	0000	0000	0000	0000	0000
X2	0000	0000	0000	0000	4776
X3	0000	0216	5000	0000	0004
X4	0000	0000	0000	0000	0005
X5	0000	0000	0000	0000	0002
X6	0102	2400	0000	0000	0000
X7	0000	0000	0100	0000	2165

	⑬					⑭					
C(A1)=	0000	0000	0000	0000	0000	C(B1)=	0000	0000	0000	0000	0000
C(A2)=	1725	2420	2524	0000	0133	C(B2)=	0000	0000	0000	0000	0000
C(A3)=	0000	0000	0100	0000	2165	C(B3)=	0000	0000	0000	0000	0000
C(A4)=	0000	0000	0000	0000	0004	C(B4)=	0000	0000	0000	0000	0000
C(A5)=	0000	0000	0240	4000	0000	C(B5)=	0000	0000	0000	0000	0000
C(A6)=	0000	0000	0000	0000	0000	C(B6)=	5140	0044	7404	0000	4613
C(A7)=	0000	0000	0100	0000	2165	C(B7)=	0000	0000	0000	0000	0000

	⑮				
00000	00000	00000	00000	00000	00000
00004	00000	00000	00000	00000	00000
00010	32323	23232	17200	00354	
00014	00000	00000	00000	00000	
00020	01022	14000	00000	05152	
00024	000000	00000	00000	00015	
00037	000000	00000	00000	00007	
00040	00000	00000	00000	00000	
00044	00000	00000	00000	00000	
00053	00000	00000	00000	00000	
00054	51100	00001	03110	00054	
00060	56124	63310	13415	21422	
00064	14071	70000	00000	00000	
00070	00000	00000	00000	00000	
00100	11162	02524	00000	00000	

00002	11162	02524	00000	00100
32323	23232	22140	00322	
00000	00000	00000	00000	
00017	23312	31726	14000	00001
03141	72305	35530	00000	
00000	00000	00000	00000	
00043	00000	00000	00000	50106
00046	55555	55555	55555	55542
64550	02500	00000	46000	
61447	77776	03040	00056	
00000	00000	00000	13607	

17252	42025	24000	02133	
03171	52023	00000	00305	
00000	00000	00000	50167	
00000	00000	00000	00000	
00032	00000	00000	50064	
00000	00000	00000	00000	
00052	00000	00000	00000	00002
00000	00000	00000	00000	
07040	00060	51600	00001	
40000	00000	00000	00100	
00000	00000	00000	00000	
13443	03040	00060	67402	
00000	00000	00000	00021	
00000	00000	40000	00000	

13452	50100	00021	20101	46000
13454	43744	20130	15117	37341
13460	03220	13465	66340	46000
13464	63510	02500	11034	46000
13470	50100	00020	43770	20106
13474	11771	12774	03345	13474
13500	15117	63510	50100	00006
13504	36441	71600	00142	46000
13510	50100	00015	43752	11771
13514	63440	61600	13530	37124
13520	20744	54710	63420	46000
13524	02500	11034	61000	46000
13530	64330	50100	00020	43770
13534	11771	74150	12771	46000
13540	03315	13540	20766	54710
13544	50100	00015	76710	20772
13550	04000	13563	00000	00000
13554	54610	04000	13551	46000
13560	13661	13161	13661	46000
13564	51100	00001	03110	13564
13570	20150	36661	01000	13552
13574	03010	13571	51100	00001
13600	01000	13552	61000	46000
13604	20636	51600	13606	74660
13610	99000	00000	00000	00000

03210	13456	50100	00002	
03330	13456	10411	46000	
50100	00020	43770	20106	
27704	51300	07654	40773	
15117	36441	72147	77765	
20766	54710	04000	10710	
43744	20130	15117	46000	
50100	00015	43701	12771	
12773	03335	13511	54710	
03210	12246	37442	46000	
61600	13522	04000	12246	
26454	56330	63340	46000	
20106	11771	12774	46000	
03315	13535	20744	54710	
50100	00015	03210	10710	
12771	54710	50100	00015	
01300	00000	00000	00000	
51100	00066	03310	13557	
51600	13551	10611	46000	
04004	13565	61000	46000	
04000	07354	00000	00000	
03110	13573	71100	00001	
04004	13601	61000	46000	
36116	20123	04000	13577	
13652	00000	00000	00000	

50100	00015	20101	46000	
11771	20766	54710	14422	
26707	36377	63373	37443	
03310	13473	10411	66331	
50500	00015	20530	73550	
50300	00015	73330	37114	
54710	46000	61000	46000	
56330	50100	00020	43770	
50100	00020	43752	20130	
64330	27444	61600	13525	
50400	00020	20430	73440	
03345	13532	20766	54710	
50100	00017	43770	20106	
54301	43744	20330	15337	
76710	20770	15717	54710	
04000	04474	00000	00000	
51100	13550	04000	13560	
51100	00001	01000	13550	
51100	00001	03110	13565	
71602	20314	20652	36662	
04000	13570	61000	46000	
20622	12161	73610	20123	
00000	00000	00000	00000	
03210	13476	37224	48000	
61600	13456	50100	00021	
20302	37443	03040	10710	
50100	00020	43770	20106	
50100	00017	43770	20106	
53550	37113	03210	13507	
03040	10710	36334	46000	
20106	15117	63310	46000	
11771	12774	03345	13517	
50100	00021	63510	10577	
04000	13514	61000	46000	
50100	00015	43752	20130	
11771	76140	12771	46000	
50100	00015	73110	37431	
61600	13456	04000	11052	
51100	00001	03110	13553	
71100	00130	20160	46000	
20652	01000	13552	46000	
71602	20314	04000	13563	
53160	20173	03310	13571	
71603	24616	12661	20651	
03210	13577	20151	13116	
71603	24616	12661	20651	





## § 7600 Load Map

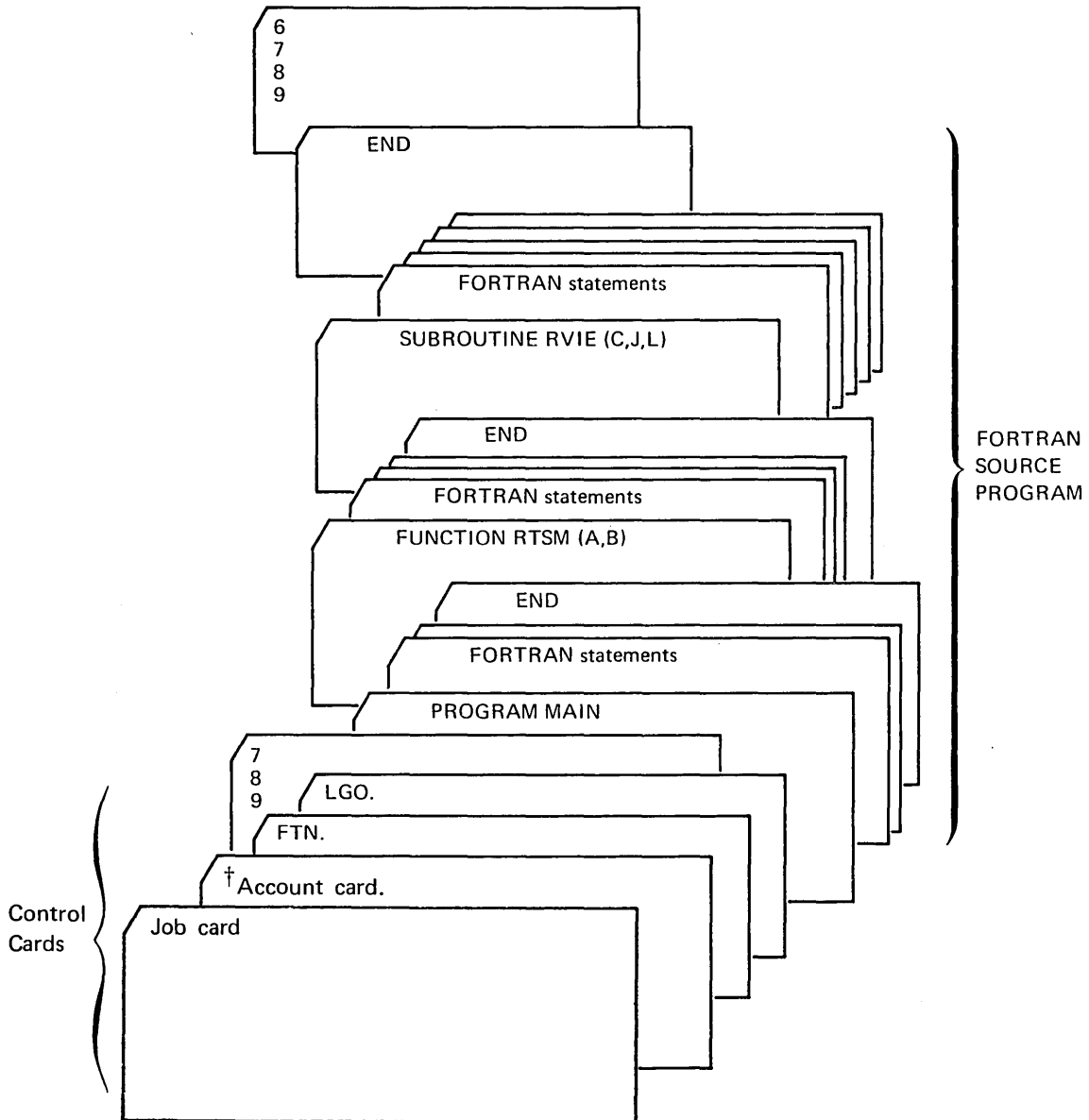
SCOPE	2	LOAD	MAP	LOADER VERSION 1.0 02/27/75 17.22.56 PAGE 2									
STOP.		1162	VARDIM2	143									
ARNORM.		1171											
SYSARG=		1205											
IOERR.		1224	NAMOUT=	2213									
SYSEND.		1247											
SYP=5		1250											
CLSLNK.		1264											
SYSERR.		1317	GETFIT=	1762									
			NAMOUT=	2210									
SYP=1		1361											
SYP=2		1366											
SYP=3		1435											
SYP=4		1443											
SYS2=		1516											
CDD.		1554											
CMD.		1563											
COD.		1570											
RFN.		1577	NAMOUT=	2053	2063								
FECOPE.		1604	NAMOUT=	2030									
LINLIM.		1634	NAMOUT=	2204									
MSGAD.		1653											
DBGFIT.		1664											
GETFIT=													
GETFIT.		1735	NAMOUT=	2007									
NAME.		1773											
NAMOUT=													
NAMOUT.		2003	VARDIM2	142									
OUTCOM=													
FEOL.		2265	NAMOUT=	2141									
FEOL.		2272	NAMOUT=	2142									
FE0XFL.		2340	FLTOUT=	615	1007								
FE0AFM.		2346	FLTOUT=	641	644	646	652	1005					
			NAMOUT=	2153									
FEORLS.		2353	FLTOUT=	575	576	577	601	773	774	1001	1003		
FEOCNV.		2366	FLTOUT=	613									
FECCHR.		2432	NAMOUT=	2222									
FEORIF.		2443	FLTOUT=	770									
FEORIO.		2447	NAMOUT=	2106	2130	2151	2163	2166	2172				
FEONTL.		2454	NAMOUT=	2156									
SYSAID=													
SYSAID=		2467	08.IO.	414									

§ Applies only to SCOPE 2.1.



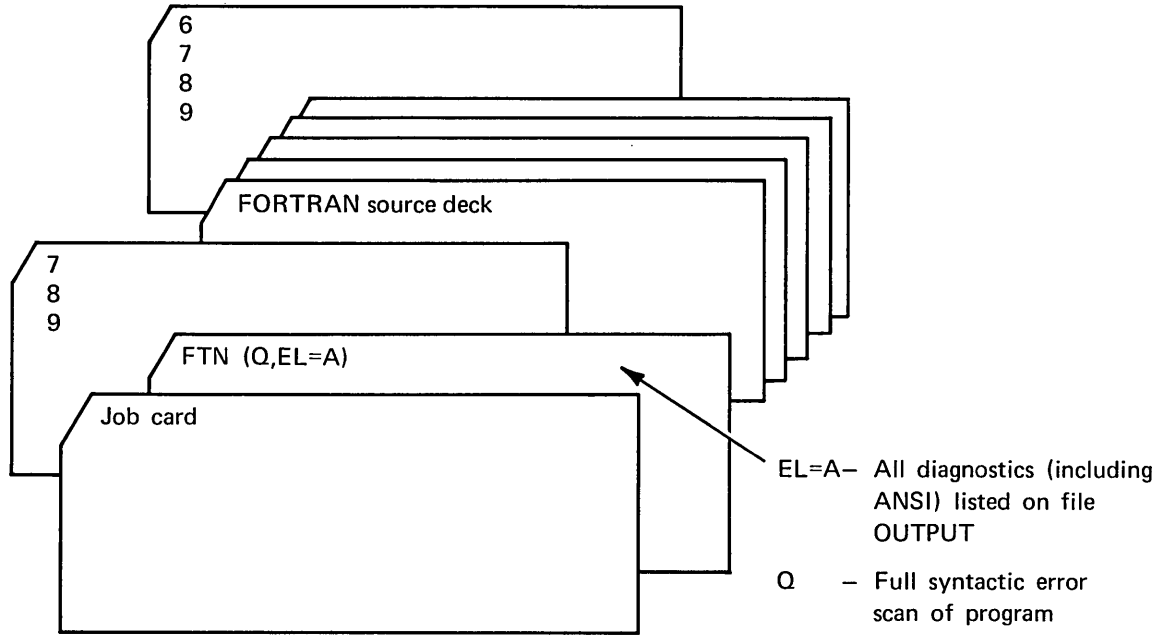
**FORTRAN SOURCE PROGRAM WITH CONTROL CARDS**

Refer to the operating system reference manual for details of control cards.

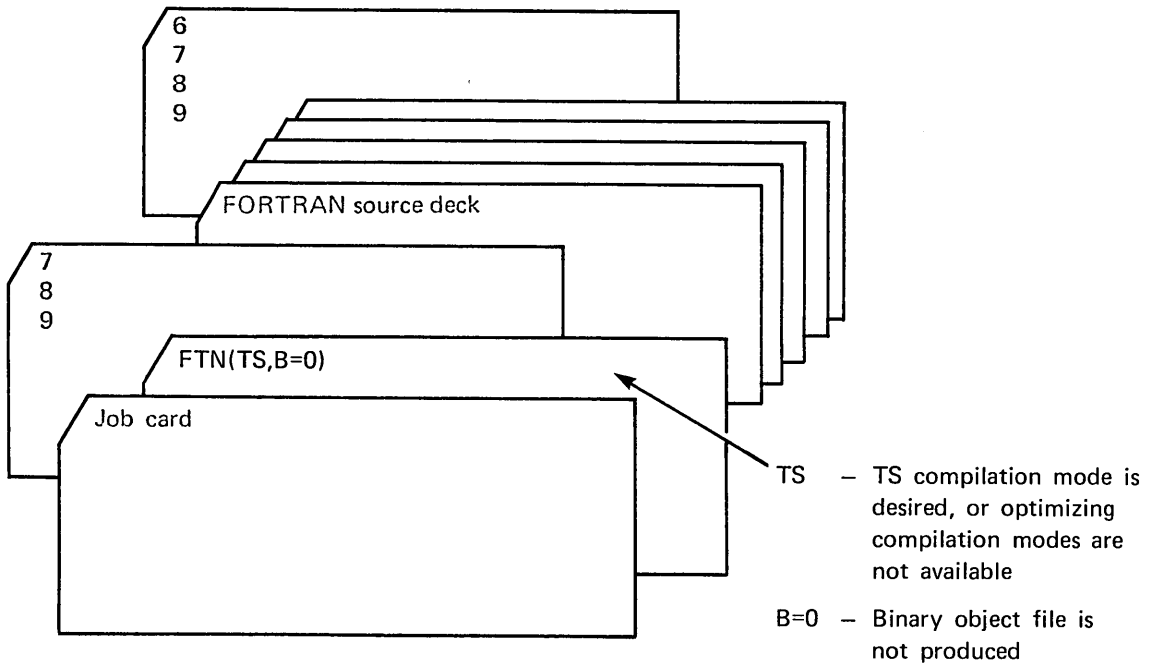


† Account card follows the job card in KRONOS and should be in all KRONOS decks.

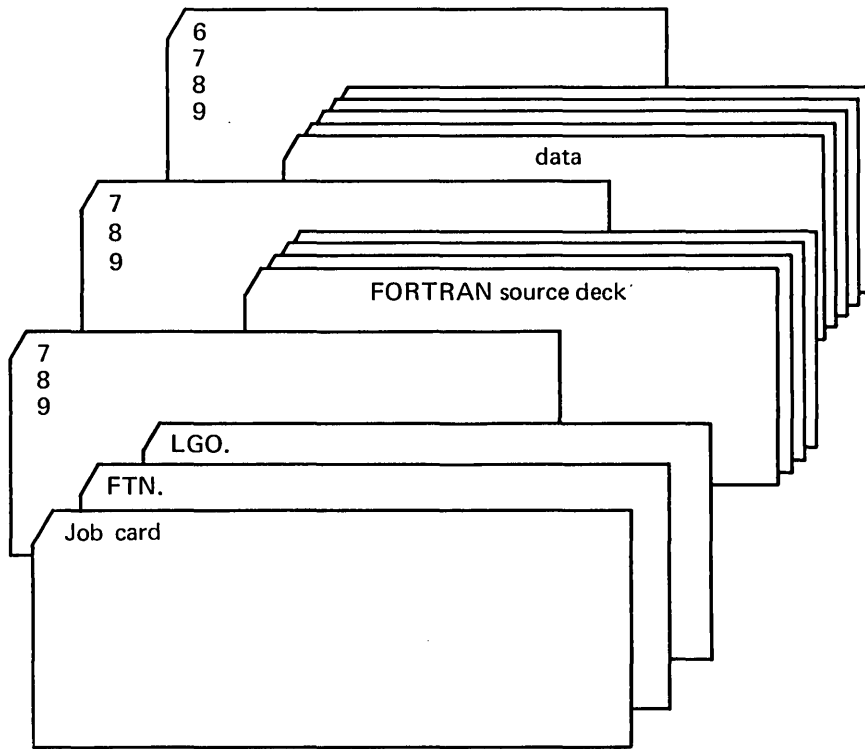
## COMPILATION ONLY



## TS MODE COMPILATION ONLY

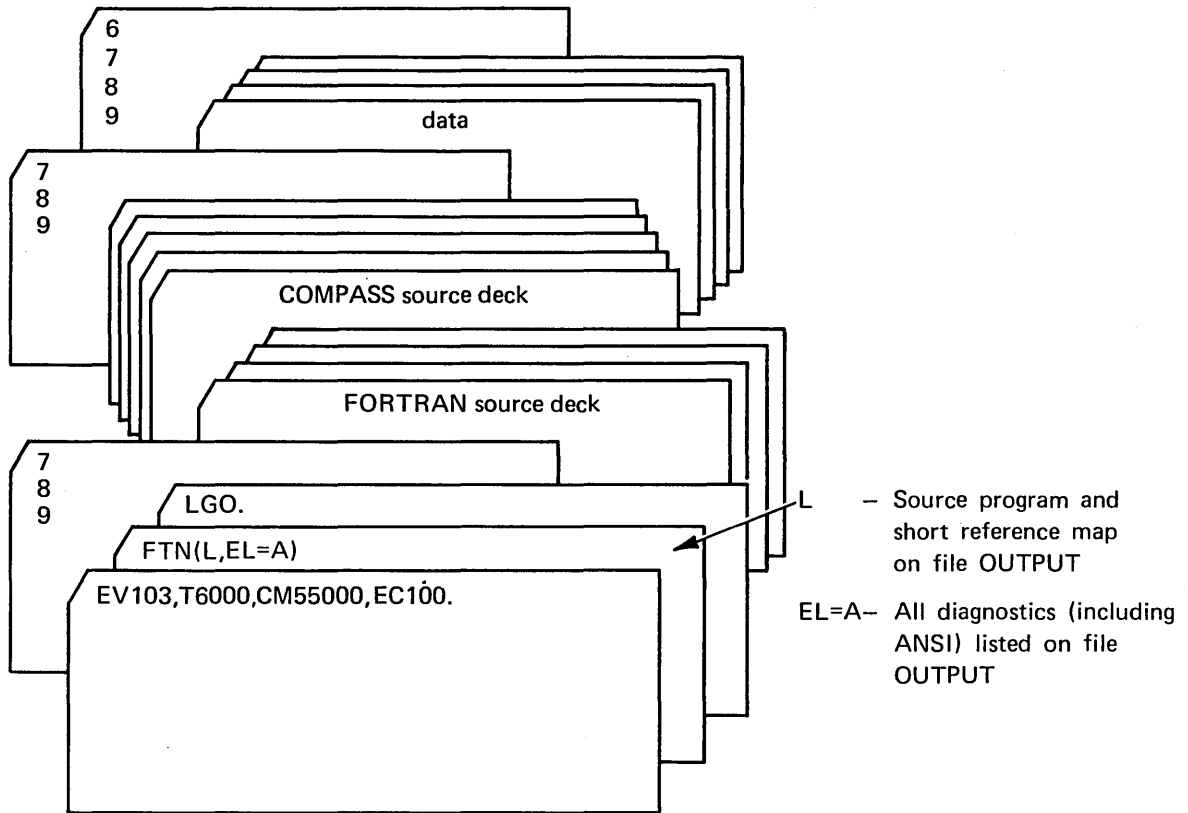


# COMPILATION AND EXECUTION

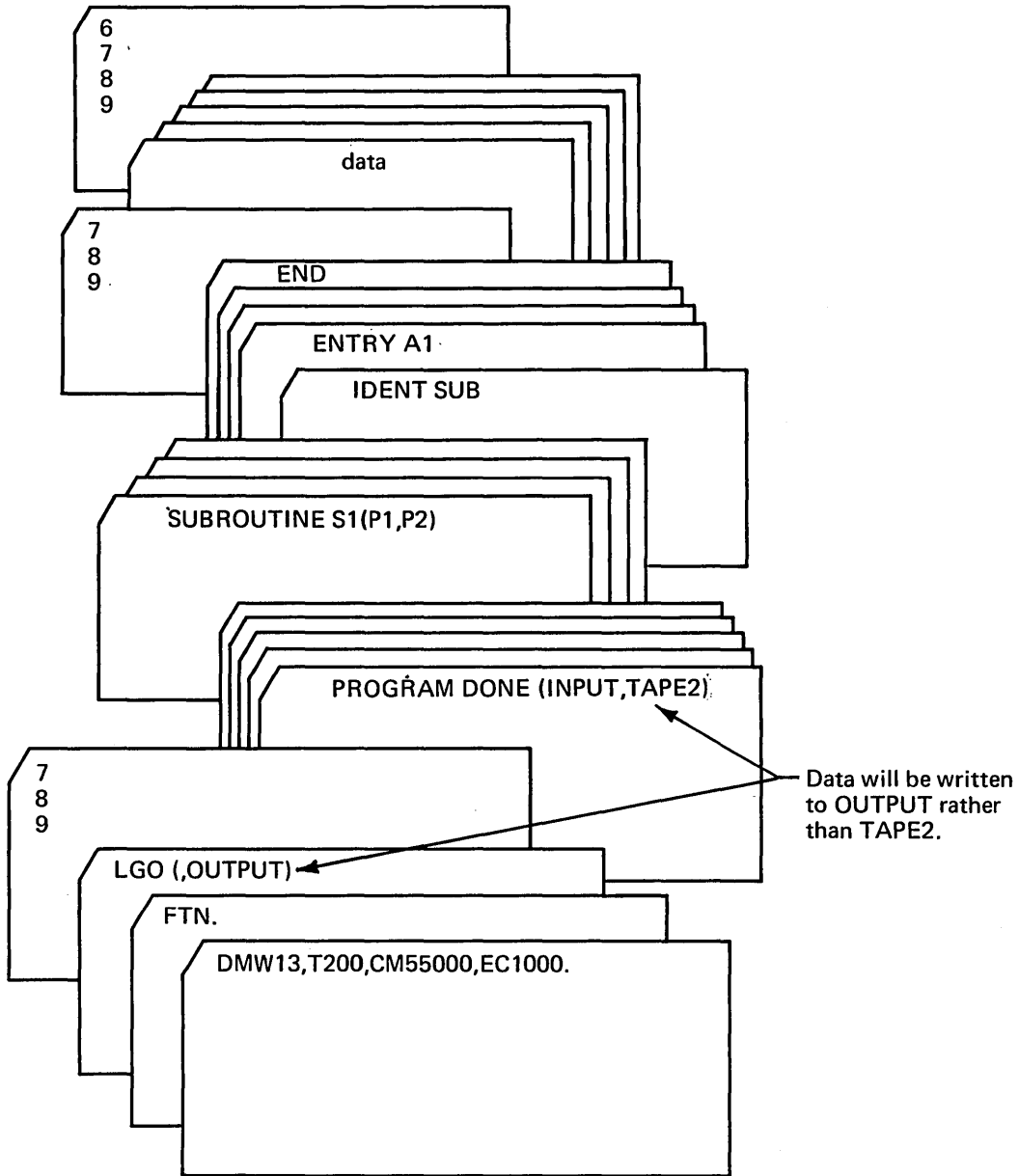


## FORTRAN COMPILATION WITH COMPASS ASSEMBLY AND EXECUTION

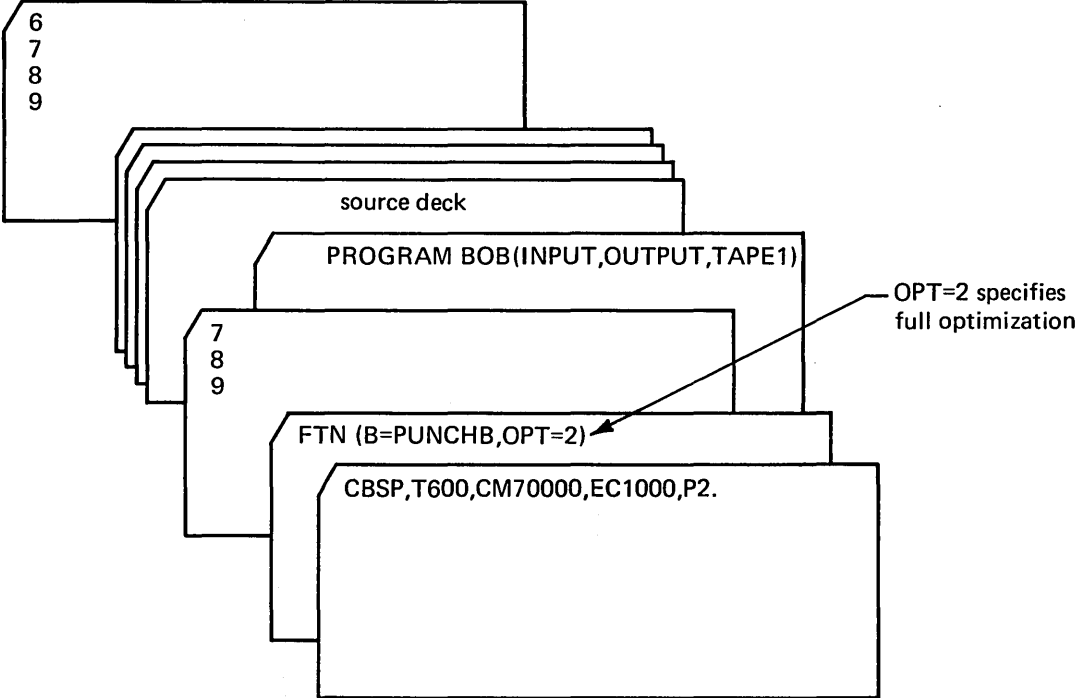
FORTRAN and COMPASS program unit source decks can be in any order. COMPASS source decks must begin with a card containing the word IDENTb in columns 11-16 and terminate with a card containing the word ENDb in columns 11-14 (b denotes a blank). Columns 1-10 of the IDENT and END cards must be blank.



# COMPILE AND EXECUTE WITH FORTRAN SUBROUTINE AND COMPASS SUBPROGRAM

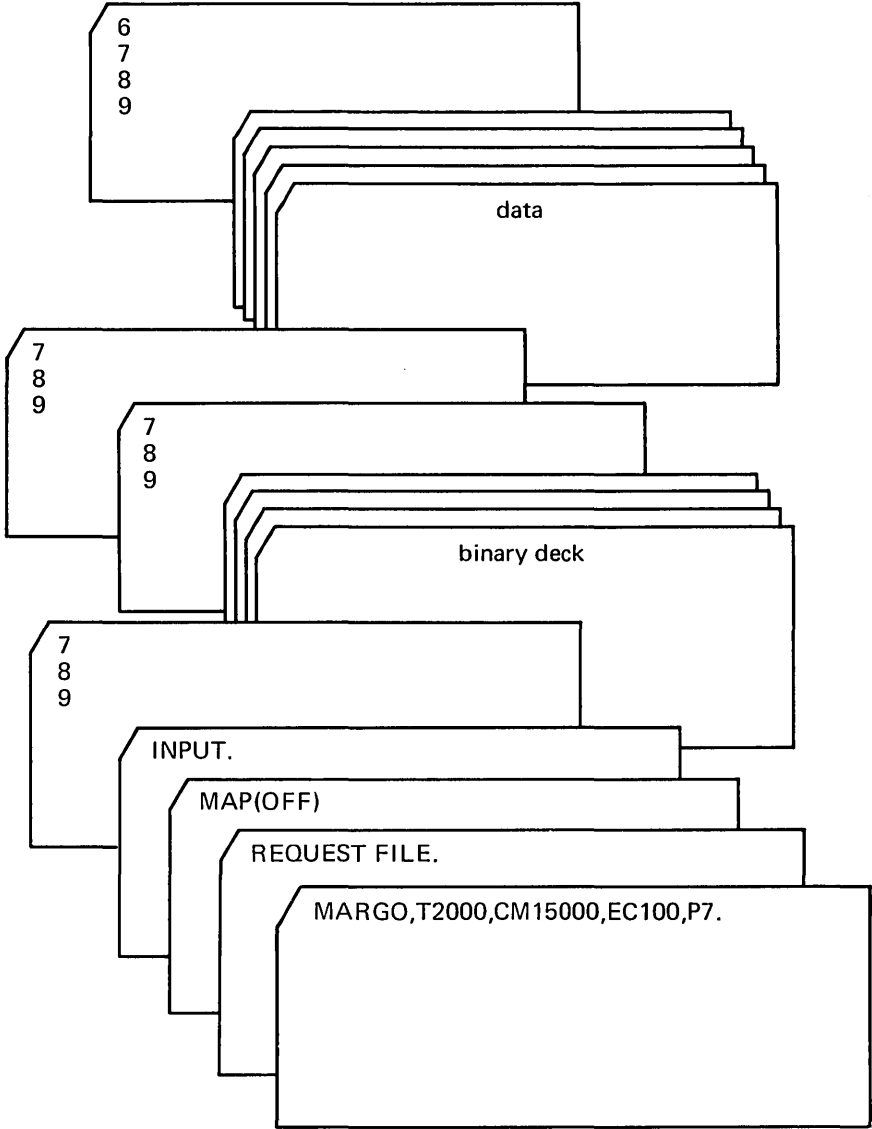


# COMPILE AND PRODUCE BINARY CARDS

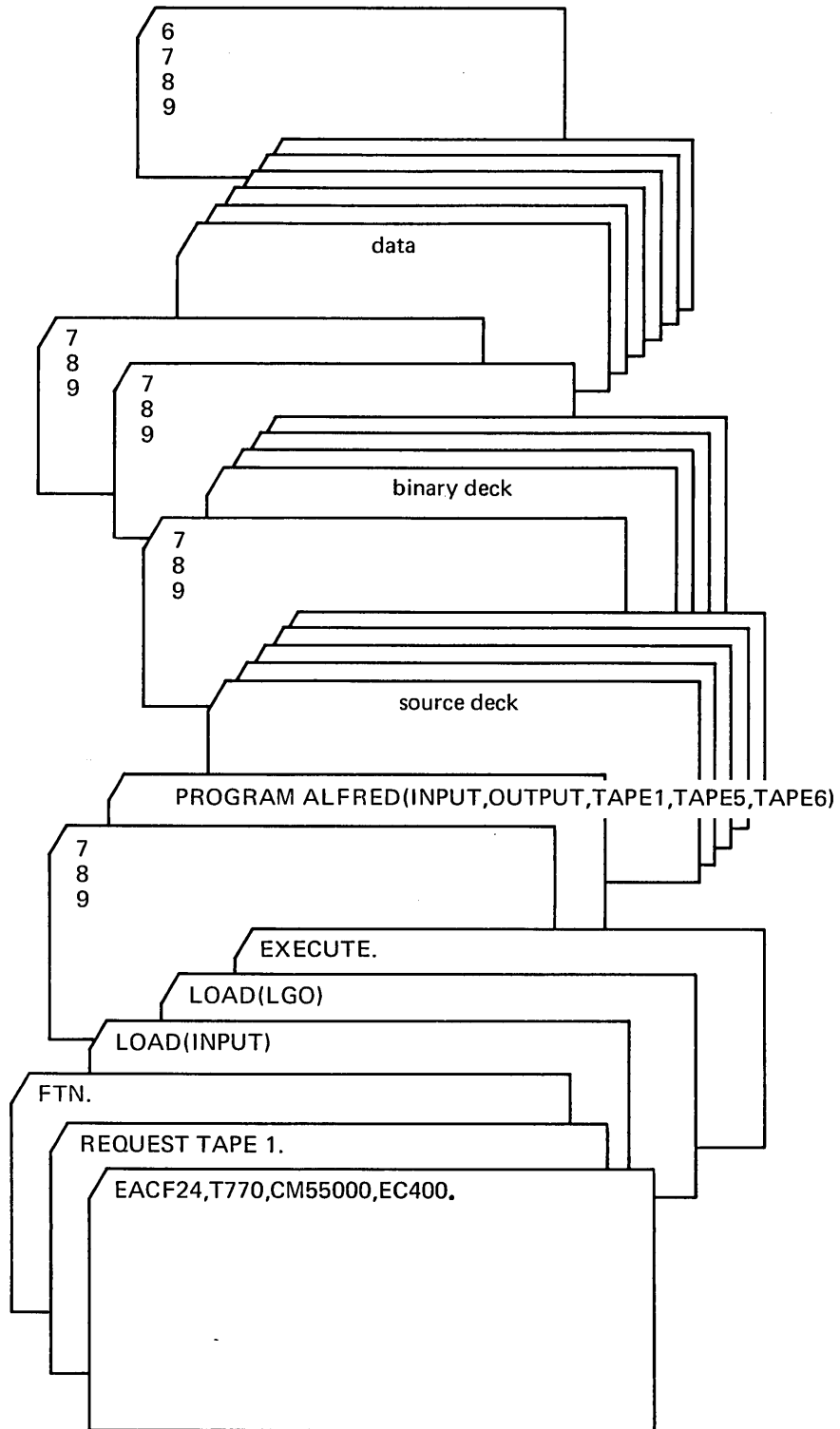




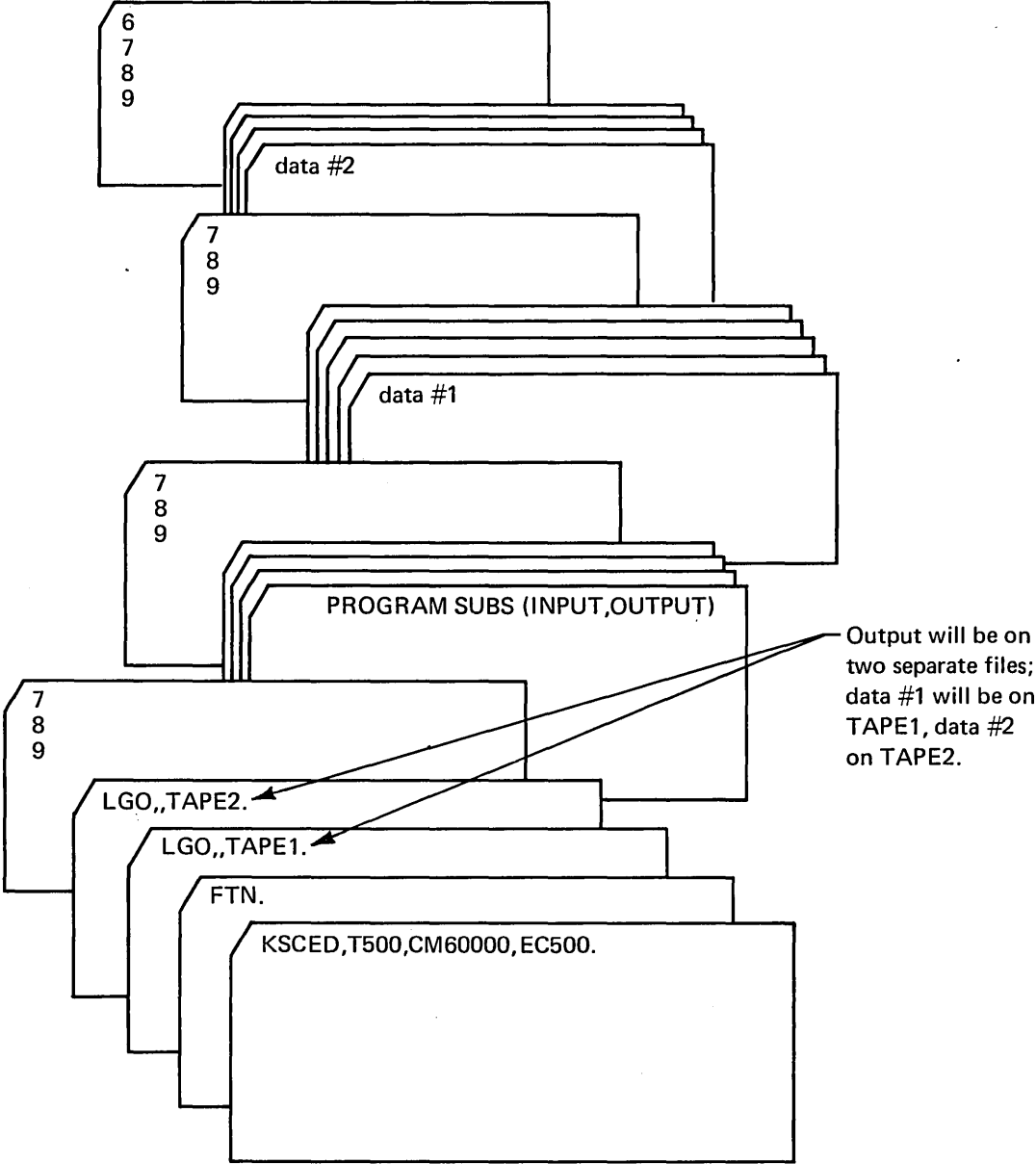
# LOAD AND EXECUTE BINARY PROGRAM



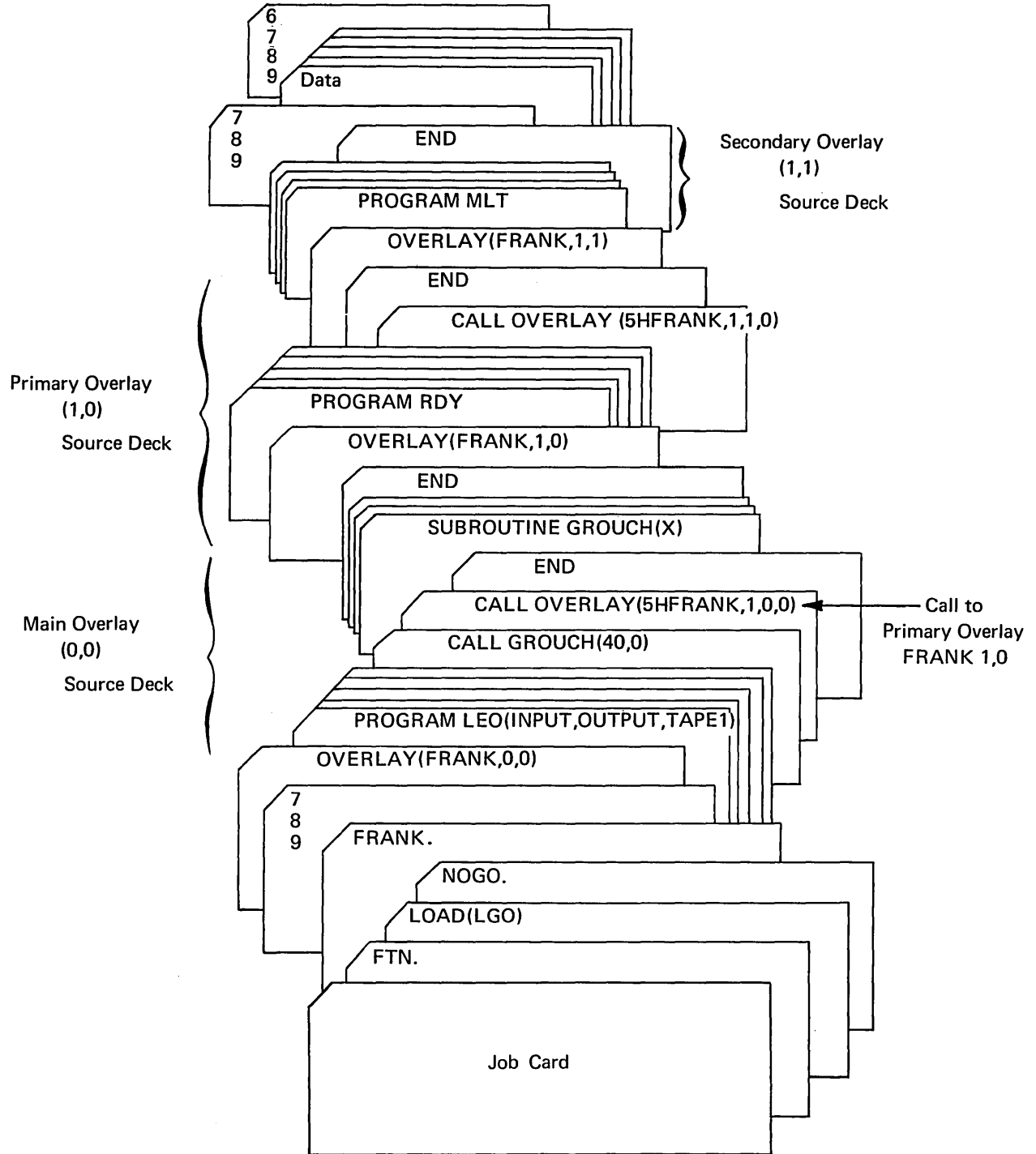
# COMPILE AND EXECUTE WITH RELOCATABLE BINARY DECK



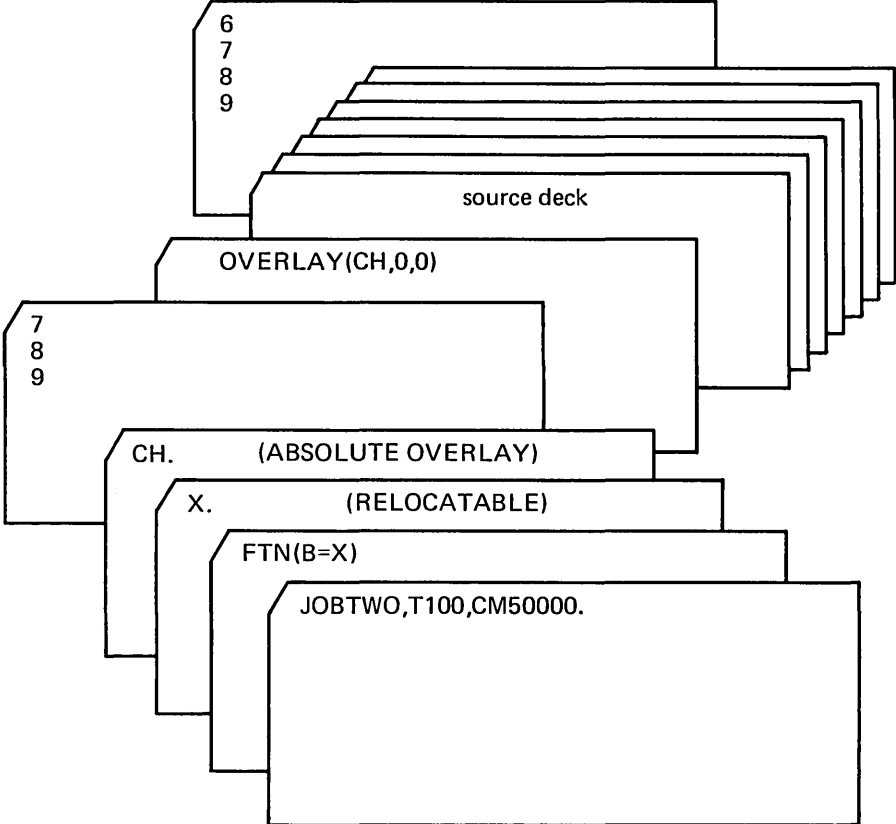
**COMPILE ONCE AND EXECUTE WITH DIFFERENT DATA DECKS**



# PREPARATION OF OVERLAYS



**COMPILATION AND 2 EXECUTIONS WITH OVERLAYS**





FORTRAN Extended provides several alternative modes for compilation. Their characteristics, together with the FTN control card parameter required to activate them, are as follows:

- Q           Fastest compilation; compiler performs full syntactic scan of source code, but produces no object code. Minimum field length required for compilation approximates that of OPT=0. OPT=0, OPT=1, and OPT=2 are ignored if specified. Expedient for finding errors in a program before attempting to execute it.
- TS          Very fast, one-pass compilation. Little optimization of object code; execution time approximates that of OPT=0. Minimum field length<sup>†</sup> for compilation is 40000<sup>‡</sup> or, 35000<sup>§</sup>. Expedient for a program which is recompiled before each execution, unless execution time is over twice as large as compilation time. For more information regarding TS mode, see section III-15.
- OPT=0      Fast, two-pass compilation; little optimization of object code. Most programs can be compiled in the minimum field length of 46000<sup>‡</sup> or 43000<sup>§</sup>.
- OPT=1      Two-pass compilation; moderate optimization of object code. Most programs can be compiled in the minimum field length of 46000<sup>‡</sup> or 43000<sup>§</sup>. Expedient for programs which are recompiled before each execution but require excessive execution time in TS mode.
- OPT=2      Relatively slow, two-pass compilation; extensive optimization of object code; fastest execution. Minimum field length required for compilation is 54000<sup>‡</sup> or 51000<sup>§</sup>. Programs in which the longest program unit consists of less than about 600 statements can be compiled in a field length of 60000; above that, field length required for compilation is proportional to the number of executable statements in, and the complexity of, the longest program unit. This optimization level is expedient for programs whose code is executed many times per compilation; it should not be used for undebugged programs since code redistribution in optimization renders debugging difficult if the executing program terminates abnormally.
- D           Activates FORTRAN Extended debugging facility (see section I-13). Minimum field length required for compilation is 61000<sup>‡</sup> or 56000<sup>§</sup>. Automatically activates OPT=0; OPT=1 and OPT=2 are ignored if specified. Necessary for programs wherein execution-time debugging is desired.
- UO          Provides additional potentially unsafe object code optimization when both the OPT=2 and UO options are specified. Prefetches indexed array references in small loops unconditionally and preserves the values in certain B registers across basic external function calls.

---

<sup>†</sup> Field lengths are given in octal.

<sup>‡</sup> Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

<sup>§</sup> Applies only to SCOPE 2.1.

## OBJECT CODE OPTIMIZATION

### OPT=0

In the OPT=0 compilation mode, compile time evaluations are made of constant subexpressions, redundant instructions and expressions within a statement are eliminated, and PERT critical path scheduling is done to utilize the multiple functional units efficiently.

### OPT=1

In the OPT=1 compilation mode, the following optimizations are effected in addition to those in OPT=0:

1. Redundant instructions and expressions within a sequence of statements are eliminated.
2. Subscript calculations are simplified, and values of simple integer variables are stored in machine registers throughout loop execution, for innermost loops satisfying all of the following conditions:

No entries other than by normal entry at the beginning of the loop.

No exits other than by normal termination at the end of the loop.

No external references (user function references or subroutine calls; input/output, STOP, or PAUSE statements) in the loop.

No IF or GOTO statement in the loop branching backward to a labeled statement appearing previously in the loop.

### OPT=2

In the OPT=2 compilation mode, the compiler collects information about the program unit as a whole and the following optimizations are attempted in addition to those in both OPT=0 and OPT=1:

1. Values of simple variables are not retained when they are not referenced by succeeding statements.
2. Invariant (loop-independent) subexpressions are evaluated prior to entering the loops containing them.
3. For all loops, the evaluation of subscript expressions containing a recursively defined integer variable (such as  $I=I+1$ ) is reduced from multiplications to additions.
4. Array addresses, values of simple variables in central memory, and subscript expressions are stored in machine registers throughout loop execution for all loops.
5. In all loops and in complicated sections of straight-line code, array references and subscript values are stored in machine registers.
6. In small loops, indexed array references are prefetched after safety checks are made to ensure that the base address of the array and its increment are reasonable and should not cause an out-of-bounds reference (mode 1 error).



## UO

In unsafe optimization mode, the optimizations listed below are made, in addition to the optimizations made under OPT=2, since OPT=2 must also be selected. If OPT=2 is omitted, UO is not invoked.

1. In small loops, indexed array references are prefetched unconditionally without any safety checks.

Example:

```
REAL B(100,100)
DO 20 I = 1,100,10
20 S = S + B(J,I)
```

When the compiler prefetches the reference to B, the last reference to B in the loop is B(J,110) which might cause an out-of-bounds error at execution time.

2. When a math library function is referenced, the compiler assumes that the contents of certain B registers are preserved for use following the function processing.

Example:

```
REAL A(10),B(10)
DO 10 I = 1,N
10 B(J) = EXP(A(I))
```

The compiler assigns I and N to B registers during the loop.

In a loop, the registers available for assignment are determined by the presence or absence of external references. External references are user functions and subroutines, calls, I/O statements, and FORTRAN math library functions (SIN, COS, SORT, EXP, and so on).

When UO is not selected, the compiler assumes that any external reference modifies all the registers; therefore it does not expect any register contents to be preserved across function calls.

If a math library other than the FORTRAN Common Library is used at an installation, the B register portion of the UO option can be deactivated by reinstallation of the compiler.

## SOURCE CODE OPTIMIZATION

To achieve maximum object code optimization regardless of optimization level, the user should observe the following practices for programming source code:

1. Since arrays are stored in column major order, DO loops (including implied DO loops in input/output lists) which manipulate multi-dimensional arrays should be nested so that the range of the DO loop indexing over the first subscript is executed first.

Example:

```
DIMENSION A(20,30,40), B(20,30,40)
      .
      .
      .
      DO 10 K = 1, 40
      DO 10 J = 1, 30
      DO 10 I = 1, 20
10 A(I,J,K) = B(I,J,K)
```

2. The number of different variable names in subscript expressions should be minimized.

Example:

```
X = A(I+1,I-1) + A(I-1,I+1)
```

is more efficient than:

```
IP1 = I+1
IM1 = I-1
X = A(IP1,IM1) + A(IM1,IP1)
```

3. The use of EQUIVALENCE statements should be avoided, especially those including simple variables and arrays in the same equivalence class.
4. Common blocks should not be used as a scratch storage area for simple variables.
5. Program logic should be kept simple and straightforward; program unit length should be less than about 600 executable statements.
6. The use of dummy arguments (formal parameters) and variable dimensions should be avoided if possible; common or local variables should be used instead.
7. The first n-1 dimensions of an n-dimensional array should be either a power of 2 or the sum or difference of two powers of 2.

8. Common expressions should be grouped so that they can be recognized for optimization.

Example:

```
AA = X*A/Y  
BB = X*B/Y
```

is less efficient than

```
AA = A*(X/Y)  
BB = B*(X/Y)
```

Likewise, invariant and constant expressions should be grouped appropriately.

Example:

```
DO 10 I = 1, 50  
10 B(I) = 1. + A(I) + X
```

is less efficient than

```
DO 10 I = 1, 50  
10 B(I) = (1. + X) + A(I)
```

Example:

```
X = 1024. * B * 3.14159
```

is less efficient than

```
X = (1024. * 3.14159) * B
```

9. Multiple references to a basic external function within a statement should be algebraically reduced to a single reference.



---

When the TS option is specified on the control card, FTN operates in Time-sharing (TS) mode. Compilation is one-pass; therefore, no overlay reloading is required to compile multiple program units, and the number of disk accesses is reduced. The minimum compilation field length is 40000 octal. The CPU time spent in compilation is 30% to 75% less than that for optimizing mode (OPT=0, OPT=1, or OPT=2). The object code is not highly optimized and thus executes approximately at the rate of that produced by OPT=0.

Time-sharing mode is permissive in that it accepts some keyword misspellings and punctuation errors. When this occurs, a warning level diagnostic is issued, since the program may not compile under two-pass mode.

Misspelled keywords will be recognized if the string length matches the keyword length, the first four characters match, and the context is unambiguous.

For example,

```
COMMUN A(2)
```

will be recognized as a COMMON declaration and a warning diagnostic will be issued. However,

```
COMMUNC(I) = 2+I
```

will be correctly interpreted as a replacement statement or a statement function definition, depending on whether or not COMMUNC was previously dimensioned.

Some punctuation errors which do not inhibit the compiler from correctly interpreting a statement will be accepted.

For example, in

```
DO 10, I = 1,10
```

the first comma will be diagnosed and ignored.

## SOURCE LISTING FORMAT

In TS mode, certain listing differences occur. The main source listing differences are as follows:

Source line numbers are not printed on every fifth line as in other modes. Instead, a code address appears before each line.

When the OL option is selected, generated object code is listed interspersed with the source code. Only the mnemonic instruction is listed, not the octal equivalent.

Diagnostics are listed on the output file immediately after the statement which caused them.

For a description of the cross reference map in TS mode, see section III-1. For a description of the compile-time diagnostics in TS mode, see section III-2.

## SEQUENCED LINE FORMAT

When TS mode is selected for program compilation, a FORTRAN Extended program may be coded in sequenced line format as well as in the standard format described in section I-1. If the source code is in sequenced line format, the option SEQ should be specified on the FTN control card.

The format for sequenced line coding is as follows:

seqnum d sl stat

seqnum    Sequence number consisting of 1-5 digits, assigned in ascending order

d            blank            First line of a statement

          +            Continuation line

          Any other    Comment line  
          Character

sl            Optional statement label consisting of 1-5 digits; must be followed by a blank

stat          FORTRAN source statement; may begin anywhere after d and continue through column  
          80

### Example:

```
00100 PROGRAM XYZ (OUTPUT)
00110C COMPUTE AREA
00120 DIMENSION A(100), B(100),
00130+ C(200)
00140 10 CALL SUB(A,B,C,100)
00150 STOP
00160 END
```

FORTRAN Extended provides the capability for processing data records under the Sort/Merge system from within a FORTRAN program. The FORTRAN user of this feature should be familiar with the autonomous functioning of the Sort/Merge system as described in the Sort/Merge Reference Manual. A job containing a FORTRAN program interfacing with Sort/Merge must contain an RFL card (SCOPE 3.4 and SCOPE 2.1) or a REDUCE(-) card (KRONOS 2.1 and NOS 1.0) to reserve field length and a LIBRARY(COBOL) or a LDSET (LIB=COBOL) card before the LGO card.

The FORTRAN subroutines interfacing with Sort/Merge are listed below under the corresponding Sort/Merge macro or directive. The series of calls to Sort/Merge subroutines must begin with a call to SMSORT, SMSORTB, SMSORTP, or SMMERGE.

**SORT**

**CALL SMSORT (mrl,ba)**

mrl Maximum length in characters of records to be sorted.

ba§ LCM buffer area in decimal for intermediate scratch files constructed by Sort/Merge.

SMSORT calls for a sort on rotating mass storage.

**SORTB**

**CALL SMSORTB (mrl)‡**

mrl Maximum length in characters of records to be sorted.

SMSORTB calls for a balanced tape sort. SMTAPE (see below) must also be called.

**SORTP**

**CALL SMSORTP (mrl)‡**

mrl Maximum length in characters of records to be sorted.

SMSORTP calls for a polyphase tape sort. SMTAPE must also be called.

‡Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§Applies only to SCOPE 2.1.

## MERGE

### CALL SMMERGE (mrl,ba)

mrl     Maximum length in characters of records to be merged.

ba§     LCM buffer area in decimal for intermediate scratch files constructed by Sort/Merge.

SMMERGE calls for merge-only processing.

## FILE

### CALL SMFILE (dis,i/o,lfm,action)

dis     File disposition:

≠SORT≠	File to be sorted.
≠MERGE≠	File to be merged.
≠OUTPUT≠	File to receive output.

i/o     Mode of file input/output:

≠FORMATTED≠	} File accessed with formatted input/output.
≠CODED≠	
≠BINARY≠	File accessed with unformatted input/output.
0 ‡	File accessed with interfacing Record Manager subroutines.

lfm     Logical file name:

u	Logical unit number, 1 to 99.
nLfilename	File name left justified with zero fill.
fit ‡	When i/o is specified as 0, an array containing the file information table.

action   File disposition following sort or merge:

≠REWIND≠
≠UNLOAD≠
≠NONE≠ (default)

SMFILE must be called for each file to be sorted or merged, and once for the file to receive the output (unless SMOWN is called). If a file is to be accessed with formatted or unformatted FORTRAN input/output, its name must be declared in the PROGRAM statement. ‡If a file is to be accessed with Record Manager subroutines, OPENM should be called prior to SMFILE. Files should be properly positioned before they are sorted or merged.

‡Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§Applies only to SCOPE 2.1.



## KEY

### CALL SMKEY (charpos,bitpos,nchar,nbits,code,colseq,order)

charpos	Integer specifying position of first character of sort key, considering the first character as position number 1.
bitpos	Integer (usually 1) specifying position of first bit of sort key in character (or 6-bit byte) specified by charpos, considering the first bit as position number 1.
nchar	Integer specifying number of characters or complete 6-bit byte in sort key.
nbits	Integer (usually 0) specifying number of bits in sort key in excess of those indicated by nchar.
code	Coding identifier: ≠ DISPLAY≠           Internal display code. ≠ FLOAT≠            Floating point data. ≠ INTEGER≠          Signed integer data. ≠ LOGICAL≠          Unsigned integer data (default). ≠ SIGN≠             Numeric data in display code; sign represented by overpunch on low order character of sort key.
colseq	Collating sequence (applicable only if code is specified as ≠DISPLAY≠): ≠ ASCII6≠           6-bit ASCII collating sequence (default for installations using ASCII 64-character set). ≠ COBOL6≠           6-bit COBOL collating sequence (default for installations using ASCII or CDC 63-character set or CDC 64-character set). ≠ DISPLAY≠          Internal display collating sequence. ≠ INTBCD≠           Internal BCD collating sequence. seqname             Name of a collating sequence specified in a call to SMSEQ (see below).
order	Order of sort processing: ≠ A≠                 Ascending (default). ≠ D≠                 Descending.

One SMKEY call is required to describe each sort key to be used. The first SMKEY call indicates the major key; subsequent calls indicate additional or minor keys in the order encountered.

## SEQUENCE

### CALL SMSEQ (seqname,seqspec)

seqname	Name of user supplied collating sequence.
seqspec	Name of integer array, terminated with a negative number, containing entire sequence of characters in order of collation.

SMSEQ specifies a user's collating sequence, or redefines the default to be a user collating sequence or a standard collating sequence other than the system default.

The characters in seqspec can be specified as their octal equivalents in the form ijB or as Hollerith constants in the form 1Rx. Characters to collate equal are specified in a call to SMEQU (see below). Unspecified characters collate high (following the last character specified in seqspec) and equal.

## EQUATE

### CALL SMEQU (colseq,equespec)

colseq Collating sequence determined by a previous call to SMKEY (and perhaps SMSEQ).

equespec Name of an integer array, terminated with a negative number, containing characters to collate equal to the last character, which must be included in colseq.

SMEQU specifies that two or more characters in the collating sequence are equal for comparison purposes.

## OPTIONS

### CALL SMOPT (optlist)

optlist Non-ordered series of options as follows:

≠VERIFY≠	Check output for correct sequencing (important for insertions during output and merge input).
≠RETAIN≠	Retain records with identical sort keys in order of appearance on input file.
≠VOLDUMP≠ ‡	Checkpoint dump at end-of-volume.
≠DUMP≠ ‡	Checkpoint dump after 50,000 records.
≠DUMP≠,n ‡	Checkpoint dump after (decimal) n records.
≠NODUMP≠ ‡	No checkpoint dumps.
≠NODAY≠ ‡	Suppress dayfile messages.
≠ORDER≠,mo ‡	Merge order = mo
≠ORDER*≠,mo ‡	Merge order ≤ mo

(default: mo = 5).

SMOPT specifies special record handling options.

§ If SMOPT is called, it must be done immediately after the call to SMSORT or SMMERGE.

‡ Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

§ Applies only to SCOPE 2.1.

## TAPE

### CALL SMTAPE (taplist)<sup>‡</sup>

taplist List of logical file names, each in the form nLfilename, to be used in balanced or polyphase tape merge.

The file names in taplist must not be declared in the PROGRAM statement. A balanced merge requires a minimum of four tapes; a polyphase merge, a minimum of three tapes.

## OWNCODE

### CALL SMOWN (exitnum<sub>1</sub>,subname<sub>1</sub>, . . . ,exitnum<sub>n</sub>,subname<sub>n</sub>)

exitnum Number of the owncode exit.

subname Name of the user-supplied owncode exit subroutine

Each subname specified in a call to SMOWN must appear in an EXTERNAL statement in the calling program. For each subname specified, the user must supply a subroutine which exits through a call to system subroutine SMRTN, in accordance with the owncode exit number and return address as follows:

exitnum	entry	exit
1 or 3	SUBROUTINE subname (a,rl)	CALL SMRTN (retaddr), for retaddr = 1 or 3 CALL SMRTN (retaddr,b,rl), for retaddr = 0 or 2
2 or 4	SUBROUTINE subname	CALL SMRTN (retaddr), for retaddr = 0 CALL SMRTN (retaddr,b,rl), for retaddr = 1
5	SUBROUTINE subname (a <sub>1</sub> ,rl <sub>1</sub> ,a <sub>2</sub> ,rl <sub>2</sub> )	CALL SMRTN (b <sub>1</sub> ,rl <sub>1</sub> ,b <sub>2</sub> ,rl <sub>2</sub> ), for retaddr = 0 CALL SMRTN (b <sub>1</sub> ,rl <sub>1</sub> ), for retaddr = 1

retaddr Return address:

- 0 Normal return address
- 1 Normal return address + 1
- 2 Normal return address + 2
- 3 Normal return address + 3

a } Integer array of length rl/10 in which Sort/Merge stores a record when subname is called  
b }

rl Record length in characters

No parameters are needed on SUBROUTINE subname for exit number 1 if there are no input files.

---

<sup>‡</sup>Applies only to NOS 1.0, KRONOS 2.1, and SCOPE 3.4.

**END**

**CALL SMEND**

Required as the last in a series of Sort/Merge interfacing subroutines, SMEND initiates execution of the sort or merge.

## STANDARD CHARACTER SETS

A

---

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

These character sets are listed in table A-1. The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). The user, however, may specify the alternate mode by a 26 or 29 punched in columns 79 and 80 of the job card or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS and KRONOS, the alternate mode can be specified also by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

When the 63-character set is used, the display code character 00<sub>g</sub> under A or R FORMAT conversion will be converted to a space, display code 55<sub>g</sub> for ENCODE and DECODE as well as FORMATTED I/O statement.

No conversions occur with the A or R FORMAT element when the 64-character set is used.

## STANDARD CHARACTER SETS

CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code	CDC Graphic	ASCII Graphic Subset	Display Code	Hollerith Punch (026)	External BCD Code	ASCII Punch (029)	ASCII Code
:†	:	00†	8-2	00	8-2	3A	6	6	41	6	06	6	36
A	A	01	12-1	61	12-1	41	7	7	42	7	07	7	37
B	B	02	12-2	62	12-2	42	8	8	43	8	10	8	38
C	C	03	12-3	63	12-3	43	9	9	44	9	11	9	39
D	D	04	12-4	64	12-4	44	+	+	45	12	60	12-8-6	2B
E	E	05	12-5	65	12-5	45	-	-	46	11	40	11	2D
F	F	06	12-6	66	12-6	46	*	*	47	11-8-4	54	11-8-4	2A
G	G	07	12-7	67	12-7	47	/	/	50	0-1	21	0-1	2F
H	H	10	12-8	70	12-8	48	(	(	51	0-8-4	34	12-8-5	28
I	I	11	12-9	71	12-9	49	)	)	52	12-8-4	74	11-8-5	29
J	J	12	11-1	41	11-1	4A	\$	\$	53	11-8-3	53	11-8-3	24
K	K	13	11-2	42	11-2	4B	=	=	54	8-3	13	8-6	3D
L	L	14	11-3	43	11-3	4C	blank	blank	55	no punch	20	no punch	20
M	M	15	11-4	44	11-4	4D	, (comma)	, (comma)	56	0-8-3	33	0-8-3	2C
N	N	16	11-5	45	11-5	4E	. (period)	. (period)	57	12-8-3	73	12-8-3	2E
O	O	17	11-6	46	11-6	4F	≡	#	60	0-8-6	36	8-3	23
P	P	20	11-7	47	11-7	50	{	{	61	8-7	17	12-8-2	5B
Q	Q	21	11-8	50	11-8	51	}	}	62	0-8-2	32	11-8-2	5D
R	R	22	11-9	51	11-9	52	%††	%	63	8-6	16	0-8-4	25
S	S	23	0-2	22	0-2	53	≠	" (quote)	64	8-4	14	8-7	22
T	T	24	0-3	23	0-3	54	→	_ (underline)	65	0-8-5	35	0-8-5	5F
U	U	25	0-4	24	0-4	55	∨	!	66	11-0 or 11-8-2†††	52	12-8-7 or 11-0†††	21
V	V	26	0-5	25	0-5	56	^	&	67	0-8-7	37	12	26
W	W	27	0-6	26	0-6	57	↑	' (apostrophe)	70	11-8-5	55	8-5	27
X	X	30	0-7	27	0-7	58	↓	?	71	11-8-6	56	0-8-7	3F
Y	Y	31	0-8	30	0-8	59	<	<	72	12-0 or 12-8-2†††	72	12-8-4 or 12-0†††	3C
Z	Z	32	0-9	31	0-9	5A	>	>	73	11-8-7	57	0-8-6	3E
0	0	33	0	12	0	30	∞	@	74	8-5	15	8-4	40
1	1	34	1	01	1	31	∩	\	75	12-8-5	75	0-8-2	5C
2	2	35	2	02	2	32	∪	˘ (circumflex)	76	12-8-6	76	11-8-7	5E
3	3	36	3	03	3	33	∩	;	77	12-8-7	77	11-8-6	3B
4	4	37	4	04	4	34	∪	;					
5	5	40	5	05	5	35	;	;					

† Twelve or more zero bits at the end of a 60-bit word are interpreted as end-of-line mark rather than two colons. End-of-line mark is converted to external BCD 1632.

†† In installations using the CDC 63-graphic set, display code 00 has no associated graphic or Hollerith code; display code 63 is the colon (8-2 punch).

††† The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

## HEXADECIMAL–OCTAL CONVERSION TABLE

		First Hexadecimal Digit															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Second Hexadecimal Digit	0	000	020	040	060	100	120	140	160	200	220	240	260	300	320	340	360
	1	001	021	041	061	101	121	141	161	201	221	241	261	301	321	341	361
	2	002	022	042	062	102	122	142	162	202	222	242	262	302	322	342	362
	3	003	023	043	063	103	123	143	163	203	223	243	263	303	323	343	363
	4	004	024	044	064	104	124	144	164	204	224	244	264	304	324	344	364
	5	005	025	045	065	105	125	145	165	205	225	245	265	305	325	345	365
	6	006	026	046	066	106	126	146	166	206	226	246	266	306	326	346	366
	7	007	027	047	067	107	127	147	167	207	227	247	267	307	327	347	367
	8	010	030	050	070	110	130	150	170	210	230	250	270	310	330	350	370
	9	011	031	051	071	111	131	151	171	211	231	251	271	311	331	351	371
	A	012	032	052	072	112	132	152	172	212	232	252	272	312	332	352	372
	B	013	033	053	073	113	133	153	173	213	233	253	273	313	333	353	373
	C	014	034	054	074	114	134	154	174	214	234	254	274	314	334	354	374
	D	015	035	055	075	115	135	155	175	215	235	255	275	315	335	355	375
	E	016	036	056	076	116	136	156	176	216	236	256	276	316	336	356	376
	F	017	037	057	077	117	137	157	177	217	237	257	277	317	337	357	377
Octal		000 – 037	040 – 077	100 – 137	140 – 177	200 – 237	240 – 277	300 – 337	340 – 377								





# INDEX

---

- Abort
  - dump III-12-3
  - recovery I-8-13
- Actual arguments I-7-7, I-7-9, I-7-12
- AREA debug statement I-13-26
- Arguments
  - actual I-7-7, I-7-9, I-7-12
  - dummy or formal I-7-7, I-7-9, I-7-12
  - using I-7-12
- Arithmetic
  - assignment I-4-1
  - complex III-4-8
  - double precision III-4-7
  - IF statement I-5-6
  - integer III-4-7
  - logical and masking III-4-8
  - mode errors III-4-8
  - operators I-3-1
  - statement function I-7-10
- Arrays
  - dimensions I-6-7, I-6-10, I-7-14, II-1-26
  - element location I-2-16, I-10-3
  - EQUIVALENCE I-6-14
  - NAMELIST I-9-19
  - names I-7-7
  - structure I-2-15
  - subscripts I-2-14
  - transmission I-6-21, I-10-2
  - type statements I-6-1
- ARRAYS debug statement I-13-4
- ASSIGN statement I-5-3
  - assigned GO TO I-5-4
- Assignment statements
  - arithmetic I-4-1
  - logical I-4-5
  - masking I-4-5
  - multiple I-4-6
- Asterisk
  - comment I-1-3
  - Hollerith I-10-27
  - multiplication I-3-1
- B suffix for octal I-2-5
- Binary
  - card punch I-9-2, III-13-5
  - I/O, see unformatted
  - Program execution III-13-6
- BACKSPACE I-9-12, III-5-8
- Basic external function I-7-9, I-8-6
- Blank
  - line I-1-3
  - common I-6-8
- Block
  - common I-6-6
  - data subprogram I-7-5
- Boolean, see masking
- Buffer I-7-4
  - input/output III-5-10
  - IN statement I-9-13, III-5-10
  - OUT statement I-9-14, III-5-11
  - PROGRAM statement I-7-2
- C comment line I-1-3
- C/ listing control III-12-2
- C\$ debug statement I-13-1
- CALL statement I-7-16
- Call-by-Name III-8-1, III-10-1
- Call-by-Value III-8-1, III-10-2
- Calling
  - overlay I-12-5
  - subroutine subprogram I-7-16
  - tracing subroutine calls I-13-6
- CALLS debug statement I-5-13
- Carriage control I-9-2, I-10-32
- Character set I-1-1, A-1
- CHECK III-6-6
- CHEKPTX I-8-11
- CLOSEM III-6-4
- CLOSMS I-8-17
- Coding column significance I-1-2
- Comma I-10-7
- Comment line I-1-3

- Common
  - statement I-6-6
  - and equivalence I-6-10, I-6-13
  - using common I-7-14
  - overlay communication I-12-3
- COMPASS assembler
  - calling sequence III-8-1, III-10-5
  - program entry points III-10-5
  - subprogram II-10-2
- Compilation modes III-1-1
  - listings III-12-1
  - modes III-1-1
- Compiler
  - call I-11-1
  - diagnostics III-2-1
  - supplied functions I-8-1
- Complex
  - arithmetic III-4-7
  - constants I-2-4
  - type statement I-6-2
  - variables I-2-11
- Computed GO TO I-5-2
- CONNEX I-8-21, III-11-1
- Constants
  - complex I-2-4, I-4-4
  - double precision I-2-3, I-4-3
  - Hollerith I-2-6, II-1-37
  - integer I-2-1, I-4-2
  - logical I-2-8
  - octal I-2-5, II-1-37
  - real I-2-2, I-4-3
- Continuation I-1-2
- Control
  - carriage I-9-1, I-10-32
  - column (Tn) I-10-34
  - listing III-12-2
  - statements I-5-1
- Control card
  - FILE III-5-6
  - OVERLAY I-12-4
  - parameters for FTN I-11-1
- Conversion
  - data on input/output I-10-6
  - mixed mode I-3-5
  - octal to Hexadecimal A-3
  - specifications for input/output I-10-7
- Cross reference map III-1-1
- D double precision constant I-2-3
- Data conversion on input/output I-10-6
- DATE I-8-9
- Dayfile messages I-8-10
- DEBUG statement I-13-24
- Debug
  - deck structure I-13-19
  - printing output I-13-30
  - statements I-13-1
  - STRACE entry point I-13-30
- Debugging aids I-8-15, I-13-1
  - ERRSET I-8-16, III-3-6
  - using reference map III-1-16
- Deck structure
  - program III-13-1
  - debug I-13-19
- Declarations I-6-1
- DECODE statement I-9-24, II-1-19
- Diagnostics II-2-1
  - compilation time III-2-1
  - execution time III-2-31
  - mass storage III-7-13
  - record manager III-6-9
- DIMENSION
  - statement I-6-5
  - adjustable I-7-14
- DISCON I-8-21, III-11-2
- DISPLA I-8-10
- Display code A-1
- Division I-3-1
  - by zero III-4-5
- DLTE III-6-5
- DO loops I-5-8
  - implied in DATA list I-6-21
  - implied in I/O list I-10-2
  - nested DO loops I-5-9
  - range I-5-7
  - transfer I-5-7, I-5-10
- DO statement I-5-7
- Dollar sign
  - comment column 1 I-1-3
  - multiple statement separator I-1-2
- Double precision
  - arithmetic III-4-7
  - constants I-2-3
  - conversion I-4-3
  - type declaration I-6-1
  - variables I-2-11

DUMP I-8-15, III-12-3  
Dw conversion, output and input I-10-16, I-10-17  
scaling I-10-24

ECS/LCM subprograms I-8-20  
LEVEL I-6-15  
ENCODE statement I-9-21, II-1-19  
END statement I-5-15  
EOF function I-8-19  
Equals sign I-10-35

EQUIVALENCE  
statement I-6-10  
and common I-6-13  
LEVEL I-6-15

Error codes  
execution time III-2-31  
mass storage III-7-11  
mode error III-4-10

Error processing  
by FORSYS= III-3-1  
by Record Manager III-6-9  
SYSTEM or SYSTEMC III-3-1

Errors, arithmetic mode III-4-8  
ERRSET I-8-16, III-3-6  
Evaluation of expressions I-3-2, I-3-8

Execution time  
diagnostics III-2-31  
file name handling III-3-9  
FORMAT I-10-38  
input/output III-5-1  
options III-3-8

EXIT I-8-11

Exponentiation I-3-6

Expressions I-3-1  
arithmetic I-3-1  
logical I-3-9  
masking I-3-13  
relational I-3-7  
subscripts I-2-14

Extended range of DO loop I-5-8

EXTERNAL  
function I-8-6  
statement I-6-16

Ew conversion, output and input I-10-9, I-10-11  
scaling I-10-24

FALSE I-2-8

File  
control card III-5-6

definition III-5-1  
labeled III-5-11  
name handling III-3-8  
name (TAPEu) I-7-3  
structure III-5-2

File information table (FIT)  
defaults for standard I/O III-5-3  
direct call by Record Manager III-6-1

FILExx III-6-1, III-6-10

Floating point III-4-1  
indefinite and infinite results III-4-4  
overflow III-4-3  
underflow III-4-3

Formal argument (parameter) See Dummy  
argument

FORMAT  
execution time I-10-38  
repeat specification I-10-31  
slash I-10-29  
statement I-10-5

FORSYS= III-3-1

FORTRAN compiler call I-11-1

FTN control card I-11-1

FUNCS debug statement I-13-8

Function  
basic external I-7-9, I-8-5  
intrinsic I-7-10, I-8-1  
referencing a I-7-15  
statement I-7-10  
subprogram I-7-6  
tracing a reference I-13-8

Fw conversion, output and input I-10-13, I-10-14  
scaling I-10-23

GET III-6-4

GETN III-6-5

GETP III-6-7

GO TO statements I-5-1  
assigned GO TO I-5-4  
computed GO TO I-5-2  
unconditional GO TO I-5-1

GOTOS debug statement I-13-15

Gw conversion, input and output I-10-15  
scaling I-10-24

H specification I-2-6

Hashing key for direct access file III-6-8

Hexadecimal/Octal conversion A-3

Hierarchy in expressions I-3-2  
 Hollerith  
   constant I-2-6  
   data interpreted by STORES I-13-14  
   format element I-10-25  
   input specification I-10-25  
   output specification I-10-26  
  
 IF statements I-5-5  
   standard-form logical I-5-6  
   three-branch arithmetic I-5-5  
   two-branch arithmetic I-5-5  
   two-branch logical I-5-7  
 IMPLICIT  
   statement I-6-3  
   typing of variables I-2-9  
 Implied DO in  
   DATA list I-6-21, II-1-22  
   I/O list I-10-2  
 Indefinite result, floating point III-4-4  
 Index  
   DO loop I-5-8  
   mass storage files III-7-2  
   MIP files III-6-10  
   STINDEX III-7-11  
 Infinite result, floating point III-4-4  
 Input  
   BUFFER IN III-5-10  
   file I-7-3  
   list directed I-9-9  
   NAMELIST I-9-17  
   statements I-9-7  
 Input/Output  
   BUFFER III-5-10  
   execution time III-5-1  
   lists I-10-1, I-10-31  
   statement definition I-9-1  
   status checking I-8-18  
 IOCHEC I-8-19  
 Integer  
   arithmetic I-2-1, III-4-7  
   constants I-2-1  
   conversion I-4-2  
   statement I-6-2  
   variables I-2-10  
 INTERCOM, terminal I/O III-11-1  
 Intrinsic functions I-7-10, I-8-1  
 Iw conversion, input and output I-10-8  
  
 JDATE I-8-9  
 Job decks, sample III-13-1  
  
 L specification I-2-6  
 LABEL I-8-20, III-5-11  
 Labels  
   statement number I-1-3  
   RETURNS list I-7-7  
 Labeled  
   common I-6-6  
   files III-5-11  
 LEGVAR I-8-15  
 LENGTH I-8-19  
 LCM, see ECS  
 LEVEL  
   statement I-6-7, I-6-15  
   OVERLAY I-12-1  
 LGO I-11-2  
 Library functions I-8-1  
 List directed  
   input data forms I-9-10, II-1-40  
   output data forms I-9-11, II-1-40  
   READ I-9-9  
   WRITE I-9-7  
 Listings  
   control III-12-1  
   map III-1-1  
 Load map III-12-7  
 Loader control cards  
   LDSET III-6-2  
   OVERLAY I-12-4  
 LOGICAL  
   assignment statement I-4-5  
   constants I-2-8  
   file names III-3-8  
   and masking operations III-4-8  
   statement I-6-3  
   unit number I-7-3  
   variables I-2-11  
 Looping, DO I-5-8  
 Lw conversion, input and output I-10-22  
  
 Main program I-7-2  
 Map symbolic or cross reference  
   optimizing mode III-1-1  
   TS mode III-1-17  
 Masking  
   expression I-3-13

- assignment statement I-4-5
- Mass storage input/output III-7-1
- OPENMS I-8-16, III-7-9
- STINDEX I-8-17, III-7-11
- CLOSMS I-8-17, III-7-11
- READMS I-8-17, III-7-10
- WRITMS I-8-17, III-7-9
- Memory layout III-9-3
- Messages
  - TS mode diagnostic III-2-14
  - optimizing mode diagnostic III-2-1
  - special compilation III-2-10
  - execution diagnostics III-2-30
- Mixed-Mode arithmetic conversion I-3-5
- Mode
  - arithmetic errors III-4-9
  - debug I-13-1
  - optimizing III-14-1
  - time-sharing (TS) III-15-1
- MOVLEV I-8-20
- Multiple
  - assignment statement I-4-6
  - index processing III-6-10
  - statement separator \$ I-1-2

- NAMELIST statement I-9-15
- READ I-9-16
- WRITE I-9-18

- Names
  - common block I-6-7
  - file I-7-3
  - PROGRAM I-7-2
  - variable I-2-9

- Nesting
  - DO loops I-5-8
  - parentheses I-10-2
- NOGO debug statement I-13-18

- Number
  - common block I-6-7
  - formats, see constants
  - statement label I-1-3

- Object program execution III-13-6
- Octal
  - constants I-2-5
  - hexadecimal conversion A-3
- OFF debug statement I-13-28
- OPENM III-6-4
- OPENMS I-8-16, III-7-9

- Operands
  - evaluation I-3-6
  - result III-4-5
- Operating system interface routines I-8-9
- Operators I-3-1
- Optimization
  - object code III-14-2
  - source code III-4-4
  - unsafe III-14-5
- Options FTN control card I-11-1
- Order, statements in program unit III-9-1
- Output
  - BUFFER OUT III-5-11
  - file I-7-3
  - list directed data forms I-9-11
  - NAMELIST data form I-9-18
  - print limit specification III-3-10
  - record length I-9-2
  - statements I-9-3
- Overflow, arithmetic III-4-3
- Overlays I-12-1
  - directive I-12-4
  - sample deck I-12-7, III-13-9
- Ow conversion, input and output I-10-7, I-10-8
- Owncode, COMPASS III-10-1

- P scale factors I-10-22
- Parameter, see argument
- Parentheses, nesting I-10-2
- PAUSE statement I-5-14
- PDUMP I-8-15
- Precedence of operators I-3-2
- Print

- control characters I-9-2, I-10-32
- error frequency III-3-1
- limit specification III-3-10
- statement I-9-3, I-9-11

- Procedure communication I-7-12
  - passing values I-7-12
  - using arguments I-7-12

- Program
  - maps III-1-1
  - sample II-1-1
  - statement I-7-2
  - units I-7-1

- Punch
  - codes A-2
  - file I-7-3
  - statements I-9-4
- PUT III-6-5

PUTP III-6-8

R specification I-2-6

Random
 

- access III-7-1
- files III-5-6, III-6-1
- number routines I-8-16

RANF I-8-16

Range of DO loop I-5-8

READ statements I-9-7
 

- formatted I-9-7
- list directed I-9-9, I-9-10
- NAMELIST I-9-16
- unformatted I-9-8

READEC I-8-20

READMS I-8-17

Real
 

- constant I-2-2
- conversion I-4-3
- statement I-6-2
- variable I-2-10

Record
 

- definition III-5-1
- I/O record length I-7-3, I-9-2
- types III-5-1

Record Manager
 

- file handling III-5-1
- files/direct handling III-6-1

Recovery I-8-13, III-3-3

RECOVER I-8-13

Reference maps III-1-1

Register names III-8-1

Relational
 

- evaluation I-3-8
- operators I-3-7

REMARK I-8-10

REPLC III-6-6

RETURN statement I-5-15

RETURNS list I-5-16, I-7-7, I-7-16

REWIND I-9-12, III-5-8

REWND III-6-7

RMKDEF III-6-10

RMOPNX III-6-10

Rw conversion, input and output I-10-21

FTN control card I-11-9
 

- programs II-1-1

Scale factors I-10-22

Scaling I-10-23

SECOND I-9-10

SEEKF III-6-6

Sense
 

- light I-8-11
- switch I-8-10

Separator
 

- slash and comma I-10-7
- \$ statement I-1-3

Sequential file structure III-5-2

SKIP III-6-6

Slash in FORMAT statement I-10-29, I-10-31

SLITE I-8-10

SLITET I-8-11

Sort/Merge interface III-16-1

Specification statements I-6-1

SSWTCH I-8-11

Standard, FORTRAN ANSI v

Statement
 

- format I-1-2
- FORTRAN, see individual statement name
- function I-7-10
- labels or numbers I-1-3
- order in program unit III-9-1
- separator, multiple I-1-2

STINDEX I-8-17, III-7-11

STOP statement I-5-14

STOREF III-6-3

STORES debug statement I-13-11

STRACE I-8-15

Structure
 

- debug decks I-13-19
- memory III-9-3
- program units III-9-1

Subprograms
 

- block data I-7-5
- function I-7-8
- subroutine I-7-6

Subroutine
 

- calling I-7-16
- statement I-7-6

Subscripts I-2-14
 

- and arrays I-2-12
- checking in debug I-13-4

Symbolic
 

- or cross reference map III-1-1, III-1-17
- name I-2-9

Syntax xi

Sample
 

- coding form I-1-4
- COMPASS subprogram III-10-4
- decks III-13-1

SYSTEM and SYSTEMC I-8-15, III-3-1

TAPEu I-7-3

Terminal interface III-11-1

CONNEC I-8-21, III-11-1

DISCON I-8-21, III-11-1

Texts, system I-11-7

TIME I-8-10

Time-Sharing (TS) mode

compilation diagnostics III-2-18

characteristics III-15-1

cross reference map III-1-17

terminal interface III-11-1

Tn (tab) specification I-10-34

TRACE

debug statement I-13-16

reference I-3-18

Traceback mode I-13-30

TS mode III-15-1

TRUE I-2-8

Type of

arithmetic expressions I-3-5

function I-7-9, I-8-1

masking expression I-3-15

variable I-2-9

Type statements

dimension information in I-6-5

explicit I-6-2

implicit I-6-3

Unary operators and evaluation I-3-3

Unconditional GO TO I-5-1

Underflow, arithmetic III-4-3

Unformatted

READ I-9-8

WRITE I-9-6

UNIT I-8-18

Unit number I-7-3

Utility subprograms I-8-9

V specification I-10-35

Variable

dimensions in a subprogram I-7-7, I-7-14

FORMAT statements I-10-38

name and type I-2-9

Variables I-2-9

complex I-2-11

double precision I-2-11

integer I-2-10

logical I-2-11

real I-2-10

WEOR III-6-7

WRITE statement I-9-5, I-9-6

formatted I-9-5

list directed I-9-7

NAMELIST I-9-18

unformatted I-9-6

WRITEC I-8-20

WRITMS I-8-17

WTMK III-6-7

X specification I-10-24

Zero operand III-4-5

Zw conversion, input and output I-10-19

.AND. I-3-9

.EQ. I-3-7

.FALSE. I-2-8

.GE. I-3-7

.GT. I-3-7

.LE. I-3-7

.LT. I-3-7

.NE. I-3-7

.NOT. I-3-9

.OR. I-3-9

.TRUE. I-2-8

\$ I-1-3

\* I-1-3, I-10-27

≠ I-10-27

/ I-10-29, I-10-31





COMMENT SHEET



TITLE: FORTRAN Extended Version 4 Reference Manual

PUBLICATION NO. 60305600

REVISION G

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

General comments:

FROM NAME: \_\_\_\_\_ POSITION: \_\_\_\_\_

COMPANY  
NAME: \_\_\_\_\_

ADDRESS: \_\_\_\_\_

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.  
FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS  
PERMIT NO. 8241

MINNEAPOLIS, MINN.

**BUSINESS REPLY MAIL**  
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

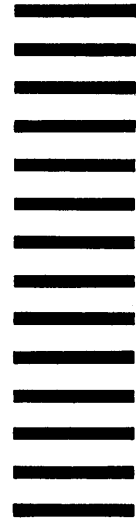
POSTAGE WILL BE PAID BY

**CONTROL DATA CORPORATION**

*Publications and Graphics Division*

**215 Moffett Park Drive**

**Sunnyvale, California 94086**



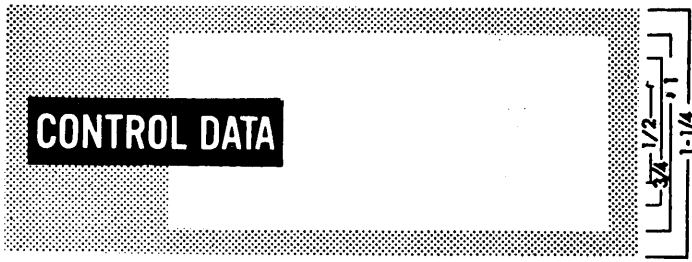
FOLD

FOLD

STAPLE

STAPLE





▶▶ CUT OUT FOR USE AS LOOSE-LEAF BINDER TITLE TAB

**CONTROL DATA**  
CORPORATION

8100 34th AVE. SO., MINNEAPOLIS, MINN. 55440