



**FORTRAN
VERSION 5
REFERENCE MANUAL**

CDC® OPERATING SYSTEMS:

NOS 1

NOS 2

NOS/BE 1

SCOPE 2

REVISION RECORD

<u>Revision</u>	<u>Description</u>
A (07/20/79)	Original release.
B (09/28/79)	Revised to reflect the released version of the FORTRAN 5 compiler. Numerous technical and miscellaneous corrections, including added shading, have been made.
C (02/15/80)	Revised to include SCOPE 2 operating system. Additional technical and miscellaneous corrections have been made.
D (09/26/80)	Released at PSR level 527. Revised to include CMM interface and update to PMD. Additional technical and miscellaneous corrections have been made.
E (01/16/81)	Revised at PSR level 533. Revised to reflect release of PMD and STATIC option under SCOPE 2 operating system.
F (05/14/82)	Revised at PSR level 564. Revised to reflect the addition of control statement parameters to the source listing header. Additional technical and editorial corrections have been made. This is a complete reprint.
G (07/15/83)	This revision documents FORTRAN 5 at PSR level 577 which includes the addition of the extended recovery feature. Additional technical and editorial corrections have been made.
H (01/04/85)	This revision documents FORTRAN 5 at PSR level 587. Additional technical and editorial corrections have been made.

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

Address comments concerning this manual to:

©COPYRIGHT CONTROL DATA CORPORATION
1979, 1980, 1981, 1982, 1983, 1985
All Rights Reserved
Printed in the United States of America

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

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

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

<u>Page</u>	<u>Revision</u>	<u>Page</u>	<u>Revision</u>	<u>Page</u>	<u>Revision</u>
Front Cover	-	5-29	G	11-3	G
Title Page	-	5-30	H	11-4	G
ii	H	5-31	G	11-5	F
iii/iv	H	5-32	H	11-6	G
v	H	5-33	F	11-7	G
vi	F	5-34 thru 5-41	H	11-8	F
vii/viii	F	5-42 thru 5-44	F	11-9	H
ix thru xiv	H	6-1	G	11-10	H
xv/xvi	H	6-2	H	11-11	F
xvii	F	6-3	H	11-12	F
1-1 thru 1-3	F	6-4	G	11-13	G
1-4	H	6-5	F	11-14 thru 11-20	F
1-5	F	6-6	F	11-21	G
1-6	F	6-7	H	11-22 thru 11-26	F
1-7	H	6-8	F	11-27 thru 11-29	H
1-8	H	6-9 thru 6-11	H	12-1	F
1-9	G	6-12	F	12-2 thru 12-8	G
1-10	G	6-13	F	12-8.1	G
1-11 thru 1-13	F	7-1	G	12-8.2	G
2-1	G	7-2	G	12-9 thru 12-18	F
2-2	F	7-2.1/7-2.2	G	12-19	G
2-3	G	7-3 thru 7-11	F	12-20 thru 12-35	F
2-4	F	7-12	G	A-1	F
2-5	H	7-13	H	A-2	D
2-6 thru 2-10	G	7-14	F	A-3	E
2-11 thru 2-14	F	7-15	G	A-4	A
3-1 thru 3-3	F	7-16	H	B-1 thru B-23	F
3-4	G	7-17 thru 7-22	G	B-24	G
3-5	F	7-22.1/7-22.2	G	B-25	G
3-6	H	7-23 thru 7-28	F	B-26 thru B-31	F
3-7	F	7-29	G	B-32 thru B-34	G
3-8	G	7-30 thru 7-33	F	B-34.1/B-34.2	G
3-9	H	8-1	G	B-35 thru B-39	F
3-10	F	8-2	H	B-40	H
4-1 thru 4-6	F	8-3 thru 8-13	F	B-41	H
4-7 thru 4-10	H	8-14	H	C-1	F
5-1	F	8-15	H	C-2	H
5-2	G	8-16 thru 8-18	F	C-2.1/C-2.2	G
5-3	F	8-19	H	C-3 thru C-5	F
5-4	F	8-20 thru 8-22	F	D-1 thru D-7	F
5-5	G	8-23	H	E-1	G
5-6	F	8-24	H	E-2 thru E-4	F
5-7	F	8-25	F	F-1	F
5-8	G	8-26	G	F-2	G
5-9	H	8-27	F	F-3	F
5-10	H	8-28	F	F-4	G
5-11	F	8-29	G	F-5	F
5-12	G	8-30	F	F-6	H
5-13	F	8-31	F	F-7	C
5-14	F	8-32	G	F-8	F
5-15	G	8-33 thru 8-38	F	F-9	F
5-16	F	9-1	F	G-1	F
5-17 thru 5-19	H	9-2	G	G-2	G
5-20	F	9-3 thru 9-6	F	Index-1 thru -6	H
5-21	F	10-1 thru 10-6	F	Comment Sheet/Mailer	H
5-22 thru 5-24	H	11-1	G	Back Cover	-
5-25	G	11-2	H		
5-26 thru 5-28	F	11-2.1/11-2.2	H		

PREFACE

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

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

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

NOS 1 for the CONTROL DATA® CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

NOS 2 for the CDC® CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

NOS/BE 1 for the CDC CYBER 180 Series; CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

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

Extended memory for the CYBER 170 Model 176 is large central memory (LCM) or large central memory extended (LCME). Extended memory for the CYBER 170 800 Series Computer Systems is unified extended memory (UEM). Extended memory for all other computer systems is extended core storage (ECS) or extended semi-conductor memory (ESM). In this manual, the acronym ECS refers to all forms of extended memory unless otherwise noted. Programming information for the various forms of extended memory can be found in the COMPASS reference manual and in the appropriate computer system hardware reference manual.

Related material is contained in the listed publications. These publications are listed alphabetically and grouped according to their importance to the FORTRAN user. The NOS 1, NOS 2, and NOS/BE 1 manual abstracts are pocket-sized manuals containing brief descriptions of the contents and intended audience of all operating system and product set manuals. The abstracts manuals can be useful in determining which manuals are of greatest interest to a particular user.

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

The following manuals are of primary interest:

<u>Publication</u>	<u>Publication Number</u>	<u>NOS 1</u>	<u>NOS 2</u>	<u>NOS/BE 1</u>	<u>SCOPE 2</u>
FORTRAN Extended Version 4 to FORTRAN Version 5 Conversion Aid Program Reference Manual	60483000	X	X	X	X
FORTRAN Version 5 Common Library Mathematical Routines Reference Manual	60483100	X	X	X	X
FORTRAN Version 5 Instant	60483900	X	X	X	X
NOS Version 1 Reference Manual, Volume 1 of 2	60435400	X			
NOS Version 2 Reference Set, Volume 3, System Commands	60459680		X		
NOS/BE Version 1 Reference Manual	60493800			X	
SCOPE Version 2 Reference Manual	60342600				X

The following manuals are of secondary interest:

<u>Publication</u>	<u>Publication Number</u>	<u>NOS 1</u>	<u>NOS 2</u>	<u>NOS/BE 1</u>	<u>SCOPE 2</u>
Common Memory Manager Version 1 Reference Manual	60499200	X	X	X	X
COMPASS Version 3 Reference Manual	60492600	X	X	X	X
CYBER Interactive Debug Version 1 Reference Manual	60481400	X	X	X	
CYBER Loader Version 1 Reference Manual	60429800	X	X	X	
CYBER Record Manager Advanced Access Methods Version 2 Reference Manual	60499300	X	X	X	
CYBER Record Manager Advanced Access Methods Version 2 User's Guide	60499400	X	X	X	
CYBER Record Manager Basic Access Methods Version 1.5 Reference Manual	60495700	X	X	X	
CYBER Record Manager Basic Access Methods Version 1.5 User's Guide	60495800	X	X	X	
DMS-170 FORTRAN Data Base Facility Version 1 Reference Manual	60482200	X	X	X	
FORTRAN Version 5 User's Guide	60484000	X			
FORTRAN Version 5 User's Guide	60484010				X
Information Management Facility Version 1 Application Programming Reference Manual	60484500	X	X		
INTERCOM Version 5 Reference Manual	60455010				X
Loader User's Guide	60485200		X		
Network Products Interactive Facility Version 1 Reference Manual	60455250	X			
NOS Version 1 Diagnostic Index	60455720	X			
NOS Version 1 Manual Abstracts	84000420	X			
NOS Version 1 Time-Sharing User's Reference Manual	60435500	X			
NOS Version 2 Diagnostic Index	60459390		X		
NOS Version 2 Manual Abstracts	60485500		X		
NOS Version 2 Reference Set, Volume 1, Introduction to Interactive Usage	60459660		X		
NOS/BE Version 1 Manual Abstracts	84000470				X
NOS/BE Version 1 Diagnostic Index	60456490				X
SCOPE Version 2 Loader Reference Manual	60454780				X

SCOPE Version 2 Record Manager Reference Manual	60495700				X
Software Publications Release History	60481000	X	X	X	X
Sort/Merge Versions 4 and 1 Reference Manual	60497500	X	X	X	X
Sort/Merge Version 5 Reference Manual	60484800		X	X	
8-Bit Subroutines Reference Manual	60495500	X	X	X	

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

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

CONTENTS

NOTATIONS	xvii	3. EXPRESSIONS AND ASSIGNMENT STATEMENTS	3-1
1. LANGUAGE ELEMENTS	1-1	Expressions	3-1
FORTRAN Statements	1-1	Arithmetic Expression	3-1
Nonsequenced Mode	1-1	Infinite and Indefinite Values	3-4
Initial Lines	1-2	Boolean Expression	3-4
Continuation Lines	1-2	Character Expression	3-5
Statement Labels	1-2	Logical Expression	3-5
Comment Lines	1-2	Relational Expression	3-6
Compiler Directive Lines	1-3	General Rules for Expressions	3-8
Columns 73 Through 80	1-3	Assignment Statements	3-9
Sequenced Mode	1-3	Arithmetic Assignment	3-9
Symbolic Names	1-4	Boolean Assignment	3-9
Constants	1-4	Character Assignment	3-9
Integer Constant	1-4	Logical Assignment	3-9
Real Constant	1-5	Multiple Assignment	3-10
Double Precision Constant	1-5	4. FLOW CONTROL STATEMENTS	4-1
Complex Constant	1-6	GO TO Statement	4-1
Logical Constant	1-6	Unconditional GO TO Statement	4-1
Boolean Constants	1-6	Computed GO TO Statement	4-1
Hollerith and Extended Hollerith		ASSIGN Statement	4-1
Boolean Constant	1-7	Assigned GO TO Statement	4-2
Octal Boolean Constant	1-7	IF Statement	4-3
Hexadecimal Boolean Constant	1-7	Arithmetic IF Statement	4-3
Character Constant	1-7	Logical IF Statement	4-3
Variables	1-8	Block IF Statement	4-3
Integer Variable	1-8	ELSE Statement	4-4
Real Variable	1-8	ELSE IF Statement	4-4
Double Precision Variable	1-8	END IF Statement	4-4
Complex Variable	1-8	Block IF Structures	4-4
Logical Variable	1-8	Nested Block IF Structures	4-5
Boolean Variable	1-8	DO Statement	4-6
Character Variable	1-9	Active and Inactive DO Loops	4-7
Arrays	1-9	Nested DO Loops	4-7
Array Storage	1-10	CONTINUE Statement	4-8
Array References	1-10	PAUSE Statement	4-9
Character Substrings	1-11	STOP Statement	4-10
Substring References	1-11	END Statement	4-10
Substrings and Arrays	1-12	5. INPUT/OUTPUT	5-1
Statement Order	1-12	Input/Output Files	5-1
2. SPECIFICATION STATEMENTS	2-1	Sequential Files	5-1
COMMON Statement	2-1	Direct Access Files	5-1
DATA Statement	2-3	Batch Job Files	5-2
Implied DO List Use in DATA Statement	2-4	input/Output Lists	5-3
Character Data Initialization	2-5	Implied DO in Input/Output List	5-3
DIMENSION Statement	2-5	Format Specification	5-4
EQUIVALENCE Statement	2-6	FORMAT Statement	5-5
EXTERNAL Statement	2-7	Character Format Specifications	5-5
IMPLICIT Statement	2-7	Noncharacter Format Specifications	5-5
INTRINSIC Statement	2-8	Edit Descriptors	5-5
LEVEL Statement	2-9	Input/Output Conversions	5-7
PARAMETER Statement	2-9	Field Separators	5-7
SAVE Statement	2-10	Repeatable and Nonrepeatable Edit	
Type Statements	2-11	Descriptors	5-7
INTEGER Statement	2-12	Repetition Factors	5-22
REAL Statement	2-12	Execution Time Format Specification	5-22
DOUBLE PRECISION Statement	2-12	Input/Output Statements	5-22
COMPLEX Statement	2-12	Keyword=Value Forms in Input/Output	
LOGICAL Statement	2-12	Statements	5-22
BOOLEAN Statement	2-13	Formatted Input/Output Statements	5-25
CHARACTER Statement	2-13		

Formatted READ	5-25	AMIN1	7-7
Formatted WRITE	5-25	AMOD	7-7
Formatted PRINT	5-26	AND	7-7
Formatted PUNCH	5-26	ANINT	7-7
Unformatted Input/Output Statements	5-26	ASIN	7-7
Unformatted READ	5-27	ATAN	7-7
Unformatted WRITE	5-27	ATANH	7-7
List Directed Input/Output Statements	5-27	ATAN2	7-8
List Directed Input	5-27	BOOL	7-8
List Directed Output	5-29	CABS	7-8
NAMELIST Input/Output Statements	5-30	CCOS	7-8
Namelist Input	5-30	CEXP	7-8
Namelist Output	5-32	CHAR	7-8
Arrays in Namelist	5-32	CLOG	7-8
Buffer Input/Output Statements	5-32	CMLPX	7-8
BUFFER IN	5-34	COMPL	7-8
BUFFER OUT	5-35	CONJG	7-8
Input/Output File Status Statements	5-36	COS	7-8
OPEN	5-36	COSD	7-8
CLOSE	5-38	COSH	7-8
INQUIRE	5-38	CSIN	7-8
Memory-to-Memory Input/Output Statements	5-38	CSQRT	7-9
Standard Internal Files	5-40	DABS	7-9
Extended Internal Files	5-41	DACOS	7-9
File Positioning Statements	5-43	DASIN	7-9
REWIND	5-43	DATAN	7-9
BACKSPACE	5-44	DATAN2	7-9
ENDFILE	5-44	DBLE	7-9
		DCOS	7-9
		DCOSH	7-9
		DDIM	7-9
		DEXP	7-9
		DIM	7-9
		DINT	7-9
		DLOG	7-9
		DLOG10	7-9
		DMAX1	7-9
		DMIN1	7-9
		DMOD	7-9
		DNINT	7-10
		DPROD	7-10
		DSIGN	7-10
		DSIN	7-10
		DSINH	7-10
		DSQRT	7-10
		DTAN	7-10
		DTANH	7-10
		EQV	7-10
		ERF	7-10
		ERFC	7-10
		EXP	7-10
		FLOAT	7-10
		IABS	7-10
		ICHAR	7-10
		IDIM	7-10
		IDINT	7-10
		IDNINT	7-10
		IFIX	7-11
		INDEX	7-11
		INT	7-11
		ISIGN	7-11
		LEN	7-11
		LGE	7-11
		LGT	7-11
		LLE	7-11
		LLT	7-11
		LOCF	7-11
		LOG	7-12
		LOG10	7-12
		MASK	7-12
		MAX	7-12
		MAXO	7-12
		MAX1	7-12
		MIN	7-12
		MINO	7-12
		MIN1	7-12
6. USER-WRITTEN PROGRAMS, SUBPROGRAMS, AND FUNCTIONS	6-1		
Main Program	6-1		
PROGRAM Statement	6-1		
PROGRAM Statement Usage	6-1		
Procedures, Subprograms, and Statement Functions	6-3		
Subroutine Subprogram	6-3		
SUBROUTINE Statement	6-4		
CALL Statement	6-5		
Function Subprogram	6-5		
FUNCTION Statement	6-5		
Function Subprogram Reference	6-6		
ENTRY Statement	6-6		
RETURN and END Statements	6-7		
Single and Multiple Returns	6-7		
Alternate Return	6-8		
Statement Functions	6-8		
Statement Function Definition	6-8		
Statement Function Reference	6-9		
Program Unit and Procedure Communication	6-9		
Passing Arguments	6-9		
Actual Arguments	6-10		
Dummy Arguments	6-10		
Argument Association	6-10		
Using Common Blocks	6-12		
Block Data Subprograms	6-13		
7. FORTRAN SUPPLIED PROCEDURES	7-1		
Intrinsic Functions	7-1		
Generic and Specific Functions	7-1		
Boolean Arguments Passed to Intrinsic Functions	7-2		
Function Descriptions	7-2		
ABS	7-7		
ACOS	7-7		
AIMAG	7-7		
AINT	7-7		
ALOG	7-7		
ALOG10	7-7		
AMAXO	7-7		
AMAX1	7-7		
AMINO	7-7		

MOD	7-12	File Processing	8-6
NEQV	7-12	CRM Utilities	8-16
NINT	7-12	Sort/Merge	8-16
OR	7-12	Sort/Merge 5	8-18
RANF	7-12	Sort/Merge 5 Keys	8-18
REAL	7-12	Summing	8-18
SECOND	7-13	Sort/Merge 5 Owncode Routines	8-21
SHIFT	7-13	Sort/Merge 4 and 1	8-21
SIGN	7-13	Common Memory Manager	8-24
SIN	7-13	COMPASS Assembly Language Intermixed	
SIND	7-13	Subprograms	8-28
SINH	7-13	Compiler and Subprogram Linkage	8-28
SNGL	7-13	Pass By Reference Sequence	8-28
SQRT	7-13	Pass By Value Sequence	8-29
TAN	7-13	Function Result	8-29
TAND	7-13	Entry Point	8-29
TANH	7-13	Restrictions on Using Intrinsic Function	
XOR	7-13	Names	8-29
GETPARG Subroutine	7-13	8-Bit Subroutines	8-30
Random Number Routines	7-14	CYBER Database Control System	8-30
RANSET Subroutine	7-14	Information Management Facility	8-35
RANGET Subroutine	7-14	Queued Terminal Record Manager (QTRM)	8-36
Operating System Interfaces	7-14	Transaction Facility (TAF)	8-37
DATE Function	7-14		
JDATE Function	7-14		
TIME or CLOCK Function	7-14		
DISPLA Subroutine	7-14		
REMARK Subroutine	7-15	9. OVERLAYS AND OVCAPS	9-1
SSWTCH Subroutine	7-15	Overlays	9-1
EXIT Subroutine	7-15	Main, Primary, and Secondary Overlays	9-1
CHEKPTX Subroutine	7-15	Overlay Communication	9-2
RECOVR Subroutine	7-15	Creating Overlays	9-2
Input/Output Subprograms	7-17	Calling Overlays	9-4
Status Checking	7-17	OVCAPS	9-5
UNIT Function	7-18	OVCAP Communication	9-5
EOF Function	7-18	Creating OVCAPS	9-5
IOCHEC Function	7-18	Calling OVCAPS	9-5
LENGTH Function and LENGTHX			
Subroutine	7-19	10. DEBUGGING AIDS	10-1
Extended Memory	7-19	CYBER Interactive Debug	10-1
MOVLEV Subroutine	7-19	Program Compilation	10-1
MOVLCH Subroutine	7-19	DEBUG Control Statement	10-1
Interactive Terminals	7-20	DB Parameter	10-1
CONNAC Subroutine	7-20	Debug Session	10-2
DISCON Subroutine	7-21	SET,BREAKPOINT Command	10-2
Tape LABEL Subroutine	7-21	SET,TRAP Command	10-2
Mass Storage	7-22	PRINT Command	10-2
Random File Access	7-22	Assignment Command	10-2
OPENMS Subroutine	7-22.1	Other CID Features	10-3
WRITMS Subroutine	7-22.1	Post Mortem Dump	10-3
READMS Subroutine	7-22.1	Compilation	10-3
CLOSMS Subroutine	7-23	Loading	10-3
STINDX Subroutine	7-23	Dump Information	10-3
Debugging Routines	7-25	Dump Variables	10-4
DUMP and PDUMP Subroutines	7-27	PMD Traceback	10-4
STRACE Subroutine	7-27	PMD Subroutines	10-5
LEGVAR Function	7-27	PMDARRAY	10-5
SYSTEM Subroutine	7-27	PMDDUMP	10-5
SYSTEMC Subroutine	7-27	PMDLOAD	10-6
LIMERR Subroutine and NUMERR Function	7-28	PMDSTOP	10-6
Collating Sequence Control Subprograms	7-30		
COLSEQ Subroutine	7-31		
WTSET Subroutine	7-32		
CSOWN Subroutine	7-32		
Static Loading Option	7-32		
		11. COMPILATION AND EXECUTION	11-1
8. FORTRAN 5 INTERFACES	8-1	FTN5 Control Statement	11-1
Permanent File Interface	8-1	Parameters	11-1
PF Call	8-2	Binary Value Parameters	11-1
Error Processing	8-2	Specified Value Parameters	11-1
CYBER Record Manager (CRM)	8-5	Multiple Binary Value Parameters	11-1
File Information Table (FIT)	8-6	Multiple Appearances of Parameters	11-2
Call Syntax	8-6	Parameter Options	11-2
		AL Automatic Level	11-2
		ANSI Diagnostics	11-2.1
		ARG Argument List Attributes	11-3

B	Binary Output File	11-3
BL	Burstable Listing	11-3
CS	Collating Sequence	11-3
DB	Debugging Option	11-3
DO	Loop Control	11-4
DS	Directive Suppression	11-4
E	Error File	11-4
EC	Extended Memory Usage	11-5
EL	Error Level	11-5
ET	Error Terminate	11-5
G	Get System Text File	11-5
GO	Automatic Execution	11-6
I	Input File	11-6
L	List File	11-6
LCM	Extended Memory (LCM, ECS, or UEM Storage) Access	11-6
LO	Listing Options	11-6
MD	Machine Dependent Diagnostics	11-7
ML	MODLEVEL Micro	11-7
OPT	Optimization Level	11-7
PD	Print Density	11-9
PL	Print Limit	11-9
PN	Pagination	11-9
PS	Page Size	11-9
PW	Page Width	11-9
QC	Quick Syntax Check	11-10
REW	Rewind Files	11-10
ROUND	Rounded Arithmetic Computations	11-10
S	System Text File	11-10
SEQ	Sequenced Input	11-11
STATIC	Static Load	11-11
TM	Target Machine	11-11
X	External Text Name	11-11
FTN 5	Control Statement Examples	11-11
Compiler Listings		11-12
Short Line Listing		11-12
Listing Control Directive		11-13
Reference Map		11-13
General Format of Maps		11-13
Variable Map		11-13
Symbolic Constant Map		11-13
Procedure Map		11-13
Statement Label Map		11-16
Entry Point Map		11-16
Input/Output Unit Map		11-16
NAMELIST Map		11-16
DO Loop Map		11-16
Common and Equivalence Map		11-16
Stray Names		11-19
Program Statistics		11-19
Debugging Using the Reference Map		11-19
Object Listing		11-20
Program Unit Structure		11-26
Naming Conventions		11-26
Register Name Conflicts		11-26
System-Supplied Procedure Names		11-26
Listing Format		11-26
Execution Control Statement		11-27
File Name Substitution		11-27
Print Limit Specification		11-27
User Parameters		11-27
Post Mortem Dump Parameters		11-28
Post Mortem Dump Output Parameter		11-28
Subscript Limit Specification		11-28
12. EXAMPLES		12-1
Sample Deck Structures		12-1
FORTRAN Source Program With Control Statements		12-1
Compilation Only		12-1
OPT=0 Compilation		12-1
Compilation and Execution		12-1

FORTRAN Compilation With COMPASS		
Assembly and Execution		12-3
Compilation and Execution With FORTRAN Subroutine and COMPASS Subprogram		12-4
Compilation With Binary Card Output		12-4
Loading and Execution of Binary Program		12-4
Compilation and Execution With Relocatable Binary Deck		12-4
Compilation and Two Executions With Different Data Decks		12-7
Preparation of Overlays		12-7
Compilation and Two Executions With Overlays		12-8.1
Sample Programs		12-8.1
Program OUT		12-8.1
Program B		12-9
Program STATES		12-9
Program EQUIV		12-9
Program COME		12-10
Program LIBS		12-12
Program ADD		12-13
Read		12-14
Write		12-14
Program PASCAL		12-14
Program PIE		12-14
Program X		12-16
Program ADIM		12-17
Program ADIM2		12-18
Subroutine SET		12-20
Subroutine IOTA		12-20
Function PVAL		12-20
Function AVG		12-20
Function MULT		12-21
Main Program: ADIM2		12-21
Program CIRCLE		12-21
Program BOOL		12-22
Program EASY IO		12-23
Program BLOCK		12-24
Programs ONE and TWO		12-24
Program PMD2		12-25
Program PMD		12-29
Program DEBUG		12-29
Program GOTO		12-32
Program ASK		12-34
Program SCORE		12-34

APPENDIXES

A	Standard Character Sets	A-1
B	FORTRAN Diagnostics	B-1
C	Glossary	C-1
D	Language Summary	D-1
E	C\$ Directives	E-1
F	Input/Output Implementation	F-1
G	Future System Migration Guidelines	G-1

INDEX

FIGURES

1-1	Program on FORTRAN Coding Form	1-1
1-2	Normal Column Usage	1-2
1-3	Listing of Sequenced Program	1-3
1-4	Sequenced Column Usage	1-3
1-5	Duplicated Keyword Names Examples	1-4
1-6	Integer Constant	1-4
1-7	Integer Constant Examples	1-4
1-8	Real Constant	1-5
1-9	Real Constants Examples	1-5
1-10	Double Precision Constant	1-5
1-11	Double Precision Constants Examples	1-6
1-12	Complex Constant	1-6

1-13	Complex Constants Examples	1-6	4-6	Examples of ASSIGN Statement	4-2
1-14	Logical Constant	1-6	4-7	Assigned GO TO Statement	4-2
1-15	Logical Constants Examples	1-6	4-8	Example of Assigned GO TO Statement	4-2
1-16	Hollerith Constant	1-7	4-9	Arithmetic IF Statement	4-3
1-17	Octal Constant	1-7	4-10	Example of Arithmetic IF Statement	4-3
1-18	Hexadecimal Constant	1-7	4-11	Logical IF Statement	4-3
1-19	Character Constant	1-7	4-12	Examples of Logical IF Statements	4-3
1-20	Character Constants Examples	1-8	4-13	Block IF Statement	4-3
1-21	Declaration of Array Dimensions	1-9	4-14	ELSE Statement	4-4
1-22	1-Dimensional Array Storage	1-9	4-15	ELSE IF Statement	4-4
1-23	2-Dimensional Array Storage	1-9	4-16	END IF Statement	4-4
1-24	3-Dimensional Array Storage	1-10	4-17	Simple Block IF Structure	4-4
1-25	Array Element Reference	1-10	4-18	Example of Block IF Statement	4-4
1-26	Examples of References to Array Elements	1-11	4-19	Block IF Structure With ELSE Statement	4-4
1-27	Character Substring Reference	1-11	4-20	Example of Block IF Structure With ELSE Statement	4-5
1-28	Examples of Character Substring References	1-12	4-21	Block IF Structure With ELSE IF Statements	4-5
1-29	Example of Substring and Array References	1-12	4-22	Example of Block IF Structure With ELSE IF Statements	4-5
1-30	Statement Order	1-13	4-23	Nested Block IF Structure	4-5
2-1	COMMON Statement	2-1	4-24	Example of Nested Block IF Structure	4-5
2-2	Examples of COMMON Statement	2-2	4-25	DO Statement	4-6
2-3	DATA Statement	2-3	4-26	DO Loop Examples	4-7
2-4	Examples of DATA Statement	2-4	4-27	Branch to Shared Terminal Statement	4-8
2-5	Examples of Implied DO List With DATA Statement	2-4	4-28	Nested DO Loops	4-8
2-6	Examples of Character Data Initialization With DATA Statement	2-5	4-29	Nested DO Loop Transfers	4-8
2-7	DIMENSION Statement	2-5	4-30	Nested DO Loop Examples	4-9
2-8	Examples of DIMENSION Statement	2-5	4-31	Nested DO Loops With Different Terminal Statements	4-9
2-9	EQUIVALENCE Statement	2-6	4-32	CONTINUE Statement	4-9
2-10	Examples of EQUIVALENCE Statement	2-6	4-33	CONTINUE Statement Examples	4-9
2-11	EXTERNAL Statement	2-7	4-34	PAUSE Statement	4-9
2-12	Examples of EXTERNAL Statement	2-7	4-35	STOP Statement	4-10
2-13	IMPLICIT Statement	2-7	4-36	END Statement	4-10
2-14	Examples of Typing With IMPLICIT Statement	2-8	5-1	Direct Access File Record Length Calculation Example	5-2
2-15	INTRINSIC Statement	2-8	5-2	Direct Access File Processing Examples	5-2
2-16	Examples of INTRINSIC Statement	2-8	5-3	Input/Output List Examples	5-3
2-17	LEVEL Statement	2-9	5-4	Implied DO in Input/Output List Examples	5-4
2-18	Example of LEVEL Statement	2-9	5-5	Format Terminating Data Read Examples	5-4
2-19	PARAMETER Statement	2-9	5-6	FORMAT Statement	5-5
2-20	Example of PARAMETER Statement	2-10	5-7	Character Format Specification Examples	5-5
2-21	Intrinsic Functions Permitted in Extended Constant Expressions	2-10	5-8	Complex Data Item Format Specification Example	5-7
2-22	SAVE Statement	2-10	5-9	A Input Examples	5-8
2-23	Example of SAVE Statement	2-11	5-10	Apostrophe and Quote Descriptor Examples	5-8
2-24	Examples of Explicit Typing	2-11	5-11	Printer Carriage Control Examples	5-9
2-25	INTEGER Statement	2-12	5-12	Carriage Control Program Example	5-10
2-26	REAL Statement	2-12	5-13	D Input Field	5-10
2-27	DOUBLE PRECISION Statement	2-12	5-14	D Output Example	5-10
2-28	COMPLEX Statement	2-12	5-15	E Input Field	5-11
2-29	LOGICAL Statement	2-12	5-16	Example Showing E Input Incorrectly Read	5-12
2-30	BOOLEAN Statement	2-13	5-17	Integer Variable Ew.d Specification Examples	5-13
2-31	CHARACTER Statement	2-13	5-18	End-of-Record Slash Examples	5-13
2-32	Examples of CHARACTER Statement	2-13	5-19	G Output Examples	5-16
3-1	Arithmetic Expression	3-1	5-20	H Descriptor Example	5-16
3-2	Examples of Integer Constant Expressions	3-2	5-21	I Input Example	5-16
3-3	Boolean Expression	3-4	5-22	I Output Examples	5-16
3-4	Character Expression	3-5	5-23	L Output Example	5-17
3-5	Logical Expression	3-6	5-24	O Input Example	5-17
3-6	Relational Expression	3-7	5-25	Scaled F Output	5-18
3-7	Arithmetic Assignment Statement	3-9	5-26	Scaled E Output	5-18
3-8	Boolean Assignment Statement	3-9	5-27	Scaled G Output	5-18
3-9	Character Assignment Statement	3-9	5-28	R Input Example	5-19
3-10	Logical Assignment Statement	3-9	5-29	Sign Control Example	5-19
3-11	Multiple Assignment Statement	3-10	5-30	T, TL, and TR Descriptor Examples	5-19
4-1	Unconditional GO TO Statement	4-1	5-31	T Output Example	5-20
4-2	Example of Unconditional GO TO Statement	4-1			
4-3	Computed GO TO Statement	4-1			
4-4	Examples of Computed GO TO Statements	4-2			
4-5	ASSIGN Statement	4-2			

5-32	Termination by Colon Example	5-21	6-20	Using Common	6-12
5-33	X Descriptor Example	5-21	6-21	Block DATA Statement	6-13
5-34	Z Input Example	5-21	6-22	Example of BLOCK DATA	6-13
5-35	Repetition Factor Examples	5-23	7-1	IOCF Result for Character Argument	7-11
5-36	Execution Time Format Specification Examples	5-24	7-2	GETPARM Call	7-13
5-37	Formatted READ Statement	5-25	7-3	RANSET Call	7-14
5-38	Formatted READ Statement Examples	5-25	7-4	RANGET Call	7-14
5-39	Formatted WRITE Statement	5-26	7-5	DATE Function	7-14
5-40	Formatted WRITE Statement Example	5-26	7-6	JDATE Function	7-14
5-41	Formatted PRINT Statement	5-26	7-7	TIME Function	7-14
5-42	Formatted PRINT Example	5-26	7-8	CLOCK Function	7-14
5-43	Formatted PUNCH Statement	5-26	7-9	DISPLA Call	7-15
5-44	Formatted PUNCH Example	5-26	7-10	REMARK Call	7-15
5-45	Unformatted READ Statement	5-27	7-11	SSWTCH Call	7-15
5-46	Unformatted READ Example	5-27	7-12	EXIT Call	7-15
5-47	Unformatted WRITE Statement	5-27	7-13	CHEKPTX Call	7-16
5-48	Unformatted WRITE Example	5-27	7-14	CHEKPTX Example	7-16
5-49	List Directed READ Statement	5-27	7-15	RECOVER Call	7-17
5-50	List Directed Input Examples	5-28	7-16	UNIT Function	7-18
5-51	List Directed WRITE Statement	5-29	7-17	EOF Function	7-18
5-52	List Directed PRINT Statement	5-29	7-18	IOCHEC Function	7-18
5-53	List Directed PUNCH Statement	5-29	7-19	LENGTH Function	7-19
5-54	List Directed Output Examples	5-29	7-20	LENGTHX Call	7-19
5-55	NAMelist Statement	5-30	7-21	MOVLEV Call	7-19
5-56	NAMelist Example	5-30	7-22	MOVLCH Call	7-20
5-57	NAMelist READ Statement	5-31	7-23	CONNEC Call	7-20
5-58	NAMelist Group Format	5-31	7-24	DISCON Call	7-21
5-59	NAMelist Group Input Example	5-31	7-25	LABEL Call	7-21
5-60	NAMelist WRITE Statement	5-32	7-26	OPENMS Call	7-22.1
5-61	NAMelist PRINT Statement	5-32	7-27	WRITMS Call	7-23
5-62	NAMelist PUNCH Statement	5-32	7-28	READMS Call	7-23
5-63	NAMelist Output Example	5-32	7-29	CLOSMS Call	7-23
5-64	NAMelist WRITE and Subsequent NAMelist READ Example	5-32	7-30	STINDX Call	7-24
5-65	Arrays in NAMelist Examples	5-33	7-31	Random File With Number Index	7-25
5-66	BUFFER IN Statement	5-35	7-32	Random File With Name Index	7-26
5-67	BUFFER IN Example	5-35	7-33	Subindexed File With Number Index	7-26
5-68	BUFFER OUT Statement	5-35	7-34	DUMP Call	7-27
5-69	OPEN Statement	5-36	7-35	PDUMP Call	7-27
5-70	OPEN Overriding PROGRAM Declaration Example	5-37	7-36	STRACE Call	7-27
5-71	OPEN Statement Examples	5-37	7-37	LEGVAR Function	7-27
5-72	CLOSE Statement	5-38	7-38	SYSTEM Call	7-27
5-73	INQUIRE Statement	5-39	7-39	SYSTEMC Call	7-28
5-74	INQUIRE Statement Example	5-40	7-40	Error Table Entry	7-29
5-75	Internal File Input Examples	5-41	7-41	Suppressing an Error Message	7-29
5-76	Internal File Output Examples	5-41	7-42	LIMERR Call	7-30
5-77	ENCODE Statement	5-42	7-43	NUMERR Function	7-30
5-78	ENCODE Statement Examples	5-42	7-44	Suppressing Fatal Termination	7-30
5-79	DECODE Statement	5-42	7-45	COLSEQ Call	7-32
5-80	DECODE Statement Examples	5-43	7-46	WTSET Call	7-32
5-81	REWIND Statement	5-43	7-47	CSOWN Call	7-32
5-82	BACKSPACE Statement	5-44	8-1	FILE Control Statement	8-6
5-83	ENDFILE Statement	5-44	8-2	Example of Creating Indexed Sequential File Having Alternate Keys	8-16
6-1	PROGRAM Statement	6-3	8-3	Sort/Merge 5 Owncode Subroutine	8-21
6-2	Examples of PROGRAM Statement	6-3	8-4	Example of CMM Use	8-28
6-3	File Equivalencing Example	6-4	8-5	COMPASS IDENT Statement	8-28
6-4	Subroutine Statement	6-4	8-6	Program SUBLNK and Function ZEUS	8-30
6-5	Subroutine Call Example	6-4	8-7	Object Listing for Program SUBLNK	8-31
6-6	CALL Statement	6-5	8-8	Object Listing for Function ZEUS	8-32
6-7	FUNCTION Statement	6-5	8-9	DML Preprocessor Control Statement	8-35
6-8	Function Reference Example	6-5	8-10	IML Preprocessor Control Statement	8-36
6-9	Function Reference Name	6-6	8-11	Execution-Time Parameters for IMF	8-36
6-10	ENTRY Statement	6-6	9-1	Overlay Positioning	9-1
6-11	Examples of ENTRY Statements	6-7	9-2	Overlay Positioning Showing Common	9-3
6-12	END Statement	6-7	9-3	OVERLAY Statement	9-4
6-13	RETURN Statement	6-7	9-4	OVERLAY Call	9-4
6-14	MULTIPLE Return Example	6-7	9-5	Sample Overlay Structure	9-5
6-15	Alternate Return Example	6-8	9-6	OVCAP Directive	9-5
6-16	Statement Function	6-8	9-7	LOVCAP Call	9-6
6-17	Examples of Statement Functions	6-9	9-8	XOVCAP Call	9-6
6-18	Statement Function Reference	6-9	9-9	UOVCAP Call	9-6
6-19	Example of Adjustable Dimensions in Subroutine	6-12	9-10	Batch Job Set Up for OVCAPS	9-6
			10-1	PMDARRAY Call	10-5
			10-2	PMDDUMP Call	10-5
			10-3	PMDLOAD Call	10-6

10-4	PMDSTOP Call	10-6	12-43	Hollerith Examples	12-23
11-1	FTN5 Control Statement	11-1	12-44	Program EASYIO	12-24
11-2	Possible OPT=3 Error Example 1	11-8	12-45	Sample Input and Output for Program EASYIO	12-25
11-3	Possible OPT=3 Error Example 2	11-8			
11-4	Optimization Example 1	11-9	12-46	Program BLOCK	12-25
11-5	Variable Map	11-14	12-47	Sample Input and Output for Program BLOCK	12-26
11-6	Symbolic Constants Map	11-15			
11-7	Procedures Map	11-16	12-48	Programs ONE and TWO	12-26
11-8	Statement Label Map	11-17	12-49	Program PMD2	12-27
11-9	Entry Point Map	11-17	12-50	Post Mortem Dump Output for Program PMD2	12-28
11-10	Input/Output Unit Map	11-18			
11-11	Namelist Map	11-18	12-51	Program PMD	12-29
11-12	DO Loop Map	11-18	12-52	Post Mortem Dump Output for Program PMD	12-30
11-13	Common Equivalence Map	11-19			
11-14	Program Statistics Map	11-20	12-53	Program DBUG	12-32
11-15	Program MAPS	11-21	12-54	Debug Session	12-32
11-16	Reference Map Example	11-22	12-55	Program GOTO	12-33
12-1	FORTTRAN Source Program With Control Statements	12-1	12-56	Sample Input and Output for Program GOTO	12-33
12-2	Compilation Only	12-2	12-57	Program ASK	12-34
12-3	OPT=0 Compilation	12-2	12-58	Program ASK Output	12-34
12-4	Compilation and Execution	12-3	12-59	Program SCORE and Subroutine AVG	12-35
12-5	Compilation With COMPASS Assembly and Execution	12-3	12-60	Sample Input and Output for Program SCORE	12-35
12-6	Compilation and Execution With FORTRAN Subroutines and COMPASS Subprogram	12-4			
12-7	Compilation With Binary Card Output	12-5			
12-8	Loading and Execution of Binary Program	12-5			
12-9	Compilation and Execution With Relocatable Binary Deck	12-6			
12-10	Compilation and Execution With Different Data Decks	12-7			
12-11	Preparation of Overlays	12-8			
12-12	Compilation and Two Executions With Overlays	12-8.1			
12-13	Program OUT With Control Statements	12-8.1			
12-14	Program OUT Output	12-8.2			
12-15	Program B	12-9			
12-16	Program B Output	12-9			
12-17	Program STATES	12-9			
12-18	Sample Input and Output for Program STATES	12-10			
12-19	Program EQUIV	12-10			
12-20	INTEGER and REAL Internal Formats	12-10			
12-21	Program EQUIV Output	12-11			
12-22	Program COME	12-11			
12-23	Storage Layout for Variables in Program COME	12-11			
12-24	Program COME Output	12-11			
12-25	Program LIBS	12-12			
12-26	Program LIBS Output	12-13			
12-27	Program ADD	12-13			
12-28	Program ADD Input and Output	12-15			
12-29	Program PASCAL	12-15			
12-30	Program PASCAL Output	12-16			
12-31	Program PIE and Output	12-16			
12-32	Program X, Function EXTRAC, Output: INTEGER Declaration Omitted From Main Program	12-17			
12-33	Program X, Function EXTRAC, Output: INTEGER Declaration Included in Main Program	12-17			
12-34	Program ADIM and Subroutine IOTA	12-18			
12-35	Program ADIM Output	12-19			
12-36	Program ADIM2	12-19			
12-37	Program ADIM2 Output	12-21			
12-38	Program CIRCLE, Function DIM, Output	12-21			
12-39	Rectangle and Circumscribed Circle	12-22			
12-40	Program Circle With Correction and Output	12-22			
12-41	Program BOOL and Output	12-22			
12-42	Program BOOL With Correction and Output	12-23			
			TABLES		
			1-1	FORTTRAN Character Set	1-2
			1-2	Array Element Position	1-11
			2-1	Correspondence of Data Types in DATA Statements	2-4
			3-1	Arithmetic Operators	3-2
			3-2	Resulting Data Type for X1**X2 Operations	3-3
			3-3	Resulting Data Type for X1+X2, X1*X2 or X1/X2 Operations	3-3
			3-4	Result of Logical Operators in Boolean Expressions	3-5
			3-5	Character Operator	3-5
			3-6	Logical Operators	3-6
			3-7	Result of Logical Operators	3-7
			3-8	Relational Operators	3-7
			5-1	Repeatable Edit Descriptors	5-6
			5-2	Nonrepeatable Edit Descriptors	5-6
			5-3	Printer Control Characters	5-9
			5-4	Ew.d Input Examples	5-12
			5-5	F Input Examples	5-14
			5-6	F Output Examples	5-15
			6-1	Characteristics of Program Components	6-2
			7-1	Intrinsic Functions	7-2.1
			7-2	Summary of Mathematical Intrinsic Functions	7-5
			7-3	LABINFO Block Content	7-21
			7-4	Information Available to Error Recovery Routine	7-30
			7-5	Collating Weight Tables	7-31
			7-6	Static Capsule Loading Routines	7-33
			8-1	PF Subroutine Parameters	8-3
			8-2	Meanings of PF Command Options	8-4
			8-3	CYBER Record Manager FIT Fields	8-7
			8-4	CYBER Record Manager Calls	8-14
			8-5	CYBER Record Manager Utilities	8-17
			8-6	Sort/Merge 5 Calls	8-19
			8-7	Sort/Merge 5 Numeric Key Types	8-21
			8-8	Sort/Merge 5 Owncode Routines	8-22
			8-9	Sort/Merge 4 and 1 Calls	8-23
			8-10	Common Memory Manager Calls	8-25
			8-11	Argument List Format	8-29
			8-12	8-Bit Subroutine Calls	8-33
			8-13	DML Statements and Utility Calls	8-34
			8-14	IML Statements	8-37
			8-15	Queued Terminal Record Manager Calls	8-38
			11-1	Defaults for FTN5 Control Statement	11-2

NOTATIONS

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

UPPERCASE In language syntax, uppercase indicates a statement keyword or character that is to be written as shown.

Lowercase In language syntax, lowercase indicates a name, number, symbol, or entity that is to be supplied by the programmer.

[] In language syntax, brackets indicate an optional item that can be used or omitted.

{ } In language syntax, braces indicate that only one of the vertically stacked items can be used.

... In language syntax, an ellipsis indicates that the preceding optional item in brackets can be repeated as necessary.

.
. In program examples, a vertical ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.

Δ A delta indicates a blank character.

Shading In language syntax, language descriptions, and program examples, shading indicates extensions to FORTRAN 77.

LANGUAGE ELEMENTS

This section discusses the language elements of FORTRAN 5. These elements include: FORTRAN statements, symbolic names, constants, variables, arrays, character substrings, and statement order.

FORTRAN STATEMENTS

FORTRAN statements are written using the FORTRAN character set. The FORTRAN character set consists of 26 letters, 10 digits, and 13 or 14 special characters. The FORTRAN character set is shown in table 1-1.

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

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

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

NONSEQUENCED MODE

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

STATEMENT NO.	C O M M E N T	FORTRAN STATEMENT	SERIAL NUMBER
PROGRAM		NAME	
ROUTINE		DATE	PAGE OF
1	PROGRAM	PASCAL	
2	THIS PROGRAM PRODUCES A PASCAL TRIANGLE WITH 15 ROWS		
3	INTEGER	LR0W(15)	
4	DO	10, I=1, 15	
5	LR0W(I) = 1		
6	PRINT	' ("1 PASCAL TRIANGLE" /1X, I5, /1X, 2I5) ', LR0W(15)	
7	* LR0W(14), LR0W(15)		
8	DO	50 J = 14, 2, -1	
9	DO	40 K = J, 14	
10	LR0W(K) = LR0W(K) + LR0W(K+1)		
11	PRINT	' (1X, 15I5) ', (LR0W(M), M=J-1, 15)	
12	CONTINUE		
13	STOP		
14	END		

Figure 1-1. Program on FORTRAN Coding Form

TABLE 1-1. FORTRAN CHARACTER SET

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

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

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

Initial Lines

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

Continuation Lines

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

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

Statement Labels

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

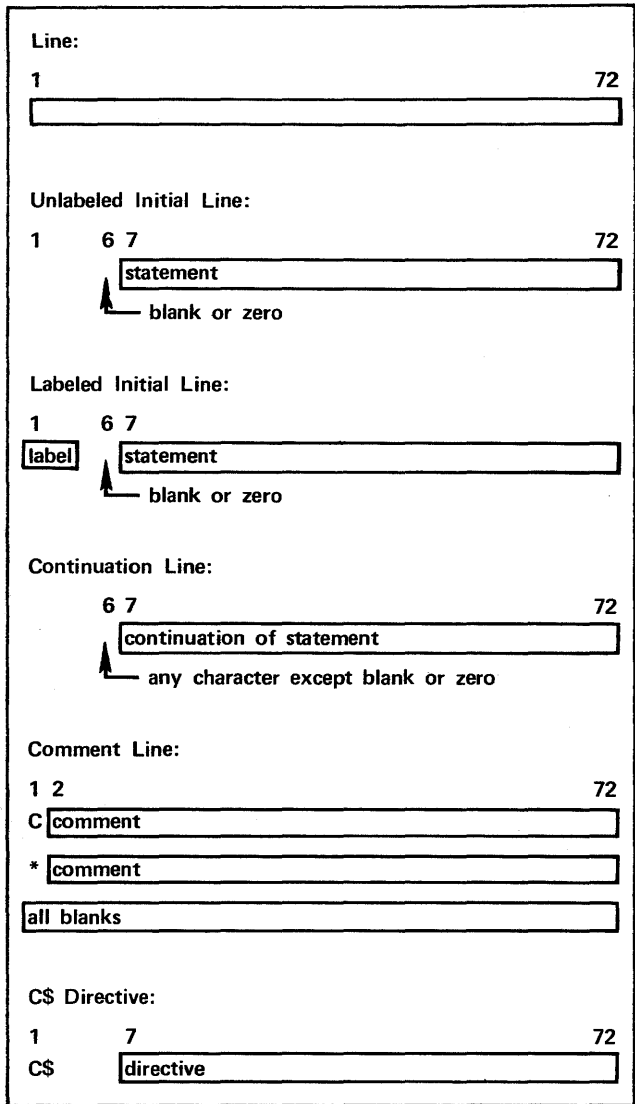


Figure 1-2. Normal Column Usage

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

Comment Lines

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

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

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

Compiler Directive Lines

The two characters C\$ in columns 1 and 2 indicate a compiler directive line. Compiler directives control interpretation of following lines; they allow, for example, conditional compilation listing control. A compiler directive must appear on a single line, and any compiler directive terminates statement continuation.

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

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

Columns 73 Through 80

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

SEQUENCED MODE

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

```

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

```

Figure 1-3. Listing of Sequenced Program

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

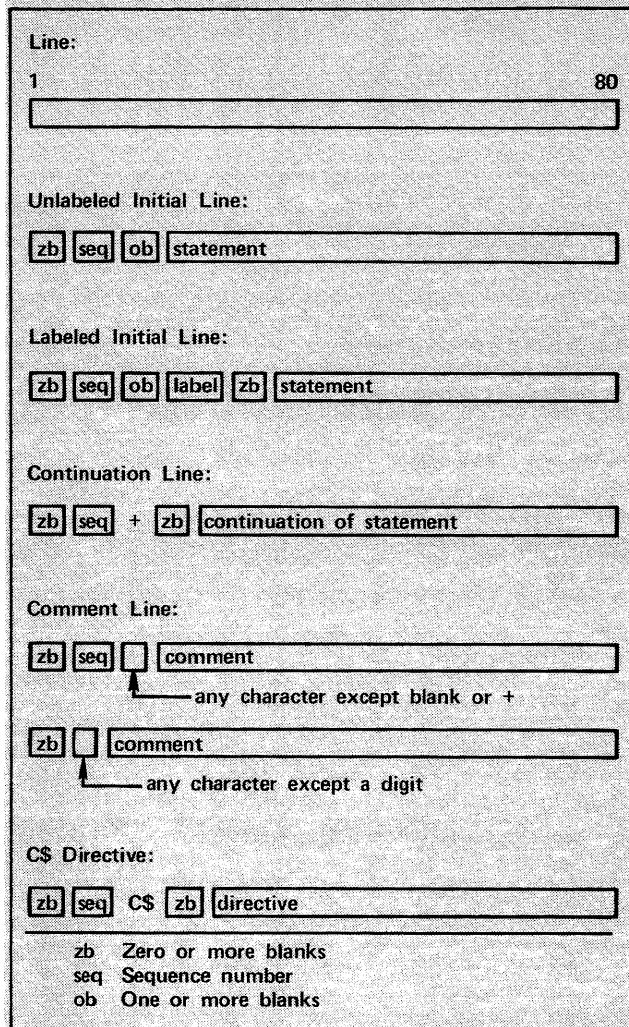


Figure 1-4. Sequenced Column Usage

A line consists of characters in columns 1 through 80. The sequence number of a sequenced line must appear to the left of all other nonblank characters in the line. The sequence number consists of one to five digits, usually at the beginning of the line. Blanks can precede the sequence number.

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

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

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

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

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

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

SYMBOLIC NAMES

Symbolic names are assigned by the user. They consist of one to seven letters and digits (ANSI only allows 6) beginning with a letter. Symbolic names are used for the following:

- Main program name
- Common block name
- Subroutine name
- External function name
- Block data subprogram name
- Variable name
- Array name
- Symbolic constant name
- Intrinsic function name
- Statement function name
- Dummy procedure name
- NAMELIST group name

Names that are FORTRAN keywords can be used as user-assigned symbolic names without conflict. In general, however, it is good programming practice to avoid naming conflicts by assigning unique names to program entities. Certain of these conflicts are illegal and are diagnosed. See figure 1-5 for an example of duplicated keyword names.

CONSTANTS

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

Example 1:

```
PROGRAM TEST
PRINT = 1.0
PRINT*, PRINT
.
```

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

Example 2:.

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

The name ALPHA is illegally used as a program unit name and a variable name.

Figure 1-5. Duplicated Keyword Names Examples

INTEGER CONSTANT

An integer constant (figure 1-6 with examples in figure 1-7) is a string of 1 to 18 decimal digits written without a decimal point. It can be positive, negative, or zero. If the integer is positive, the plus sign can be omitted; if it is negative, the minus sign must be present. An integer constant must not contain a comma. The range of an integer constant is $-(2^{59}-1)$ to $2^{59}-1$ ($2^{59}-1=576460752303423487$). For values outside this range, the high-order bits are lost and no diagnostic is provided.

[±] d[d]...

d Is a decimal digit.

Figure 1-6. Integer Constant

Examples of valid integer constants:

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

Examples of invalid integer constants:

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

Figure 1-7. Integer Constant Examples

Integers used in multiplication, division, and exponentiation, whether constant or variable, should be in the range $-(248-1)$ to $248-1$ ($248-1=281474976710655$). The result of such operations must also be in this range. For integer addition and subtraction (where both operands are integers), the full 60-bit word is used.

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

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

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

When values are converted (in an expression or assignment statement) from real to integer or from integer to real, the valid range is also from $-(248-1)$ to $248-1$. For values outside this range, the high order bits are lost and no diagnostic is provided.

REAL CONSTANT

A real constant (figure 1-8 with examples in figure 1-9) consists of a string of decimal digits written with a decimal point, with an exponent, or with both. Commas are not allowed. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

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

Figure 1-8. Real Constant

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

Optionally, a real constant can be followed by a decimal exponent, written as the letter E and an integer constant that indicates the power of ten by which the number is to be multiplied. If the E is present, the integer constant following the letter

Examples of valid real constants:	
7.5	
-3.22	
+4000.	
.5	
Examples of invalid real constants:	
33,500.	Comma not allowed
2.5A	Letter not allowed
Examples of valid real constants with exponents:	
42.E1	Value $42. \times 10^1 = 420.0$
.00028E+5	Value $.00028 \times 10^5 = 28.0$
6.205E6	Value $6.205 \times 10^6 = 6205000.0$
700.E-2	Value $700. \times 10^{-2} = 7.0$
Example of invalid real constant with exponent:	
7.2E3.4	Exponent not an integer

Figure 1-9. Real Constants Examples

E must not be omitted. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

DOUBLE PRECISION CONSTANT

A double precision constant (figure 1-10 and examples in figure 1-11) is written in the same way as a real constant with exponent, except that the exponent is prefixed by the letter D instead of E. Double precision values are represented internally by two computer words, giving additional precision. A double precision constant is accurate to approximately 29 decimal digits. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

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

Figure 1-10. Double Precision Constant

Examples of valid double precision constants:

5.834D2	Value $5.834 \times 10^2 = 583.4$
14.D-5	Value $14. \times 10^{-5} = .00014$
9.2D03	Value $9.2 \times 10^3 = 9200.0$
3120D4	Value $3120. \times 10^4 = 31200000.0$

Examples of invalid double precision constants:

7.2D	Exponent missing
D5	Exponent alone not allowed
2,001.3D2	Comma illegal
3.14159265	D and exponent missing

Figure 1-11. Double Precision Constants Examples

COMPLEX CONSTANT

Complex constants (figure 1-12 and examples in figure 1-13) are written as a pair of real or integer constants **or symbolic constants** separated by a comma and enclosed in parentheses.

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

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

LOGICAL CONSTANT

A logical constant (figure 1-14 and examples in figure 1-15) takes the form of .TRUE. or .FALSE.. The periods are part of the constant and must appear.

BOOLEAN CONSTANTS

NOTE

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

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

(real,imag)

real	Is a real or integer constant for the real part.
imag	Is a real or integer constant for the imaginary part.

Figure 1-12. Complex Constant

Examples of valid complex constants:

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

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

Examples of invalid complex constants:

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

Figure 1-13. Complex Constants Examples

.TRUE.
.FALSE.

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

Figure 1-14. Logical Constant

Examples of valid logical constants:

.TRUE.
.FALSE.

Examples of invalid logical constants:

.TRUE	No terminating period
.F.	Abbreviation not recognized

Figure 1-15. Logical Constants Examples

Hollerith and Extended Hollerith Boolean Constant

A Hollerith constant (figure 1-16) has one of four forms.

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

Figure 1-16. Hollerith Constant

For the nHs form, the n specifies the number of characters in the string following the H. No more than 10 characters can be specified in the string except when used as an actual argument to an external subroutine or function; extra characters are truncated. Blanks are significant, and characters that are not in the FORTRAN character set can be used.

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

```
2HAB Value 010255...55 octal
```

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

```
L"AB" Value 010200...00 octal
```

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

```
R"AB" Value 00...000102 octal
```

The "s" form is equivalent to the nH form except the characters need not be counted. No more than 10 characters can be represented in the string except when used as an actual argument to an external subroutine or function. Any quote within the string is represented by two consecutive quote characters. Note that the string might be 11 characters long if one character is a quote represented by two consecutive quotes. Blanks are significant, and characters that are not in the FORTRAN character set can be used. For example,

```
"AB" Value 010255...55 octal
"C""D" Value 03640455...55 octal
```

An extended Hollerith constant can have any of the forms shown in figure 1-16 and may have a string length greater than 10 characters. An extended Hollerith constant can only be used as an actual argument to an external procedure reference.

Octal Boolean Constant

An octal constant (figure 1-17) allows a string of digits to be interpreted as an octal number. As many as 20 octal digits can be represented in a 60-bit computer word. (An octal digit is one of the digits 0 through 7.) The octal number is right-justified with binary zero fill.

Example:

```
0"77" Value 00...0077 octal
```

<pre>O"o"</pre>
<p>o Is a string of 1 through 20 octal digits.</p>

Figure 1-17. Octal Constant

Hexadecimal Boolean Constant

A hexadecimal constant (figure 1-18) allows a string of digits to be interpreted as a hexadecimal number. As many as 15 hexadecimal digits can be represented in a 60-bit computer word. (A hexadecimal digit is one of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F.) The hexadecimal number is right-justified with binary zero fill.

Example:

```
Z"1A" Value 00...0032 octal
```

<pre>Z"z"</pre>
<p>z Is a string of 1 through 15 hexadecimal digits.</p>

Figure 1-18. Hexadecimal Constant

CHARACTER CONSTANT

A character constant (figure 1-19 with examples in figure 1-20) is a string of characters enclosed in apostrophes. Within the character string, an apostrophe is represented by two consecutive apostrophes.

<pre>'s'</pre>
<p>s Is a string of characters.</p>

Figure 1-19. Character Constant

The minimum number of characters in a character constant is one, and the maximum number of characters in a character constant is $(2^{15}-1)$ or 32767. The length is the number of characters in the string. Blanks are significant in a character constant. Any characters in the operating system character set can be used.

```

Examples of valid character constants:

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

Examples of invalid character constants:

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

```

Figure 1-20. Character Constants Examples

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

VARIABLES

A variable represents a quantity with a value that can be changed repeatedly during program execution. Variables are identified by a symbolic name of one to seven letters or digits (ANSI only allows six), beginning with a letter. A variable is associated with a storage location. Whenever a variable is used, it references the value currently in that location. A variable must be defined before being referenced for its value.

The types of variables are integer, real, double precision, complex, Boolean, logical, and character. Variables typed by default are integer if the first letter is I, J, K, L, M, or N, and are real if the first letter is any other letter. Implicit and explicit typing of variables is described in section 2.

INTEGER VARIABLE

An integer variable is a variable that is typed explicitly, implicitly, or by default as integer. An integer variable occupies one storage word. The range restrictions for integer variables are the same as for integer constants.

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

Example:

```

ITEM1  N72
J      K2S04
NSUM   JSUM

```

All variables are integer by default, as the variable name begins with the letter I through N.

REAL VARIABLE

A real variable is a variable that is typed explicitly, implicitly, or by default as real. The

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

Example:

```

AVAR    SUM3
RESULT  TOTAL2
BETA    XXXX

```

All variables are type real, as the variable names begin with letters other than the letters I through N.

DOUBLE PRECISION VARIABLE

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

Example:

```

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

```

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

COMPLEX VARIABLE

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

Example:

```

COMPLEX ZETA, MU, LAMBDA

```

LOGICAL VARIABLE

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

Example:

```

LOGICAL L33, PRAVDA, VALUE

```

BOOLEAN VARIABLE

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

Example:

```

BOOLEAN HVAL, ZZZ, R34

```

CHARACTER VARIABLE

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

Example:

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

ARRAYS

A FORTRAN array is a set of elements identified by a single name. The name is composed of one to seven letters and digits (ANSI only allows six) and begins with a letter. Each array element is referenced by the array name and a subscript.

The type of the array elements is determined by the array name in the same manner as the type of a variable is determined by the variable name. The array name can be typed explicitly with a type statement, implicitly with an IMPLICIT statement, or by default typing.

The array name and its dimensions must be declared in a DIMENSION, COMMON, or type statement. When an array is declared, the declaration of array dimensions takes the form shown in figure 1-21. Arrays can have 1 to 7 dimensions.

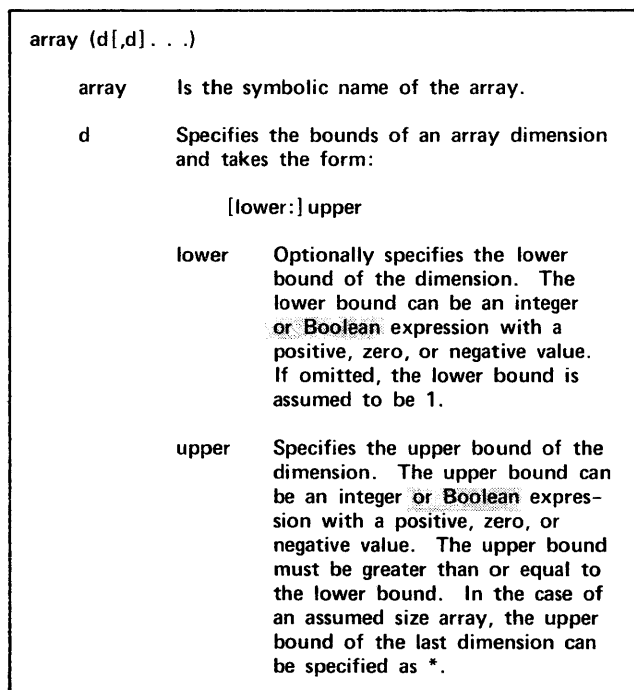


Figure 1-21. Declaration of Array Dimensions

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

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

declares a 1-dimensional array of six elements such as that shown in figure 1-22.

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

declares a 2-dimensional array of four rows and three columns, for a total of twelve elements such as that shown in figure 1-23.

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

declares a 3-dimensional array of six rows, six columns and three planes, for a total of 108 elements.

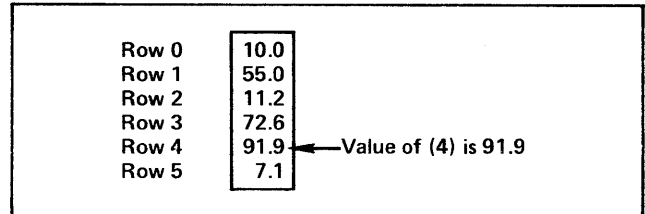


Figure 1-22. 1-Dimensional Array Storage

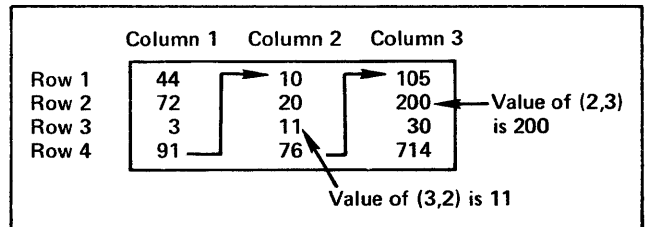


Figure 1-23. 2-Dimensional Array Storage

The span of an array dimension is given by (upper-lower+1), where upper is the upper dimension bound and lower is the lower dimension bound. An array of type integer, Boolean, real, or logical occupies n words of storage, where n is the product of the spans of all dimensions. An array of type complex or double precision occupies 2*n words. An array of type character occupies (n*len+offset+9)/10 words, where len is the length in characters of an array element, and offset is the starting character position (0 to 9) of the array in the first word of the array storage. Character operations involve character lengths, not word lengths. The length, in words, of a character array can be determined by placing the array in a common block and applying the above formula, (n*len+offset+9)/10. For a character array in a common block, the value of offset is MOD(totch,10), where totch is the total number of characters preceding the first element of the array in the common block. Note that in the formula for the length of a character array in words, all terms are integers and division is an integer division (truncation).

An array in central memory must occupy less than 2¹⁷ words. An array in extended memory can occupy up to 2²⁰-8 words if LCM=G is selected on the FTN5 control statement. Arrays are placed in extended memory through the LEVEL statement. (See section 2.)

If a Boolean expression is used for the lower or upper bound of a dimension, the value of the expression is converted to integer; that is, the

value is INT (lower) or INT (upper). A dimension bounds specification must not include a function reference or array element reference. Presence of a variable makes the size of the array adjustable. Presence of an asterisk as the upper bound of the last dimension makes the array an assumed-size array. An assumed-size array can only be used in a subroutine or function, as described in section 6.

ARRAY STORAGE

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

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

Storage patterns for a 1-dimensional, 2-dimensional, and 3-dimensional array are shown in figure 1-22, figure 1-23, and figure 1-24, respectively. Arithmetic values are shown for the array elements, but an array can be any data type. Array elements are stored in ascending locations by columns. The first subscript value increases most rapidly, and the last subscript value increases least rapidly.

ARRAY REFERENCES

Array references can be references to complete arrays or to specific array elements. A reference to a complete array is simply the array name. A reference to a specific element involves the array

name followed by a subscript specification. An array element reference is also called a subscripted array name.

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

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

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

A reference to an array element references a specific element and takes the form shown in figure 1-25.

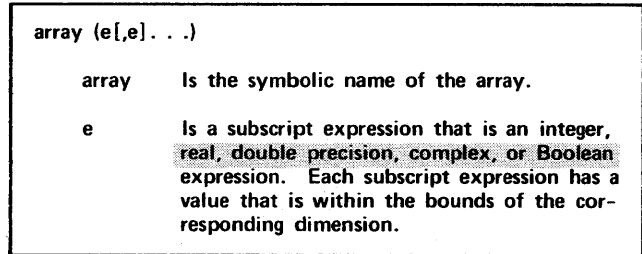


Figure 1-25. Array Element Reference

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

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

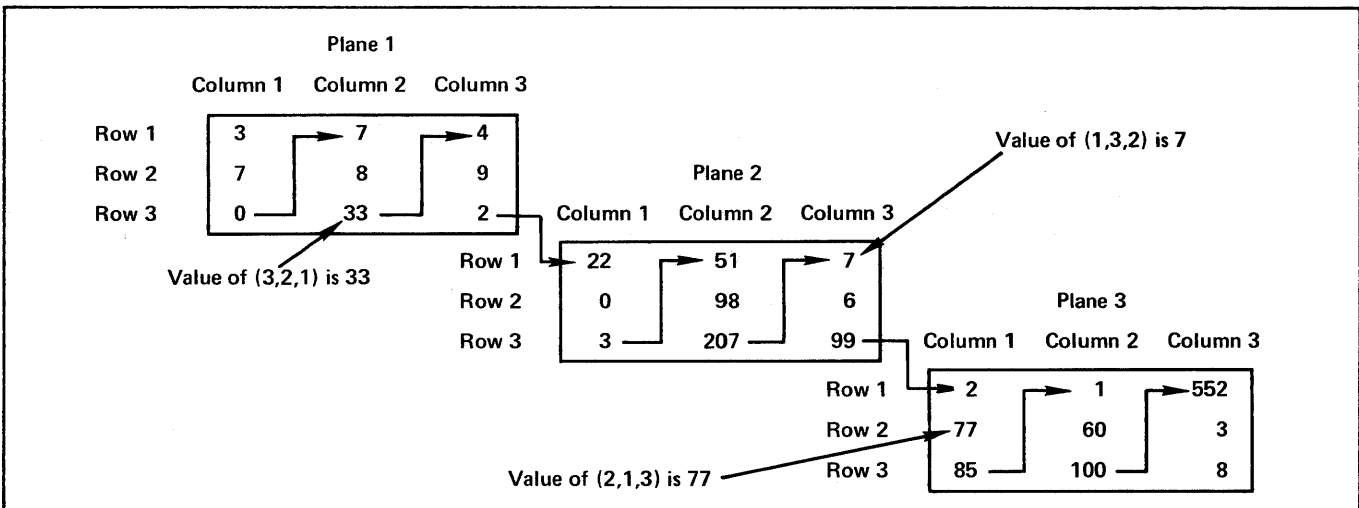


Figure 1-24. 3-Dimensional Array Storage

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

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

See figure 1-26 for array reference examples.

TABLE 1-2. ARRAY ELEMENT POSITION

Dimensions	Position of Array Element
1	$1 + (s_1 - j_1)$
2	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1$
3	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1 + (s_3 - j_3) * n_2 * n_1$
.	.
.	.
.	.
7	$1 + (s_1 - j_1) + (s_2 - j_2) * n_1 + (s_3 - j_3) * n_2 * n_1 + (s_4 - j_4) * n_3 * n_2 * n_1 + (s_5 - j_5) * n_4 * n_3 * n_2 * n_1 + (s_6 - j_6) * n_5 * n_4 * n_3 * n_2 * n_1 + (s_7 - j_7) * n_6 * n_5 * n_4 * n_3 * n_2 * n_1$
j_i	Lower bound of dimension i .
k_i	Upper bound of dimension i .
n_i	Size of dimension i . If the lower bound is one, $n_i = k_i$. Otherwise, $n_i = (k_i - j_i + 1)$.
s_i	Value of the subscript expression specified for dimension i .

CHARACTER SUBSTRINGS

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

SUBSTRING REFERENCES

If the name of a character entity is used in a reference, the value is the current value of the entire string. A reference to part of a string is written as a character substring (figure 1-27 and example in figure 1-28).

Example 1:

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

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

Example 2:

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

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

Figure 1-26. Examples of References to Array Elements

char ([first]:[last])

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

Figure 1-27. Character Substring Reference

The specification of the first character in the substring is an integer, **real, double precision, complex, or Boolean** expression that is evaluated and converted as necessary to integer. The expression can contain array element references and function references, but evaluation of a function reference must not alter the value of the other expression in the substring reference. If the specification of first is omitted, the value is 1 and all characters from 1 to the value of the specification of last are included in the substring.

Establish a character entity as follows:

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

Example of string reference:

```
S2 = S1
```

The reference to S1 is a reference to the full string 'STRING'.

Examples of substring references:

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

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

Figure 1-28. Examples of Character Substring References

The specification of last in the substring is an expression subject to the same rules as the specification of first. If last is omitted, the value is the length of the string and all characters from the specified first position to the end of the string are included in the substring. For a string length len, the value of first must be at least 1 and must not exceed last; the value of last must not exceed the value of len.

SUBSTRINGS AND ARRAYS

If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element followed by specification of the substring. See figure 1-29 for example.

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

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

Figure 1-29. Example of Substring and Array References

STATEMENT ORDER

The order of various statements within the program unit is shown in figure 1-30. Within each group, statements can be ordered as necessary, but the groups must be ordered as shown. Statements that can appear anywhere within more than one group are shown on the right in boxes that extend vertically across more than one group.

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

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

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

FORMAT statements can appear anywhere in the program unit.

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

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

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

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

PROGRAM [†] , SUBROUTINE, FUNCTION, or BLOCK DATA			Comments and compiler directives
IMPLICIT	PARAMETER (must precede first reference)	FORMAT ^{††}	
INTEGER REAL DOUBLE PRECISION COMPLEX LOGICAL CHARACTER BOOLEAN DIMENSION EQUIVALENCE COMMON LEVEL SAVE EXTERNAL INTRINSIC	(Type specification statements) (Specification statements)		
Statement function definition ^{††}	DATA	NAMELIST ^{††} (must precede first reference)	
Assignment DO CONTINUE IF ELSE ELSEIF ENDFIF GOTO ASSIGN CALL RETURN PAUSE STOP OPEN CLOSE INQUIRE READ WRITE PRINT PUNCH BUFFER IN BUFFER OUT ENCODE DECODE REWIND BACKSPACE ENDFILE			
END			
[†] Can be preceded by an OVERLAY statement. ^{††} Cannot be used in a BLOCK DATA subprogram. ^{†††} Cannot be used in a main program or BLOCK DATA subprogram.			

Figure 1-30. Statement Order

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

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

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

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

Specification statements are nonexecutable. They are used to specify the characteristics of symbolic names used in a program. Specification statements must appear before all DATA statements, NAMELIST statements, statement function definitions, and executable statements in the program unit; otherwise, a fatal diagnostic is issued.

The specification statements are:

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

These statements are discussed in this section.

DATA statements are not specification statements, but they also are described in this section. Statements are presented in alphabetical order.

COMMON STATEMENT

The COMMON statement (figure 2-1 and example in figure 2-2) provides a means of associating entities in different program units. The use of common blocks enables different program units to define and reference the same data without using arguments, and to share storage units. Within one program unit, an entity in a common block is known by a specific name. Within another program unit, the same data can be known by a different symbolic name that is valid only within the scope of that program unit.

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

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

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

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

Figure 2-1. COMMON Statement

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

The maximum number of common blocks in an executable program, including blank common and all named common, is 500. The maximum size of each common block is 131071 storage words (for character data, 1310710 characters). The use of ECS/LCM/UEM residence and LCM=G for any common block increases the maximum possible size to 1048568 storage words (for character data, 10485680 characters).

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

Example 1:

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

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

Example 2:

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

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

Example 3:

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

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

Example 4:

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

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

Figure 2-2. Examples of COMMON Statement

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

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

DATA STATEMENT

The DATA statement (figure 2-3 and example in figure 2-4) is used to provide initial values for variables, arrays, array elements, and substrings. The DATA statement is nonexecutable and can appear anywhere after the specification statements in a program unit. Usually, DATA statements are placed after the specification statements but before the statement function definitions and executable statements.

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

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

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

Within the DATA statement, each list nlist must have the same number of items as the corresponding list clist. A one-to-one correspondence exists between the items specified by nlist and the constants specified by clist. The first item of nlist corresponds to the first constant of clist, the second item to the second constant, and so forth.

If an unsubscripted array name appears as an item in nlist, a constant in clist must be specified for each element of the array. The values of the constants are assigned according to the storage order of the array.

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

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

nlist	Is a list of names to be initially defined. Each name in the list can take the form: var array element substring dolist
var	Is a variable name.
array	Is an array name.
element	Is an array element name (that is, subscripted array name).
substring	Is a substring of a character variable or array element.
dolist	Is an implied-DO list of the form: (dlist, i = init, term [,incr])
dlist	Is a list of array element names and implied-DO lists. Subscript expressions must consist of integer constants and active control variables from DO list.
i	Is an integer variable called the implied-DO variable.
init	Is an integer constant, symbolic constant, or expression specifying the initial value, as for DO loops.
term	Is an integer constant, symbolic constant, or expression specifying the terminal value, as for DO loops.
incr	Is an integer constant, symbolic constant, or expression specifying the increment, as for DO loops.
clist	Is a list of constants or symbolic constants specifying the initial values. Each item in the list can take the form: c r*c r(c[,c...]) r((c[,c...]))
c	Is a constant or symbolic constant.
r	Is a repeat count that is an unsigned nonzero integer constant or the symbolic name of such a constant. The repeat count can repeat the value of a single constant, or can repeat the values of a list of constants enclosed in parentheses. To specify repetition of a complex constant, another set of parentheses must be used.

Figure 2-3. DATA Statement

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

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

Example 1:

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

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

Example 2:

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

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

Example 3:

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

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

Example 4:

```
BOOLEAN MASK
DATA MASK/'0"7777"/
```

Variable MASK is initially defined with the octal value 7777.

Figure 2-4. Examples of DATA Statement

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

IMPLIED DO LIST USE IN DATA STATEMENT

An implied DO list can be used as an item in nlist. See figure 2-5 for implied DO list examples.

Example 1:

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

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

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

Example 2:

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

Only the following elements of array Y are initialized to 3.14159:

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

Figure 2-5. Examples of Implied DO List
With DATA Statement

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

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

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

CHARACTER DATA INITIALIZATION

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

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

If the length of the character item in nlist is less than the length of the corresponding character constant, the additional characters in the constant are ignored.

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

See figure 2-6 for a character data initialization example.

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

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

Figure 2-6. Examples of Character Data Initialization With DATA Statement

DIMENSION STATEMENT

The DIMENSION statement (figure 2-7 and example 1 in figure 2-8) defines symbolic names as array names and specifies the bounds of each array. A maximum of seven dimensions can be specified for an array. Dummy argument arrays specified within a procedure subprogram can have adjustable dimension specifications. A further explanation of adjustable dimension specifications appears in section 6.

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

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

array Is an array name.

d Specifies the bounds of a dimension in one of the forms:

upper

lower:upper

upper Is the upper bound of the dimension and is a dimension bound expression in which all constants, symbolic constants, and variables are type integer or Boolean. A maximum of seven dimensions can be specified for an array.

lower Is the lower bound of the dimension and is a dimension bound expression in which all constants, symbolic constants, and variables are of type integer or Boolean. If only the upper bound is specified, the value of the lower bound is one.

The exact form of a dimension bound expression is shown in section 1 under Arrays.

Figure 2-7. DIMENSION Statement

Example 1:

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

These statements are equivalent to:

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

Example 2:

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

Array BETA contains 6 complex elements.

Example 3:

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

Array XR contains 5 character elements, with each element having a length of 8 characters. A reference to the third and fourth characters of the second element would be XR(1)(3:4).

Example 4:

```
PARAMETER(N=100)
DIMENSION ARR(1:N*3,0:5)
```

Array ARR is a two-dimensional array that contains 1800 elements. The value of N in the dimension bound expression for the first dimension is defined in the PARAMETER statement.

Figure 2-8. Examples of DIMENSION Statement

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

EQUIVALENCE STATEMENT

The EQUIVALENCE statement (figure 2-9 and example in figure 2-10) is used to specify the sharing of storage by two or more entities in a program unit. Equivalencing causes association of the entities that share the storage. Equivalencing associates entities within a program unit, and common blocks associate entities across program units. Equivalencing and common can interact.

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

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

Figure 2-9. EQUIVALENCE Statement

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

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

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

Variables of different data types can be equivalenced, except for character data.

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

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

would be illegal.

Example 1:

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

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

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

Example 2:

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

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

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

Example 3:

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

Different data types are associated with the equivalencing of the first storage word of each entity:

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

Figure 2-10. Examples of EQUIVALENCE Statement

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

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

would also be illegal.

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

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

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

is not legal.

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

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

is not legal, whereas:

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

can be used to extend the common block.

EXTERNAL STATEMENT

The EXTERNAL statement (figure 2-11 and example in figure 2-12) is used to identify a function name as representing an external procedure and to permit such a name to be used as an actual argument. The EXTERNAL statement specifies that a function name refers to a user-written function rather than to an intrinsic function.

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

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

Figure 2-11. EXTERNAL Statement

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

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

IMPLICIT STATEMENT

The IMPLICIT statement (figure 2-13 and example in figure 2-14) is used to change or confirm the default typing that occurs according to the first letters of the names.

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

Example 1:

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

Since the name SQRT is declared external, the function reference SQRT(X) references the user-written function SQRT rather than the intrinsic function SQRT.

Example 2:

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

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

Figure 2-12. Examples of EXTERNAL Statement

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

type Is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, CHARACTER, or CHARACTER[*len].

ac Is a single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

len Specifies the length and can be an unsigned nonzero integer constant; or an integer constant expression, enclosed in parentheses, with a positive value.

Figure 2-13. IMPLICIT Statement

Example 1:

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

Names beginning with letter M are typed as character rather than integer; names beginning with X, Y, or Z are character rather than real. Default typing is effective in all other cases.

Example 2:

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

Variable L is integer, rather than logical, because it is explicitly typed. LX is integer. The name TT is integer, because an explicit type overrides the default typing.

Figure 2-14. Examples of Typing with IMPLICIT Statement

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

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

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

INTRINSIC STATEMENT

The INTRINSIC statement (figure 2-15 and example in figure 2-16) is used to identify a name as representing an intrinsic function. The INTRINSIC statement also enables use of an intrinsic function name as an actual argument. The INTRINSIC statement specifies that a function name refers to an intrinsic function rather than a user-written function.

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

fun is an intrinsic function name.

Figure 2-15. INTRINSIC Statement

Example 1:

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

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

Example 2:

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

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

Figure 2-16. Examples of INTRINSIC Statement

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

Type conversion functions BOOL, CHAR, CMLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SNGL

Lexical relationship functions LGE, LGT, LLE, and LLT

Largest/smallest value functions AMAX0, AMAX1, AMINO, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1

Logical and masking functions AND, OR, XOR, NEQV, EQV, COMPL

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

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

LEVEL STATEMENT

The LEVEL statement (figure 2-17 and example in figure 2-18) provides a means for specifying the storage level of common blocks and dummy arguments. The storage level indicates the storage residence and mode of access for entities in a common block or for actual arguments associated with dummy arguments. Only common block names and dummy argument names can appear in a LEVEL statement. No dimension or type information can be included in the LEVEL statement.

```
LEVEL n,name[,name]...
```

n is an unsigned integer constant, or symbolic constant, with the value 0, 1, 2, or 3 indicating the storage level.

name is either a common block designator of the form /[cb]/ or a dummy argument name. If n is 0, only dummy argument names can appear.

Figure 2-17. LEVEL Statement

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

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

Figure 2-18. Example of LEVEL Statement

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

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

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

Storage level 2 indicates the following residence:

Central memory residence on CDC CYBER 170 Models 171, 172, 173, 174, and 175; CYBER 70 Models 71, 72, 73, and 74; CYBER 700 Series; and 6000 Series computers; SCM on CYBER 176 without LCM; and central memory on CYBER 170 800 Series Computer Systems without UEM.

Large central memory residence on CDC CYBER 170 Model 176, CYBER 70 Model 76, and 7600 computers. Unified extended memory on CYBER 170 800 Series Computer Systems.

Mode of access for level 2 entities is unrestricted.

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

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

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

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

For more information about storage levels, see the descriptions of the MOVLEV subroutine and the LOCF function.

PARAMETER STATEMENT

The PARAMETER statement (figure 2-19 and example in figure 2-20) is used to give a symbolic name to a constant. PARAMETER statements can be used anywhere among the specification statements, but each symbolic constant must be defined in a PARAMETER statement before the first reference to the symbolic constant.

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

p is a symbolic name.

e is a constant, constant expression, or extended constant expression.

Figure 2-19. PARAMETER Statement

```

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

```

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

Figure 2-20. Example of PARAMETER Statement

An extended constant expression can contain:

A constant

A previously-defined symbolic constant

A reference to any of the intrinsic functions shown in figure 2-21 that has extended constant expressions as arguments.

An extended constant expression enclosed in parentheses

ABS	DIM	MASK
AIMAG	DINT	MAX
AINT	DNINT	MAXO
AMAXO	DMAX1	MAX1
AMAX1	DMIN1	MIN
AMINO	DMOD	MINO
AMIN1	DPROD	MIN1
AMOD	DSIGN	MOD
AND	EQV	NEQV
ANINT	FLOAT	NINT
BOOLT	IABS	OR
CMPLX	IDIM	REAL
COMPL	IDINT	SHIFT
CONJG	IDNINT	SIGN
DABS	IFIX	SNGL
DBLE	INT	XOR
DDIM	ISIGN	

†Not character

Figure 2-21. Intrinsic Functions Permitted in Extended Constant Expressions

If a symbolic name is of type integer, real, double precision, complex, or Boolean, the corresponding expression must be an arithmetic or Boolean constant expression or an extended constant expres-

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

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

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

In an expression in any subsequent statement

In a DATA statement as an initial value or a repeat count

In a complex constant as the real or imaginary part

In a LEVEL statement as the storage level

In a C\$ directive as a primary in an expression, or as a parameter value

A symbolic constant cannot appear in a FORMAT statement.

SAVE STATEMENT

The SAVE statement (figure 2-22 and example in figure 2-23) is used to retain the definition status of entities after the execution of a RETURN or END statement in a subprogram. A SAVE statement in a main program is optional and has no effect. A SAVE statement does not have the effect of retaining the definition status of an entity after the execution of a RETURN or END statement in the main program unit of an overlay.

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

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

Figure 2-22. SAVE Statement

```

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

```

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

Figure 2-23. Example of SAVE Statement

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

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

Entities specified by SAVE statements do not become undefined.

Entities in blank common do not become undefined.

Entities that have been initially defined (and not redefined) do not become undefined.

Entities in a named common block that appears in the subprogram and in the referencing program unit do not become undefined.

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

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

TYPE STATEMENTS

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

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

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

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

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

Each of the explicit type statements is discussed below. See figure 2-24 for examples of all but CHARACTER type statements.

```

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

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

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

COMPLEX CPVAR
COMPLEX RES(5,5)

BOOLEAN ALABEL,QMASK
BOOLEAN LABEL(14)

LOGICAL SWITCH,TEST

```

Figure 2-24. Examples of Explicit Typing

INTEGER STATEMENT

The **INTEGER** statement (figure 2-25) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type integer.

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

Figure 2-25. **INTEGER** Statement

REAL STATEMENT

The **REAL** statement (figure 2-26) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type real.

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

Figure 2-26. **REAL** Statement

DOUBLE PRECISION STATEMENT

The **DOUBLE PRECISION** statement (figure 2-27) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type double precision.

COMPLEX STATEMENT

The **COMPLEX** statement (figure 2-28) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type complex.

DOUBLE PRECISION name[,name]...

name Is explicitly typed as a double precision. Each name is one of the forms:

var

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

var Is a variable, function name, or function entry.

array Is an array name.

d Specifies the bounds of a dimension.

Figure 2-27. **DOUBLE PRECISION** Statement

COMPLEX name[,name]...

name Is explicitly typed as a complex. Each name is one of the forms:

var

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

var Is a variable, function name, or function entry.

array Is an array name.

d Specifies the bounds of a dimension.

Figure 2-28. **COMPLEX** Statement

LOGICAL STATEMENT

The **LOGICAL** statement (figure 2-29) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type logical.

LOGICAL name[,name]...

name Is explicitly typed as logical. Each name is one of the forms:

var

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

var Is a variable, function name, or function entry.

array Is an array name.

d Specifies the bounds of a dimension.

Figure 2-29. **LOGICAL** Statement

BOOLEAN STATEMENT

NOTE

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

The BOOLEAN statement (figure 2-30) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type Boolean.

BOOLEAN name[,name]. . .

name Is explicitly typed as Boolean. Each name is one of the forms:

var

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

var Is a variable, function name, or function array.

array Is an array name.

d Specifies the bounds of a dimension.

Figure 2-30. BOOLEAN Statement

CHARACTER STATEMENT

The CHARACTER statement (figure 2-31 and examples in figure 2-32) is used to define a variable, array, symbolic constant, function name, or dummy procedure name as type character.

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

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

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

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

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

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

name Is explicitly typed as character. Each name is one of the forms:

var [*len]

array [(d[,d]. . .)] [*len]

len Specifies the length and can be: an unsigned nonzero integer constant; an integer constant expression, enclosed in parentheses, with a positive value; or an asterisk enclosed in parentheses.

var Is a variable, function name, or function entry.

array Is an array name.

d Specifies the bounds of a dimension.

Figure 2-31. CHARACTER Statement

Example 1:

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

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

Example 2:

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

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

Figure 2-32 Examples of CHARACTER Statement

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

```
CHARACTER *(*) FUNCTION ALPHA( )  
and  
CHARACTER *3 FUNCTION ALPHA(X,Y)
```

are valid forms of the specification.

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

If a symbolic constant of type character has the length (*) specified, the constant has the length of its corresponding constant expression in a

PARAMETER statement. If the length specification is a symbolic constant, it must be enclosed in parentheses. For example,

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

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

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

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

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

Character substrings are described in section 1.

Expressions are formed from a combination of operators, operands, and parentheses.

Assignment statements are executable statements that use expressions to define or redefine the values of variables.

This section discusses the types of expressions and the types of assignment statements.

EXPRESSIONS

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

The types of expressions are: arithmetic, Boolean, character, logical, and relational. (The relational expressions are not fully independent and are used as parts of logical expressions.) Each of these types is discussed below.

ARITHMETIC EXPRESSION

An arithmetic expression (figure 3-1) is a sequence of unsigned constants, symbolic constants, variables, array elements, and function references separated by operators and parentheses. For example,

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

is a valid arithmetic expression.

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

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

arithexp	
arithexp	Is an arithmetic expression in one of the forms:
	term
	+ term
	- term
	arithexp + term
	arithexp - term
	+ boolprim
	- boolprim
	arithexp + boolprim
	arithexp - boolprim
	boolprim + term
	boolprim - term
	boolprim + boolprim
	boolprim - boolprim
term	
term	Is an arithmetic term in one of the forms:
	fact
	term * fact
	term / fact
	term * boolprim
	term / boolprim
	boolprim * fact
	boolprim / fact
	boolprim * boolprim
	boolprim / boolprim
fact	
fact	Is an arithmetic factor in one of the forms:
	prim
	prim ** fact
	boolprim ** fact
	prim ** boolprim
	boolprim ** boolprim
boolprim	
boolprim	Is a Boolean primary, as described for Boolean expressions.
prim	
prim	Is an arithmetic primary. An arithmetic primary can be an arithmetic expression enclosed in parentheses, or any of the following:
	Unsigned arithmetic constant
	Arithmetic symbolic constant
	Arithmetic variable
	Arithmetic array element reference
	Arithmetic function reference

Figure 3-1. Arithmetic Expression

TABLE 3-1. ARITHMETIC OPERATORS

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

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

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

For example, in the expression

`-A**2`

the exponentiating operator (`**`) has precedence over the negation operator (`-`). The operands of the exponentiation operator are combined to form an expression used as the operand of the negation operator. The expression is the same as the expression `-(A**2)`.

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

`2**3**2` is interpreted as `2**(3**2)`

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

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

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

An arithmetic constant expression contains only arithmetic constants, symbolic names of arithmetic

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

An integer constant expression is an arithmetic constant expression or a Boolean constant expression in which each constant or symbolic name of a constant is of type integer or Boolean. If a Boolean constant expression `e` appears, the value used is `INT(e)`. Note that variable, array element, and function references are not allowed. See figure 3-2 for integer constant expression examples.

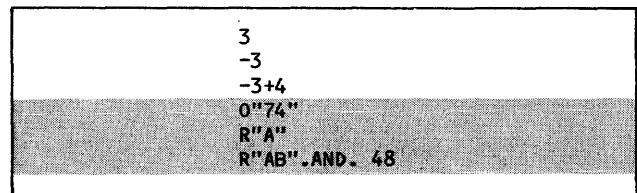


Figure 3-2. Examples of Integer Constant Expressions

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

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

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

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

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

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

TABLE 3-3. RESULTING DATA TYPE FOR X1+X2, X1-X2, X1*X2 or X1/X2 OPERATIONS (Contd)

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

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

Also, a Boolean operand in an exponentiation operation is converted to integer. For the + - * and / operations, if two operands are of different type and one type is Boolean, the result has the type of the other operand. If both operands are of type Boolean, the result has type integer. The result of the operator + or the operator - operating on a single Boolean operand is of type integer. A Boolean operand is converted to the type of the result, and the operation is performed on the converted operand.

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

Infinite and Indefinite Values

Expressions that generate illegal values (such as infinite and indefinite values) or that use operands containing such values can lead to execution errors. The point at which the error occurs depends on the computer model being used. For more information, refer to the definitions of infinite and indefinite values in the glossary.

BOOLEAN EXPRESSION

A Boolean expression (figure 3-3) is formed with logical operators and Boolean or arithmetic operands. Evaluation of a Boolean expression produces a result of type Boolean.

Boolean quantities are combined from left to right when a Boolean expression contains two or more .AND. operators, two or more .OR. operators, or two or more .EQV., .NEQV., or .XOR. operators.

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

A Boolean operator determines each bit value of the result it yields independently of other bits of the result. Each bit value is determined from the corresponding bit values of the operands. At each bit position, the bit in the result is determined as shown in table 3-4.

boolexp	
boolexp	Is a Boolean expression in one of the forms: booldis boolexp .EQV. booldis boolexp .EQV. arithexp arithexp .EQV. booldis arithexp .EQV. arithexp boolexp .NEQV. booldis boolexp .NEQV. arithexp arithexp .NEQV. booldis arithexp .NEQV. arithexp boolexp .XOR. booldis boolexp .XOR. arithexp arithexp .XOR. booldis arithexp .XOR. arithexp
booldis	Is a Boolean disjunct in one of the forms: boolterm booldis .OR. boolterm booldis .OR. arithexp arithexp .OR. boolterm arithexp .OR. arithexp
boolterm	Is a Boolean term in one of the forms: boolfact boolterm .AND. boolfact boolterm .AND. arithexp arithexp .AND. boolfact arithexp .AND. arithexp
boolfact	Is a Boolean factor in one of the forms: boolprim .NOT. boolprim .NOT. arithexp
arithexp	Is an arithmetic expression.
boolprim	Is a Boolean primary. A Boolean primary can be a Boolean expression enclosed in parentheses, or any of the following: Unsigned Boolean constant Boolean symbolic constant Boolean variable Boolean array element reference Boolean function reference

Figure 3-3. Boolean Expression

TABLE 3-4. RESULT OF LOGICAL OPERATORS IN BOOLEAN EXPRESSIONS

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

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

CHARACTER EXPRESSION

A character expression (figure 3-4) is used to express a character string. Evaluation of a character expression produces a result of type character. The simplest form of a character expression is a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, or character function reference. More complicated character expressions can be formed by using one or more character operands together with character operators and parentheses. The character operator is shown in table 3-5.

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

Figure 3-4. Character Expression

TABLE 3-5. CHARACTER OPERATOR

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

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

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

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

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

is the same as the interpretation of the character expression

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

The value of the preceding expression is the same as that of the constant 'ABCDEF'.

Note that parentheses have no effect on the value of a character expression. Thus, the expression

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

has the same value as the preceding expressions.

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

LOGICAL EXPRESSION

A logical expression (figure 3-5) is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

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

logexp	
logexp	Is a logical expression in one of the forms: logdis logexp .EQV. logdis logexp .NEQV. logdis logexp .XOR. logdis
logdis	Is a logical disjunction in either form: logterm logdis .OR. logterm
logterm	Is a logical term in either form: logfact logterm .AND. logfact
logfact	Is a logical factor in either form: logprim .NOT. logprim
logprim	Is a logical primary. A logical primary can be a logical expression enclosed in parentheses, a relational expression, or any of the following: Logical constant Logical symbolic constant Logical variable Logical array element reference Logical function reference

Figure 3-5. Logical Expression

The logical operators are shown in table 3-6.

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

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

For example, in the expression

A .OR. B .AND. C

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

A .OR. (B .AND. C)

Logical quantities are combined from left to right when a logical expression contains two or more .AND. operators, two or more .OR. operators, or two or more .EQV., .NEQV., or .XOR. operators.

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

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

RELATIONAL EXPRESSION

A relational expression (figure 3-6) can appear only within logical expressions. Evaluation of a relational expression produces a logical result with a true or false value.

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

Comparisons of variables or character strings are performed left-to-right.

The relational operators are shown in table 3-8.

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

TABLE 3-6. LOGICAL OPERATORS

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

TABLE 3-7. RESULT OF LOGICAL OPERATORS

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

relexp	
relexp	Is a relational expression used as a primary in a logical expression. A relational expression is in one of the forms: arithexp rop arithexp arithexp rop boolprim boolprim rop arithexp boolprim rop boolprim charexp rop charexp
rop	Is one of the relational operators: .LT. .LE. .EQ. .NE. .GT. .GE.
arithexp	Is an arithmetic expression.
boolprim	Is a Boolean primary, as described for Boolean expressions.
charexp	Is a character expression.

Figure 3-6. Relational Expression

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

$$X1 \text{ relop } X2$$

is the value of the expression

$$((X1) - (X2)) \text{ relop } 0$$

where 0 (zero) is of the same type as the expression.

Note that the comparison of a double precision value and a complex value is not permitted.

A character relational expression has the logical value true only if the values of the operands satisfy the relation specified by the operator. The character expression X1 is considered to be less than X2 if the value of X1 precedes the value of X2 in the collating sequence; X1 is greater than X2 if the value of X1 follows the value of X2 in the collating sequence. Note that the collating sequence in use determines the result of the comparison. The default collating sequence is ASCII6. The COLLATE compiler directive also affects collation.

TABLE 3-8. RELATIONAL OPERATORS

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

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

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

GENERAL RULES FOR EXPRESSIONS

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

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

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

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

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

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

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

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

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

Any arithmetic operation whose result is not mathematically defined is prohibited: for example, neither dividing by zero nor raising a zero-valued primary to a zero-valued or negative-valued power is allowed.

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

$A(I) = F(I)$

$Y = G(X) + X$

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

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

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

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y.

If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the expression causes Z to become undefined if L defines its argument.

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

$Y = F(G(X))$

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

Any expression contained in parentheses is always treated as an entity. For example, in evaluating the expression

$A*(B*C)$

the product of B and C is evaluated and then multiplied by A; the mathematically equivalent expression $(A*B)*C$ is not used.

ASSIGNMENT STATEMENTS

There are five types of assignment statements:

Arithmetic

Boolean

Character

Logical

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

Each type of assignment statement is discussed below.

ARITHMETIC ASSIGNMENT

The arithmetic assignment statement is shown in figure 3-7.

```
v = e

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

e  Is an arithmetic or Boolean expression.
```

Figure 3-7. Arithmetic Assignment Statement

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

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

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

BOOLEAN ASSIGNMENT

The Boolean assignment statement is shown in figure 3-8.

```
v = e

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

e  Is a Boolean or arithmetic expression.
```

Figure 3-8. Boolean Assignment Statement

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

CHARACTER ASSIGNMENT

The character assignment statement is shown in figure 3-9.

```
v = e

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

e  Is a character expression.
```

Figure 3-9. Character Assignment Statement

The character expression e is evaluated, and the result is then assigned to v.

None of the character positions being defined in v can be referenced in e.

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

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

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

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

LOGICAL ASSIGNMENT

The logical assignment statement is shown in figure 3-10.

The logical expression is evaluated and the result is then assigned to v. Note that e must have a value of either `.TRUE.` or `.FALSE.`

```
v = e

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

e  Is a logical expression.
```

Figure 3-10. Logical Assignment Statement

MULTIPLE ASSIGNMENT

The multiple assignment statement is shown in figure 3-11.

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

$v = [v=] \dots e$

- v* Is the name of a variable, array element, or character substring.
- e* Is an expression. The types of the elements *v* and the expression *e* must ensure that $v = e$ is a valid assignment for each *v* specified.

Figure 3-11. Multiple Assignment Statement

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

```

GO TO
IF
DO
CONTINUE
PAUSE
STOP
END
CALL
RETURN
    
```

This section discusses these flow control statements, except for CALL and RETURN. CALL is used to transfer control to a subroutine, and RETURN is used to return control from a subprogram; they are discussed in section 6.

Control can be transferred only to an executable statement.

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

This section discusses the flow control statements.

GO TO STATEMENT

The three types of GO TO statements are the unconditional GO TO, the computed GO TO, and the assigned GO TO. As the ASSIGN statement is used in conjunction with the assigned GO TO, it is also described below.

UNCONDITIONAL GO TO STATEMENT

The unconditional GO TO statement (figure 4-1 and example in figure 4-2) transfers control to the statement identified by the specified label. The labeled statement must appear in the same program unit as the GO TO statement.

```

GO TO slab

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

Figure 4-1. Unconditional GO TO Statement

```

10 A=B+Z
100 B=X+Y
      IF(A-B)20,20,30
20 Z=A
      GO TO 10
30 Z=B
      STOP
      END

Control transfers to statement 10 when the
GO TO statement executes.
    
```

Figure 4-2. Example of Unconditional GO TO Statement

COMPUTED GO TO STATEMENT

The computed GO TO statement (figure 4-3 and example in figure 4-4) transfers control to the statement identified by one of the specified labels.

```

GO TO (slab[,slab] . . . )[,lexp

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

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

Figure 4-3. Computed GO TO Statement

The label selected is determined by the value of the expression. If exp has a value of 1, control transfers to the statement identified by the first label in the list; if exp has a value of 4, control transfers to the statement identified by the fourth label in the list, and so forth. The value of exp is truncated and converted to integer, if necessary.

If the value of exp is less than 1 or greater than the number of labels in the list, execution continues with the statement following the computed GO TO.

ASSIGN STATEMENT

The ASSIGN statement (figure 4-5 and example in figure 4-6) assigns a statement label to an integer variable. The value assigned represents the label of an executable statement or a FORMAT statement. The labeled statement must appear in the same program unit as the ASSIGN statement. Once the variable is used in an ASSIGN statement, it cannot be used in any statement other than an assigned GO TO statement, an ASSIGN statement, or an input/output statement, until it has been redefined.

Example 1:

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

The next statement executed is:

```
10 if L = 1
```

```
20 if L = 2
```

```
30 if L = 3
```

```
20 if L = 4
```

Example 2:

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

Statement 150 is executed next.

Example 3:

```
K=2  
X=4.6
```

```
.  
.  
.
```

```
GO TO(10,110,11,12,13),X/K
```

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

Example 4:

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

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

Figure 4-4. Examples of Computed GO TO Statements

```
ASSIGN slab TO iv
```

slab Is the label of an executable or FORMAT statement.

iv Is an integer variable.

Figure 4-5. ASSIGN Statement

Example 1:

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

Control transfers to the statement labeled 10.

Example 2:

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

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

Figure 4-6. Examples of ASSIGN Statement

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

ASSIGNED GO TO STATEMENT

The assigned GO TO statement (figure 4-7 and example in figure 4-8) transfers control to the executable statement last assigned to integer variable iv by the execution of a prior ASSIGN statement.

```
GO TO iv [[,] (slab[,slab]. . . )]
```

iv Is an integer variable.

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

Figure 4-7. Assigned GO TO Statement

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

Statement 50 is executed immediately after statement 10.

Figure 4-8. Example of Assigned GO TO Statement

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

IF STATEMENT

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

Arithmetic `IF`

Logical `IF`

Block `IF`

The `ELSE`, `ELSE IF`, and `END IF` statements are also discussed as they are used in conjunction with a block `IF` statement.

ARITHMETIC IF STATEMENT

The arithmetic `IF` statement (figure 4-9 and example in figure 4-10) transfers control to one of three labeled statements, depending on the value of the expression. If the value of `exp` is negative, control transfers to the first statement label; if `exp` is 0, control transfers to the second statement label; if `exp` is greater than 0, control transfers to the third statement label. If `exp` is type `Boolean`, `INT(exp)` is used.

```
IF (exp) slab1,slab2,slab3

exp          Is an integer, real, double
              precision, or Boolean expression.

slab1,slab2, Are statement labels of execut-
slab3        able statements that appear in
              the same program unit as the
              arithmetic IF statement.
```

Figure 4-9. Arithmetic `IF` Statement

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

Figure 4-10. Example of Arithmetic `IF` Statement

LOGICAL IF STATEMENT

The logical `IF` statement (figure 4-11 and example in figure 4-12) allows for conditional execution of a statement. If the value of `exp` is true, statement `stat` is executed. If the value of `exp` is false, `stat` is not executed; execution continues with the next statement.

```
IF (exp) stat

exp          Is a logical expression.

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

Figure 4-11. Logical `IF` Statement

```
Example 1:
  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.

```
Example 2:
  IF (A.LT.B) CALL SUB1
  20 ZETA=TEMP+RES4
```

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

Figure 4-12. Examples of Logical `IF` Statements

BLOCK IF STATEMENT

The block `IF` statement (figure 4-13) allows conditional execution of a block of executable statements. The block `IF` statement is used with the `END IF` and, optionally, the `ELSE` and `ELSE IF` statements to form block `IF` structures.

```
IF (exp) THEN

exp          Is a logical expression.
```

Figure 4-13. Block `IF` Statement

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

ELSE Statement

The ELSE statement (figure 4-14) provides an alternate path of execution for a block IF statement or an ELSE IF statement.

```
ELSE
```

Figure 4-14. ELSE Statement

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

ELSE IF Statement

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

```
ELSE IF (exp) THEN  
    exp    Is a logical expression.
```

Figure 4-15. ELSE IF Statement

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

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

END IF Statement

The END IF statement (figure 4-16) terminates a block IF structure. For each block IF statement there must be a corresponding END IF statement.

```
END IF
```

Figure 4-16. END IF Statement

BLOCK IF Structures

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

The simplest form of a block IF structure is shown in figure 4-17, with an example in figure 4-18.

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

Figure 4-17. Simple Block IF Structure

```
IF (I.EQ.0) THEN  
    X=X+DX  
    Y=Y+DY  
END IF  
  
If I is zero, the subsequent statements are executed.  
If not, control transfers to the statement following  
END IF.
```

Figure 4-18. Example of Block IF Statement

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

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

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

A block IF structure can contain one ELSE statement to provide an alternative path of execution within the structure (figure 4-19 and example in figure 4-20).

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

Figure 4-19. Block IF Structure With ELSE Statement

```

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

```

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

In the structure with an ELSE statement, execution continues with the first statement in if-block-1 if exp is true. If the last statement of if-block-1 does not transfer control, control transfers to the statement following END IF. However, if exp is false, control transfers to the first statement in if-block-2. If the last statement in if-block-2 does not transfer control, execution continues with the statement following END IF.

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

An IF structure can contain one or more ELSE IF statements to provide for alternative execution of additional block IF statements (figure 4-21 and example in figure 4-22). This capability allows formation of IF structures containing a number of possible execution paths depending on the outcome of the associated IF tests.

```

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

```

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

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

```

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

```

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

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

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

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

NESTED BLOCK IF STRUCTURES

Block IF structures can be nested: that is, any if-block within a structure can itself contain block IF structures (figure 4-23 and example in figure 4-24). Within a nesting hierarchy, control can transfer from a lower level structure into a higher level structure; however, control cannot transfer from a higher level structure into a lower level structure.

```

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

```

Figure 4-23. Nested Block IF Structure

```

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

```

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

Figure 4-24. Example of Nested Block IF Structure

DO STATEMENT

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

```
DO slab [,] v=e1,e2[,e3]
```

slab	Is the label of an executable statement called the terminal statement of the DO loop.
v	Is an integer, real, or double precision control variable.
e1	Is an initial parameter.
e2	Is a terminal parameter.
e3	Is an optional increment parameter; default is 1.

e1, e2, and e3 are called indexing parameters; they can be integer, real, double precision, or Boolean constants, symbolic constants, variables, or expressions.

Figure 4-25. DO Statement

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

The range of a DO loop consists of all the executable statements following the DO statement up to and including the terminal statement. Execution of a DO statement is affected by the DO option of the compiler call. DO execution is as follows:

1. The expressions e1, e2, and e3 are evaluated and, if necessary, converted to the type of the control variable v.
2. Control variable v is assigned the value of e1.
3. The iteration count is established; this value is determined by the following expression:

$$\text{MAX}(\text{INT}((m2-m1+m3)/m3), \text{mtc})$$

m1, m2, m3

are the values of the expressions e1, e2, and e3, respectively, after conversion to the type of v.

mtc

is the minimum trip count; mtc has a value of 0 or 1, and is established by the DO control statement parameter or DO loop control directive. A zero trip count is prohibited in DO=OJ mode, and the results of such a loop are undefined.

4. If the iteration count is not 0, the range of the DO loop is executed. If the iteration count is 0, execution continues with the statement following the terminal statement of the DO loop; the control variable retains its most recent value.
5. Control variable v is incremented by the value of e3.
6. The iteration count is decremented by one.

Steps 4 through 6 are repeated until the iteration count has a value of 0.

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

$$| m1+m3 | < 2^{17}-1$$

$$| m2+m3 | < 2^{17}-1$$

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

A DO loop can be active, inactive, and nested. Each is discussed below.

ACTIVE AND INACTIVE DO LOOPS

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

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

The iteration count is determined to be 0.

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

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

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

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

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

If a DO loop executes zero times, the control variable value equals ml. Otherwise, if the index variable iterates through the terminal parameter value, the control value is the most recent value of the control variable plus the increment parameter value.

If a DO loop becomes inactive but has not executed to completion (iteration count does not equal 0), its control variable retains its most recent value unless it has become undefined.

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

Figure 4-26 shows examples of DO loops.

NESTED DO LOOPS

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

The last statement of an inner DO loop must be either the same as the last statement of the outer DO loop or must occur before it. A terminal statement that is shared by more than one DO loop can be referenced in a GO TO or IF statement from within the range of the innermost loop only (figure 4-27). If the terminal statement is referenced from any loop other than the innermost loop, results are undefined.

Example 1:

```
DO 20 I=1,11,3
  IF(ALIST(I)-ALIST(I+1))15,10,10
15  ITEMP=ALIST(I)
10  ALIST(I)=ALIST(I+1)
20  CONTINUE
300 WRITE(6,200)ALIST
```

The statements following DO up to and including statement 20 are executed four times. The DO loop is executed with I equal to 1, 4, 7, and 10. Statement 300 is then executed. After completion of the loop, I has a value of 13.

Example 2:

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

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

Example 3:

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

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

Example 4:

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

If DO=0T is specified on the FTNS control statement, the DO loop is executed once (with I=5) because J is larger than K. If DO=0T is not specified, the loop is not executed.

Figure 4-26. DO Loop Examples

See figure 4-28 for examples of possible DO loop nests. Note that loops can be completely nested or can share a terminal statement.

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

```

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

```

Figure 4-27. Branch to Shared Terminal Statement

Figure 4-29 illustrates legal and illegal transfers with nested loops. A transfer from the range of an outer DO into the range of an inner DO loop is not allowed (transfer 1 in example); 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 (transfer 2). A transfer back into the range of an innermost DO loop is allowed if a transfer has been made from the same DO loop and is still active (transfer 3). A transfer out of the range of an inner DO and a transfer back into the range of an innermost DO is not allowed (transfer 4).

See figure 4-30 for nested DO loop examples.

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

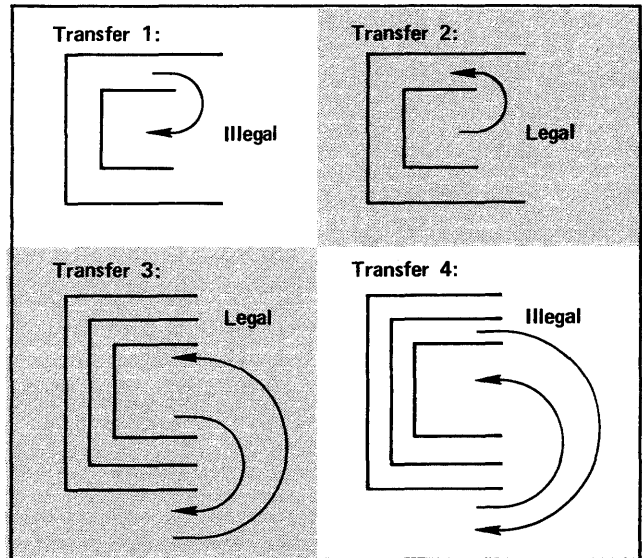


Figure 4-29. Nested DO Loop Transfers

CONTINUE STATEMENT

The CONTINUE statement (figure 4-32 and example in figure 4-33) performs no operation. It is an executable statement that can be placed anywhere in the executable statement portion of a source program without affecting the sequence of execution.

The CONTINUE statement is most frequently used as the last statement of a DO loop. It can provide loop termination when a GO TO or IF would normally be the last statement of the loop. If the CONTINUE statement does not have a label, an informative diagnostic is issued.

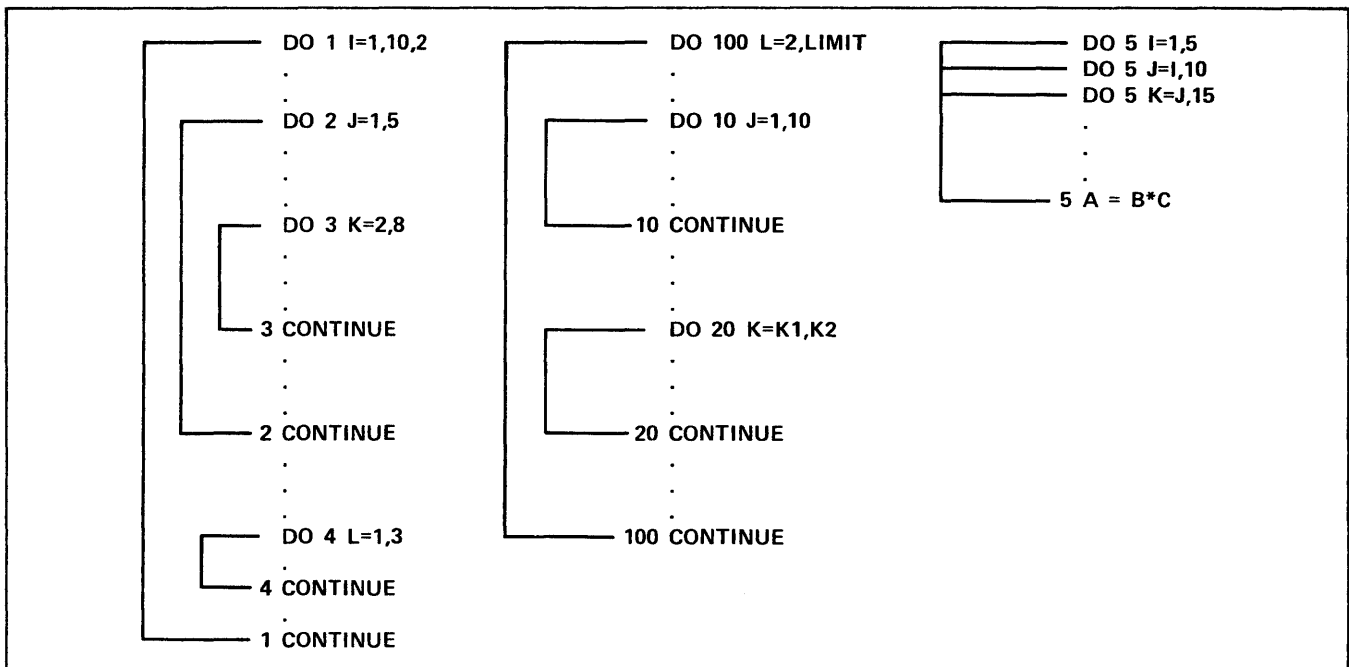


Figure 4-28. Nested DO Loops

Example 1:

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

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

Example 2:

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

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

Example 3:

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

Arrays A and B are set to zero.

Figure 4-30. Nested DO Loop Examples

PAUSE STATEMENT

The PAUSE statement (figure 4-34) causes the program to temporarily suspend execution. At the same time, PAUSE n appears as a dayfile message on the operator console. If the job is executing interactively, PAUSE n appears as a dayfile message at the user terminal (does not apply to SCOPE 2). For batch originated programs, the console operator can continue or terminate the program with an entry from the console.

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

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

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

†Transfer bypasses inner loops.

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

CONTINUE

Figure 4-32. CONTINUE Statement

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

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

Figure 4-33. CONTINUE Statement Examples

PAUSE[n]

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

Figure 4-34. PAUSE Statement

For programs executing interactively through the NOS time-sharing system, the user types STOP to terminate execution. Any other type-in causes execution to continue.

Examples:

```
PAUSE 45321
```

```
PAUSE 'EXAMPLE TWO'
```

STOP STATEMENT

The STOP statement (figure 4-35) terminates program execution.

<p>STOP[n]</p> <p>n is a string of 1 through 5 decimal digits, or a character constant of at most 70 characters.</p>

Figure 4-35. STOP Statement

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

Example:

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

END STATEMENT

The END statement (figure 4-36) indicates the end of the program unit to the compiler.

<p>END</p>

Figure 4-36. END Statement

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

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

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

All FORTRAN files are closed during END processing. The CYBER Record Manager's error file (zzzzzEG) is also checked. If open, its buffers are flushed.

This section discusses input/output files, input/output lists, implied DO input/output lists, format specification, and input/output statements.

This section does not discuss the following functions and subprograms used in connection with input/output: the EOF function, the IOCHEC function, the UNIT function, the LENGTH function, the LENGTHX subroutine, and the mass storage subroutines (READMS, WRITMS, OPENMS, CLOSMS, and STINDX). These functions and subprograms are discussed in section 7. Also, the CYBER Record Manager routines are not discussed in this section; they are discussed in section 8.

INPUT/OUTPUT FILES

Input and output involve reading records from files and writing records to files.

Every file must have a logical file name of 1 through 7 letters and digits beginning with a letter. The logical file name, which is defined only for the current job, is the name by which the file is referenced in control statements.

Input and output operations with sequential files, direct access files, and batch job files are discussed separately below.

SEQUENTIAL FILES

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

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

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

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

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

The error condition can be checked by use of the ERR= or IOSTAT= parameter in the READ or WRITE statement, for every formatted, list directed, namelist, or unformatted READ or WRITE. If an error is encountered and a test is not included, the program terminates with a fatal error.

If the end-of-file status or an error condition is encountered by a READ statement, all items in the input list, including the implied DO variables, become undefined.

Record length on card files should not exceed 80 characters. Record length on print files should not exceed 137 characters; the first character is always used as carriage control and is not printed. The second character appears in the first print position.

DIRECT ACCESS FILES

Direct access file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written, and they normally can be read back only in the same order. This can be slow and inconvenient in applications where the order of writing is not the same as the retrieval order. In addition, such processing requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, a direct access file capability is provided by the FORTRAN input/output statements.

In a direct access file, any record can be read, written, or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random access mass storage device that can be positioned to any portion of a file. Thus, the entire concept of file position does not apply to a direct access file. The notion of rewinding a direct access file is, for instance, without meaning.

To create a direct access file the user must specify an OPEN statement with ACCESS='DIRECT' and include the RECL (record length) parameter. For example,

```
OPEN(2,FILE='DAFL',ACCESS='DIRECT',RECL=120)
```

opens an unformatted file DAFL for direct access. The file is associated with unit 2 and has a record length of 120 words.

All records in a direct access file must have the same length.

The record length for a formatted direct access file is specified in characters. The record length for an unformatted direct access file is specified in words. If the iolist for an unformatted WRITE contains character data, the record length to be written must still be specified in words: it can be determined by the following rules:

1. Each noncharacter item counts as one word except for double precision and complex items, which count as two words.
2. The length in words of each contiguous group of character items is determined by adding 9 to the combined length of the items in characters and dividing this result by 10, discarding the fractional part.
3. The lengths calculated in steps 1 and 2 are added to determine the record length in words.

See figure 5-1 for an example of record length calculation.

```
CHARACTER A*7,B*9,C*10,D*20,E*15,F*12
INTEGER IA,IB,IC,ID(5)
OPEN (5,ACCESS='DIRECT',
+FORM='UNFORMATTED',RECL=16)
WRITE (5,REC=1) A,B,IA,C,IB,E,D,ID,F
```

The length of the output record is determined by the following calculation:

(length of A + length of B + 9)/10	= 2 words
IA	= 1 word
(length of C + 9)/10	= 1 word
IB	= 1 word
(length of E + length of D + 9)/10	= 4 words
ID	= 5 words
(length of F + 9)/10	= 2 words

Record length = 2+1+1+1+4+5+2=16 words

Figure 5-1. Direct Access File Record Length Calculation Example

A record number identifies a record in a direct access file. The record number is a positive decimal integer that is assigned when the record is written. Once a record number is assigned to a record, the record can always be accessed by that record number. The order of record numbers is the order of records on a direct access file.

Records can be written, rewritten, or read by specifying the record number in a READ or WRITE statement. Records can be read or written in any order; they need not be referenced in the order of their record numbers. The number of the record to be read or written is specified in a READ or WRITE statement with the REC= parameter.

The REC= parameter, on a direct access READ statement, must not be set to a record number greater than the highest record number written in the file. An attempt to read record numbers greater than the highest in the file can return unpredictable data without any error being reported.

If the length of the iolist in a direct access formatted WRITE statement is less than the record length of the direct access file, the unused portion of the record is blank filled. A direct access WRITE statement must not write a record longer than the record length.

A direct access file can be opened for formatted or unformatted input/output. However, neither list directed nor namelist input/output can be used with direct access files.

An internal file cannot be opened for direct access. A discussion of internal files follows in this section.

See figure 5-2 for examples of direct access file access.

Example 1:

```
WRITE(2,'(3E10.4)',REC=6)A,B,C
WRITE(2,'(2I4,G20.10)',REC=1)I,J,X
```

Variables A, B, and C are written to record number 6, and variables I, J, and X are written to record number 1 of the direct access file associated with unit 2.

Example 2:

```
OPEN(2,FILE='DARG',ACCESS='DIRECT',
+FORM='FORMATTED',RECL=72)
DO 14 I=10,2,-2
READ(2,99,REC=I,ERR=20) (A(J),J=1,6)
99 FORMAT (6E12.6)
.
.
.
14 CONTINUE
```

Records 10, 8, 6, 4, and 2 are read from the direct access file DARG.

Figure 5-2. Direct Access File Processing Examples

BATCH JOB FILES

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

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

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

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

INPUT/OUTPUT LISTS

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

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

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

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

Subscripts in an input/output list can be written as any valid subscript form described in section 1.

See figure 5-3 for input/output list examples.

Input record contains data:

100223456712

Example 1:

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

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

Example 2:

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

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.

Figure 5-3. Input/Output List Examples

IMPLIED DO IN INPUT/OUTPUT LIST

An implied DO specification has the following form:

```
(dlist,i=e1,e2 [,e3])
```

The elements *i*, *e1*, *e2*, and *e3* have the same meaning as in the DO statement, and *dlist* is an input/output list. The range of an implied DO specification is that of *dlist*. The value of *i* must not be changed within the range of the implied DO list by a READ statement. Changes to the values of *e1*, *e2*, and *e3* have no effect upon the execution of the implied DO. However, their values can be changed in a READ statement if they are outside the range of the implied DO, and the change does have effect. For example,

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

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

The statements:

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

read two records, each containing a value for A.

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

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

Also,

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

causes CAT, DOG, and RAT to be written 10 times each.

A variable cannot be used as a control variable more than once in the same implied DO nest, but iolist items can appear more than once. The value of a control variable within an implied DO specification is defined within that specification. On exit from the implied DO specification the control variable retains the first value to exceed the upper limit (*e2*).

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

```
((list,v1=e1,e2,e3),v2=eel,ee2,ee3)
```

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

The nested form can be used to read into and write from arrays. See figure 5-4 for examples.

Each execution of an input or output statement transmits at least one record regardless of the FORMAT statement. Once a READ is initiated, the FORMAT statement determines when a new record will be transmitted. See figure 5-5 for examples.

FORMAT SPECIFICATION

Format specifications are used in conjunction with formatted input/output statements to produce output or read input that consists of strings of display code characters. On input, data is converted from a specified format to its internal binary representation. On output, data is converted from its internal binary representation to the specified format before it is transmitted. Formats can be specified by:

The statement label of a FORMAT statement.

An integer variable which has been assigned the statement label of a FORMAT statement (see ASSIGN statement).

A character array name or any character expression, except one involving assumed-length character entities.

A noncharacter array name.

The following paragraphs discuss FORMAT statements, character format specifications, noncharacter format specifications, edit descriptors, and execution time format specification

Example 1:

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

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

Example 2:

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

Same effect as example 1.

Example 3:

```
READ (5,100) (VECTOR (I),I=1,10)
100 FORMAT (4F7.2)
```

Reads only 3 records, with 10 values read.

Example 4:

```
DO 40 I = 1,10
40 READ (5,100) VECTOR (I)
100 FORMAT (4F7.2)
```

Reads 10 records and 10 values, as in example 2.

Figure 5-5. Format Terminating Data Read Examples

Example 1:

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

Example 2:

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

is equivalent to

```
DIMENSION VECTOR(3,4,7)
READ (3,100) (((VECTOR(I,J,K),I=1,3),J=1,4),K=1,7)
100 FORMAT (I6)
```

Example 3:

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

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)
```

Figure 5-4. Implied DO in Input/Output List Examples

FORMAT STATEMENT

The **FORMAT** statement (figure 5-6) is a nonexecutable statement which specifies the formatting of data to be read or written with formatted I/O.

sl	FORMAT (flist)
sl	Is a statement label.
flist	Is a list of items, separated by commas, having the following forms: [r]ed ned [r](flist)
ed	Is a repeatable edit descriptor.
ned	Is a nonrepeatable edit descriptor.
r	Is a nonzero unsigned integer constant repeat specification.

Figure 5-6. **FORMAT** Statement

The **FORMAT** statement is used in conjunction with formatted input and output statements. It can appear anywhere in the program after the **PROGRAM**, **FUNCTION** or **SUBROUTINE** statement. An example of a **FORMAT** statement and its associated **READ** statement is as follows:

```
READ (5,100) INK,NAME,AREA
100 FORMAT (10X,I4,I2,F7.2)
```

The format specification consists of edit descriptors in parentheses. Blanks are not significant except in H, quote, and apostrophe descriptors.

Generally, each item in an input/output list is associated with a corresponding edit descriptor in a **FORMAT** statement. The **FORMAT** statement specifies the external format of the data and the type of conversion to be used. Complex variables always correspond to two edit descriptors. Double precision variables correspond to one edit descriptor when using D, E, F, or G; otherwise, they correspond to two edit descriptors. The D edit descriptor corresponds to exactly one list item. Complex editing requires two (D, E, F, G) descriptors; the two descriptors can be different.

The type of conversion should correspond to the type of the variable in the input/output list. The **FORMAT** statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete. For example, the statements

```
INTEGER N
READ (5,100) N
100 FORMAT (F10.2)
```

will assign a floating point number to the variable N which could cause unpredictable results if N is referenced later as an integer.

CHARACTER FORMAT SPECIFICATIONS

A format specification can also be specified as a character expression or as the name of a character variable or array containing a format specification. The form of these format specifications is the same as for **FORMAT** statements without the keyword **FORMAT**. Any character information beyond the terminating parenthesis is ignored. The initial left parenthesis can be preceded by blanks. See figure 5-7 example 1.

Example 1:
<pre>CHARACTER FORM*11 DATA FORM/'(I3,2E14.4)'/ READ (2,FMT=FORM,END=50) N,A,B</pre>
is equivalent to
<pre>READ (2,FMT=100,END=50) N,A,B 100 FORMAT (I3,2E14.4)</pre>
The examples above can also be expressed as:
<pre>READ (2,FMT='(I3, 2E14.4)',END=50) N,A,B or CHARACTER FORM*(*) PARAMETER (FORM='(I3,2E14.4)') READ (2,FMT=FORM,END=50) N,A,B</pre>
Example 2:
<pre>CHARACTER AR(2)*10 DATA AR/'(10X,2I2,1','0X,F6.2)'/ READ (5,AR) I,J,X</pre>
is equivalent to
<pre>READ (5,100) I,J,X 100 FORMAT (10X,2I2,10X,F6.2)</pre>

Figure 5-7. Character Format Specification Examples

If a format specification is contained in a character array, the specification may cross element boundaries. Only the array name need be specified in the input/output statement; all information up to the closing parenthesis is considered to be part of the format specification. See figure 5-7 example 2.

NONCHARACTER FORMAT SPECIFICATIONS

Format specifications can be contained in a non-character array. The rules for noncharacter format specifications are the same as for character format specifications.

EDIT DESCRIPTORS

Edit descriptors specify the data conversions to be performed. Table 5-1 describes the repeatable edit descriptors; table 5-2 describes the nonrepeatable edit descriptors.

TABLE 5-1. REPEATABLE EDIT DESCRIPTORS

Descriptor Type	Descriptor	Description
Character	A	Character with data-dependent length
Character or Boolean	Aw	Character or Boolean with specified length
Numeric	Dw.d	Double precision floating-point with exponent
	Ew.d	Single precision floating-point with exponent
	Ew.dEe	Single precision floating-point with explicitly specified exponent length
	Fw.d	Single precision floating-point without exponent
	Gw.d	Single precision floating-point with or without exponent
	Gw.dEe	Single precision floating-point with or without explicitly specified exponent length
	Iw	Decimal integer
	Iw.m	Decimal integer with minimum number of digits
Logical	Lw	Logical
Boolean	Qw	Octal integer
	Qw.m	Octal integer with leading zeros and minimum number of digits
	Rw	Boolean conversion
	Zw	Hexadecimal integer
	Zw.m	Hexadecimal with leading zeros and minimum number of digits

In both tables, uppercase letters indicate the type of conversion. Lowercase letters indicate user-supplied information that has the following meaning:

- w Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading blanks, + or - signs, decimal point, and exponent.

TABLE 5-2. NONREPEATABLE EDIT DESCRIPTORS

Descriptor Type	Descriptor	Description
Numeric input control	BN	Blanks ignored
	BZ	Blanks treated as zeros
Scale factor	kP	Scaling for numeric editing
Hollerith	nH	Output Hollerith string
Character output	"	Output character string
	'	Output character string
Skip spaces	nX	Position forward
Numeric output control	SP	Plus signs (+) produced
	SS	Plus signs (+) suppressed
	S	Plus signs (+) suppressed
Tabulation Control	Tn	Position forward or backward
	TRn	Position forward
	TLn	Position backward
Format control	:	Terminate format control
End of record	/	Indicates end of current input or output record

- d Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded.
- e Nonzero unsigned integer constant specifying the number of digits in the exponent; the value of e cannot exceed 6.
- m Unsigned integer constant specifying the minimum number of digits to be output.
- k Integer constant scale factor.
- n Positive nonzero decimal integer.

In table 5-1, the field width w must be specified for all conversion codes except A.

The following paragraphs discuss input/output conversions, field separators, repeatable and nonrepeatable edit descriptors, and repetition factors.

Input/Output Conversions

For the D, E, F, and G input conversions, a decimal point in the input field overrides the decimal point specification of the field descriptor.

Leading blanks are not significant in numeric input conversions; other blanks in numeric conversions are ignored unless `BLANK='ZERO'` is specified for the file on an OPEN statement or a BZ edit descriptor is in effect. Plus signs can be omitted. An all-blank field is considered to be zero, except for logical input, where an all-blank field is considered to be FALSE.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field unless `w.m` is specified, in which case leading zeros are produced as necessary. The number of characters produced by an output conversion must not be greater than the field width. If the field width is exceeded, asterisks are inserted throughout the field.

Complex data items are converted on input/output as two independent floating-point quantities. The format specification uses two conversion elements. See figure 5-8 for an example.

```

COMPLEX A,B,C,D
WRITE (6,10) A
10 FORMAT (F7.2,E8.2)
READ (5,11) B,C,D
11 FORMAT (2E10.3,2(F8.3,F4.1))
    
```

Figure 5-8. Complex Data Item Format Specification Example

Different types of data can be read by the same `FORMAT` specification. For example,

```
10 FORMAT (I5,F15.2)
```

specifies two values: the first of type integer, the second of type real.

Example:

```

CHARACTER R*4
READ (5,15) NO,NONE,INK,A,B,R
15 FORMAT (3I5,2F7.2,A4)
    
```

reads three integer values, two real values, and one character string.

Field Separators

Field separators are used to separate descriptors and groups of descriptors. The format field separators are the slash (/), the comma, and the colon. The slash is also used to specify demarcation of formatted records.

Repeatable and Nonrepeatable Edit Descriptors

The repeatable edit descriptors are used to specify numeric, logical, character, or Boolean data conversions. The repeatable edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. The repeatable edit descriptors are A, D, E, F, G, I, L, O, R, and Z.

The nonrepeatable edit descriptors are used for numeric input/output control, tabulation control, character output control, format control, end-of-record designation, and scaling for numeric editing. The nonrepeatable edit descriptors cannot be repeated. The nonrepeatable edit descriptors are ' and ", BN, BZ, :, /, nH, kP, S, SP, SS, Tn, TLn, and TRn.

The following paragraphs discuss the repeatable and nonrepeatable edit descriptors in alphabetical order.

A Descriptor

The A descriptor is used with an input/output list item of type character or noncharacter. The following paragraphs discuss the A descriptor for input/output list items of type character and noncharacter.

The form of the A descriptor for character list items is:

A or Aw

On input, if `w` is less than the length of the list item, the input quantity is stored left-justified in the item; the remainder of the item is filled with blanks. If `w` is greater than the length of the item, the rightmost characters are stored and the remaining characters are ignored. If `w` is omitted, the length of the field is equal to the length of the list item. See figure 5-9 for examples.

On output, if `w` is less than the length of the list item, the leftmost characters in the item are output. For example, if a variable A, declared `CHARACTER A*8`, contains

SAMPLEΔΔ

and A is output with the statement

```
WRITE (6,'(1X,A4)')A
```

then the characters `SAMP` are output.

If `w` is greater than the length of the list item, the characters are output right-justified in the field, with blanks on the left. For example, if A in the previous example is output with the statements

```

WRITE (6,400)A
400 FORMAT (1X,A12)
    
```

output is as follows:

ΔΔΔΔSAMPLEΔΔ

Example 1 (character list item):

```
CHARACTER A*9
READ (5,100) A
100 FORMAT (A7)
```

Input record:

EXAMPLE

In location A:

EXAMPLEΔΔ

Example 2:

```
CHARACTER B*10
READ (5,200)B
200 FORMAT (A13)
```

Input record:

1 13
SPECIFICATION

In location B:

1 10
CIFICATION

Example 3:

```
CHARACTER P*8,Q*12,R*9
READ (5,10) P,Q,R
10 FORMAT (A8,A12,A5)
```

Input record:

THIS IS AN EXAMPLE I KNOW
8 12 5

In storage:

P THISΔISΔ
Q ANΔEXAMPLEΔI
R ΔKNOWΔΔΔΔ

Example 4:

```
CHARACTER NAME*30,PHONE*7
READ (5,'(A,A)') NAME,PHONE
```

Note that if no length is specified for an A edit descriptor, the length of the list item is used.

Figure 5-9. A Input Examples

If w is omitted, the length of the character list item determines the length of the output field.

The form of the A descriptor for noncharacter list items is:

Aw

When the A descriptor is used with an input/output list item of noncharacter type, character code conversion (appendix A) is performed. The field width w must appear; w characters are converted.

On input, if w is less than or equal to 10 (there are 10 characters per word), the w characters of

the input item are converted to character code and stored left-justified in the word with blank fill on the right. If w is greater than 10, the rightmost 10 characters of the input item are converted and stored.

On output, if w is less than or equal to 10, the leftmost w characters of the output item are written to the output record. If w is greater than 10, the output item is right-justified in the field and preceded by blanks.

Apostrophe and Quote Descriptors

Character strings delimited by a pair of apostrophe (') or quote (") symbols can be used as alternate forms of the H specification for output. See figure 5-10 for examples. The paired symbols delineate the string. If the string is empty or invalidly delimited, a fatal compilation error occurs and an error message is printed. The apostrophe and quote descriptors must not be used on input.

Example 1:

Source statements:

```
PRINT 10
10 FORMAT ('Δ SUBTOTALS')
```

Output (before printing):

ΔSUBTOTALS

Source statements:

```
WRITE (6,20)
20 FORMAT ('Δ RESULT OF CALCULATIONS IS Δ',
+'AS FOLLOWS')
```

Output (before printing):

ΔRESULT OF CALCULATIONS IS AS FOLLOWS

Example 2:

Source statements:

```
PRINT 1
1 FORMAT ("Δ ABC'DE")
```

Output (before printing):

ΔABC'DE

Source statements:

```
PRINT 3
3 FORMAT('Δ DON' 'T')
```

Output (before printing):

ΔDON'T

Note that on some printers " is output as ≠ and ' is output as † .

Figure 5-10. Apostrophe and Quote Descriptor Examples

NOTE

It is preferable to use the apostrophe descriptor instead of the quote descriptor.

An apostrophe or quote within a string delimited by the same symbol can be represented by two consecutive occurrences of the symbol. Alternatively, if a quote or apostrophe appears within a string, the other symbol can be used as the delimiter. See figure 5-10 example 2.

BN and BZ Blank Interpretation

The nonrepeatable BN and BZ edit descriptors can be used with the repeatable D, E, F, G, and I edit descriptors, on input, to specify the interpretation of blanks (other than leading blanks). In the absence of a BN or BZ descriptor, blanks in input fields are interpreted as zeros or are ignored. Their interpretation depends on the value of the BLANK= parameter in the OPEN statement that is currently in effect for the input/output unit. BLANK='NULL' (blanks ignored) is the default for input. If a BN descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are ignored; that is, the field is treated as if blanks had been removed, the remaining portion of the field right-justified, and the field padded with leading blanks. A field of all blanks has a value of zero.

If a BZ descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are interpreted as zeros.

For example, assuming BLANK='NULL', if the statement

```
READ (6, '(I5, BZ, I3, BN, I3)') I, J, K
```

reads the input record

```
1ΔΔ2ΔΔ3ΔΔ
```

I, J, and K are assigned the following values:

```
I = 1   J = 200   K = 3
```

Carriage Control Character

The carriage control character is the first character of a printer output record and is not printed. It appears in other forms of output as data. Carriage control also applies to records listed at a terminal under INTERCOM; the meaning of carriage control characters depends on the type of terminal. (See the INTERCOM reference manual.) Carriage control does not apply to records listed at a terminal under the NOS time-sharing system; for this system, the first character is listed as data.

The carriage control characters are shown in table 5-3.

For output directed to the card punch or any device other than the line printer or terminal, control characters are not required. If carriage control characters are transmitted to the card punch, they are punched in column one.

Carriage control characters are required at the beginning of every record to be printed, including

TABLE 5-3. PRINTER CONTROL CHARACTERS

Character	Action
Blank	Space vertically one line, then print.
0	Space vertically two lines, then print.
1	Eject to the first line of the next page before printing.
+	No advance before printing; allows overprinting.
Any other character	Refer to the operating system reference manual.

new records introduced by means of a slash. Carriage control characters can be generated by any means. See figure 5-11 for examples.

```
10 FORMAT (1H0,F7.3,I2,G12.6)
20 FORMAT ('Δ',I5,'RESULT=',F8.4)
30 FORMAT ('1',I4,2(F7.3))
40 FORMAT (1X,I4,G16.8)
```

Figure 5-11. Printer Carriage Control Examples

Figure 5-12 shows a program using carriage control characters; the resulting output produces a tic tac toe diagram.

D Descriptor

The D descriptor specifies conversion between an internal double precision real number and an external floating-point number written with an exponent. This descriptor has the form:

```
Dw.d
```

NOTE

The E descriptor is preferred over the D descriptor.

On input, D editing corresponds to E editing and can be used to input all the same forms as E.

The diagram in figure 5-13 illustrates the structure of the input field. It shows the characters allowed to start a subfield.

On output, type D conversion is used to output double precision values. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed. Further description about how indefinite and infinite values are generated and how they are represented on various computer models can be found in the glossary under the appropriate definition. See figure 5-14 for examples of D output.

Example:

```

PROGRAM CHARCON
PRINT 10
FORMAT('1', 5X, 'HERE WE ARE AT THE TOP OF A NEW PAGE')
PRINT 20
FORMAT(3(/))
C
DO 50 I=2, 8
  IF (I .EQ. 4 .OR. I .EQ. 6) THEN
    PRINT 30
    FORMAT(20X, 'XXXXXXXXXX '/'+', 19X, ' ===== ')
  ELSE
    PRINT 40
    FORMAT(21X, '  X X ', '/'+', 20X, '  = = ')
  ENDIF
CONTINUE
C
PRINT 60
FORMAT('0', 5X, 'BEGIN TIC TAC TOE ')
STOP
END

```

Output after printing on a line printer:

HERE WE ARE AT THE TOP OF A NEW PAGE

```

  * *
  * *
*****
  * *
*****
  * *
  * *

```

BEGIN TIC TAC TOE

A '1' specification causes the first output line to appear at the top of a page. FORMAT statement 20 causes three lines to be skipped. In FORMAT statements 30 and 40, a slash skips to the next output record and a plus character causes the record to begin on the same line as the previous record, resulting in overprinting of a row of X characters and = characters. FORMAT statement 60 uses a '0' specification to skip two lines before writing the last output line.

Figure 5-12. Carriage Control Program Example

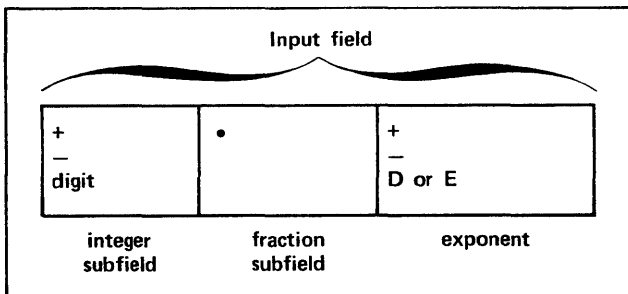


Figure 5-13. D Input Field

```

DOUBLE PRECISION A,B,C
A = 111111.11111D0
B = 222222.22222D0
C = A + B
WRITE (2,10) A,B,C
10 FORMAT (3D23.11)

```

produces output of:

```

.1111111111D+06      .2222222222D+06
.3333333333D+06

```

Figure 5-14. D Output Example

The specification Dw.d produces output in the following format:

s.a+eee

For values where the magnitude of the exponent exceeds 100

s.aD+ee

For values where the magnitude of the exponent is less than 100

s

Minus sign if the number is negative, or blank if the number is positive

a

One or more most significant digits

ee

Digits in the exponent

E Descriptor

The E descriptor specifies conversion between an internal real or double precision value and an external number written with an exponent. This descriptor has the forms:

Ew.d Ew.dEe

On input, the width w includes plus or minus signs, digits, decimal point, E, and exponent. If an external decimal point is not provided, d acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is:

(integer subfield) X 10^{-d} X 10 (exponent subfield)

For example, if the specification is E10.8, the input quantity 3267E+05 is converted and stored as: 3267 X 10⁻⁸ X 10⁵ = 3.267.

If an external decimal point is provided, it overrides d; e, if specified, has no effect on input. An input field consisting entirely of blanks is interpreted as zero.

The diagram in figure 5-15 illustrates the structure of the E input field. It shows the characters allowed to start a subfield.

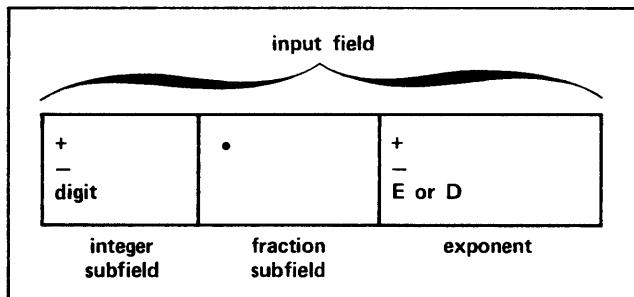


Figure 5-15. E Input Field

The integer subfield begins with a + or - sign, a digit, or a blank; and it can contain a string of digits. The integer field is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, -, or the end of the input field. It can contain a string of digits.

The exponent subfield can begin with E, + or -. When it begins with E, the + is optional between E and the string of digits in the subfield. For example, the following are valid equivalent forms for the exponent 3:

E+ 03 E 03 E03 E3 +3

The range, in absolute value, of permissible values is approximately 10⁻²⁹³ to 10³²². Numbers below the range are treated as zero; numbers above the range cause a fatal error message.

Valid subfield combinations are as follows:

+1.6327E-04

Integer-fraction-exponent

-32.7216

Integer-fraction

+328+5

Integer-exponent

.629E-1

Fraction-exponent

+136

Integer only

136

Integer only

.07628431

Fraction only

E-06 (interpreted as zero)

Exponent only

If the field length specified by w in Ew.d is not the same as the length of the field containing the input number, incorrect numbers might be read, converted, and stored.

The example in figure 5-16 illustrates a situation where numbers are read incorrectly, converted, and stored; yet there is no immediate indication that an error has occurred. First, +647E-01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number -2.36+5 is read instead of -2.36. The specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input number. Since the second speci-

```

OPEN (3,BLANK='ZERO')
READ (3,20) A,B,C
20 FORMAT (E9.3,E7.2,E10.3)

```

On the input record, quantities are in three adjacent fields, columns 1 through 24:

```

+6.47E-01 | -2.36 | +5.321E+02
  9       5       10

```

would be read as:

```

  9       7       10
+6.47E-01 |
          -2.36+5 |
                    .321E+02ΔΔ

```

Figure 5-16. Example Showing E Input Incorrectly Read

cation incorrectly took two digits from the third number, the specification for the third number is now incorrect. The field .321E+02ΔΔ is read. The OPEN statement specifies that trailing blanks are to be treated as zeros; therefore the number .321E+0200 is read converted and placed in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Some additional examples of Ew.d input specifications are shown in table 5-4.

On output, the width w must be sufficient to contain digits, plus or minus signs, decimal point, E, the exponent, and blanks. Generally, w must be at least (d+6) or (d+e+4) for negative numbers, and w must be at least (d+5) or (d+e+3) for positive

numbers. Positive numbers need not reserve a space for the sign of the number unless an SP specification is in effect. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is out of range, an R is printed. Further description about how indefinite and infinite values are generated and how they are represented on various computer models can be found in the glossary under appropriate definition.

The Ew.d specification produces output in the following formats:

s.a...aE ± ee

For values where the magnitude of the exponent is less than 100

s.a...a ± eee

For values where the magnitude of the exponent exceeds 100

s

Is a minus sign if the number is negative, and omitted if the number is positive

a...a

Are the most significant digits of the value correctly rounded

When the specification Ew.dEe is used, the exponent is preceded by E, and the number of digits used for the exponent field not counting the letter and sign is determined by e. If e is specified too small for the value being output, the entire field width as specified by w will be filled with asterisks.

If an integer variable is output under the Ew.d specification, results are unpredictable since the internal formats of real and integer values differ. An integer value normally does not have an exponent and will be printed, therefore, as a very small value or 0.0. See figure 5-17 for examples.

TABLE 5-4. Ew.d INPUT EXAMPLES

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	0.14326	All subfields present.
327.625	E7.3	327.625	No exponent subfield.
-.0003627+5	E11.7	-36.27	Integer subfield only a minus sign and a plus sign appears instead of E.
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only.
blanks	E4.1	0.	All subfields empty.
E+06	E10.6	0.	No integer or fraction subfield: zero stored regardless of exponent field contents.

```

Example 1:

Source data:

    -67.32 or +67.32

WRITE (2,10) A
10 FORMAT (E9.3)

Output (before printing):

    -.673E+02 or Δ.673E+02

Example 2:

Source data:

    -67.32 or +67.32

WRITE (2,10) A
10 FORMAT (E12.3)

Output (before printing):

    ΔΔΔ-.673E+02 or ΔΔΔΔ.673E+02

```

Figure 5-17. Integer Variable Ew.d Specification Examples

End-of-Record Slash

The slash indicates the end of a record anywhere in the FORMAT specification. When a slash is used to separate edit descriptors, a comma is allowed but not required. Consecutive slashes can be used and need not be separated from other elements by commas. When a slash is the last format specification to be processed, it causes a blank record to be written on output or an input record to be skipped. Normally, the slash indicates the end of a record during output and specifies that further data comes from the next record during input. See figure 5-18 for examples.

F Descriptor

The F descriptor specifies conversion between an internal real or double precision number and an external floating-point number without an exponent. This descriptor has the form:

Fw.d

On input, the F specification is treated the same as the E specification. See table 5-5 for examples of F input.

On output, the F descriptor outputs a real number without a decimal exponent.

```

Example 1:

WRITE (2,10)
10 FORMAT (6X,7HHEADING///1X,5HINPUT,
+7H Δ OUTPUT)

Output (before printing):

ΔΔΔΔΔ HEADING
[blank line]
[blank line]
ΔINPUT OUTPUT

Each line corresponds to a formatted record. The second and third records are blank.

Example 2:

I=5
J=6
K=7
WRITE (2,1) I,J,K
1 FORMAT (3I5/F10.4)
WRITE (2,2)
2 FORMAT ('Δ A BLANK LINE SHOULD PRECEDE Δ',
+THIS LINE')

Output (before printing):

ΔΔΔΔ5ΔΔΔΔ6ΔΔΔΔ7
[blank line]
ΔA 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; F10.4 is ignored because there is nothing to be converted.

```

Figure 5-18. End-of-Record Slash Examples (Sheet 1 of 2)

Example 3:

```
DIMENSION B(3)
READ (5,100) IA,B
100 FORMAT (I5/3E7.2)
```

The statements read two records: the first contains an integer number, and the second contains three real numbers.

Example 4:

```
WRITE (3,11) A,B,C,D
11 FORMAT (1X,2E10.2/1X,2F7.3)
```

In storage:

```
A  -11.6
B   .325
C  46.327
D  -14.261
```

Output (before printing):

```
ΔΔΔ-.12E+02ΔΔΔ.33E+00
ΔΔ46.327-14.261
```

Example 5:

```
WRITE (1,11) A,B,C,D
11 FORMAT (1X,2E10.2//1X,2F7.3)
```

Output (before printing):

```
ΔΔΔ-.12E+02ΔΔΔ.33E+00
[blank line]
ΔΔ46.327-14.261
```

The second slash causes the blank line.

Figure 5-18. End-of-Record Slash Examples (Sheet 2 of 2)

TABLE 5-5. F INPUT EXAMPLES

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field.
.62543	F6.5	.62543	No integer subfield.
.62543	F6.2	.62543	Decimal point overrides d of specification.
+144.15E-03	F11.2	.14415	Exponents are allowed in F input.
50000	F5.2	500.00	No fraction subfield; input number converted as 50000×10^{-2} .
ΔΔΔΔΔ	F5.2	0	Blanks in input field interpreted as 0.

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. Further description about how indefinite and infinite values are generated and how they are represented on various computer models can be found in the glossary under the appropriate definition.

The specification Fw.d outputs a number in the following format:

```
sn.n

n   Is a field of decimal digits

s   Is a minus sign if the number is negative,
     or omitted if the number is positive
```

See table 5-6 for examples of F output.

G Descriptor

The G descriptor specifies conversion between an internal real or double precision number and an external floating-point number written either with or without an exponent, depending on the magnitude of the number. This descriptor has the forms:

```
Gw.d   Gw.dEe
```

On input, the G specification is treated the same as the E specification. The rules which apply to the E specification also apply to the G specification. For example,

```
READ (5,11) A,B,C
11  FORMAT (G13.6,2G12.4)
```

On output, results depend on the size of the floating-point number being edited. For values in the range greater than or equal to .1 and less than 10^d the number is output under F format. For values outside this range, Gw.d output is identical to Ew.d and Gw.dEe is identical to Ew.dEe.

If a number is output under the Gw.d specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved

for the exponent field E+ee). Therefore, for output under G conversion, w must be greater than or equal to d+6. The 6 extra spaces are required for sign and decimal point plus four spaces for the exponent field. If the Gw.dEe form is used for a number output without an exponent, then e+2 spaces are inserted to the right of the field. See figure 5-19 for examples.

H Descriptor

The H descriptor is used to output strings of characters. This descriptor is not associated with a variable in the output list. The H descriptor has the form:

```
nHstring

n   Is the number of characters in the
     string including blanks

string Is a string of characters
```

The H descriptor cannot be used on input.

Note that although using apostrophes to designate a character string precludes the need to count characters, the H descriptor may be more convenient if the string contains apostrophes. See figure 5-20 for example.

I Descriptor

The I descriptor specifies integer conversion. This descriptor has the forms:

```
Iw Iw.m
```

On input, the plus sign can be omitted for positive integers. When a sign appears, it must precede the first digit in the field. The Iw and Iw.m specifications have the same effect on input. An all blank field is considered to be zero. Decimal points are not permitted. The value is stored in the specified variable. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will cause an error. See figure 5-21 for examples of I input.

TABLE 5-6. F OUTPUT EXAMPLES

Value of A	FORMAT Statement	PRINT Statement	Output (Before Printing)
+32.694	10 FORMAT (1HΔ,F6.3)	PRINT 10,A	Δ32.694
+32.694	11 FORMAT (1HΔ,F10.3)	PRINT 11,A	ΔΔΔΔΔ32.694
-32.694	12 FORMAT (1HΔ,F6.3)	PRINT 12,A	Δ*****
.32694	13 FORMAT (1HΔ,F4.3,F6.3)	PRINT 13,A,A	Δ.327ΔΔ.327
32.694	14 FORMAT (1HΔ,F6.0)	PRINT 14,A	ΔΔΔΔ 33.

The specification 1HΔ is the carriage control character.

On output, if the integer is positive, the plus sign is suppressed unless an SP specification is in effect. Leading zeros are suppressed.

If Iw.m is used and the output value occupies fewer than m positions, leading zeros are generated to fill up to m digits. If m=0, a zero value will

produce all blanks. If m=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. If the field is too short, asterisks occupy the field.

Figure 5-22 shows some examples of I output.

Example 1:

```

Y=77.132
WRITE (7,200) Y
200 FORMAT (G10.3)

```

Output (before printing):

ΔΔ77.1ΔΔΔΔ

Example 2:

```

EXIT=1214635.1
WRITE (4,100) EXIT
100 FORMAT (G10.3)

```

Output (before printing):

ΔΔ.121E+07

Example 3:

```

READ (5,50) SAMPLE
50 FORMAT (E20.5)
.
.
.
WRITE (6,60) SAMPLE
60 FORMAT (1X,G14.8)

```

Data Read By READ Statement	Output (before printing)	Format Option
.1415926535 E-10	ΔΔ.14159265E-10	E conversion
ΔΔΔ.8979323846	ΔΔ.89793238	F conversion
ΔΔΔ.2643383279.	ΔΔ.26433833E+10	E conversion
ΔΔΔ-693.9937510	Δ-693.99375	F conversion

Figure 5-19. G Output Examples

Source statements:

```

A = 1.5
WRITE (2,30) A
30 FORMAT (6H ΔLMAX=,F5.2)

```

Output (before printing):

ΔLMAX= 1.50

Replacing the H descriptor in the preceding example with ' ΔLMAX=' would produce the same output.

Figure 5-20. H Descriptor Example

```

OPEN (2,BLANK='NULL')
READ (2,10) I,J,K,L,M,N
10 FORMAT (I3,I7,I2,I3,I2,I4)

```

Input Record:

139ΔΔΔΔ-1518ΔΔ7ΔΔΔ1 Δ4

In storage:

I contains 139 L contains 7
J contains -15 M contains 0
K contains 18 N contains 14

If BLANK='ZERO' were specified on the OPEN statement, J would contain -1500 and N would contain 104. Other values would not be affected.

Figure 5-21. I Input Example

Example 1:

```

PRINT 10,I,J,K
10 FORMAT (I9,I10,I5.3)

```

I contains -3762
J contains +4762937
K contains +13

Output (before printing):

ΔΔΔΔ-3762|ΔΔΔ4762937|ΔΔ013|

8 10 5

First blank taken as printer control character.

Example 2:

```

WRITE (6,100)N,M,I
100 FORMAT (I5,I6,I9)

```

N contains +20
M contains -731450
I contains +205

Output (before printing):

ΔΔΔ20|*****|ΔΔΔΔΔΔ205|

4 6 9

First blank taken as printer control character. Specification too small, * indicates field is too short.

Figure 5-22. I Output Examples

L Descriptor

The L descriptor is used to input or output logical items. This descriptor has the form:

Lw

On input, if the first nonblank characters in the field are T or .T, the logical value .TRUE. is stored in the corresponding list item, which should be of type logical. If the first nonblank characters are F or .F, the value .FALSE. is stored. If the first nonblank characters are not T, .T, F, or .F, a diagnostic is printed. An all blank field has the value .FALSE.

On output, variables under the L specification should be of type logical. A value of .TRUE. or .FALSE. in memory is output as a right-justified T or F with blanks on the left. See figure 5-23 for examples of L output.

```
LOGICAL I,J,K
I = .TRUE.
J = .FALSE.
K = .TRUE.
WRITE (4,5) I,J,K
5 FORMAT (3L3)

Output (before printing):

  ΔΔTΔFΔΔT
```

Figure 5-23. L Output Example

O Descriptor

The O descriptor (letter O) is used to input or output items in octal format. This descriptor has the forms:

Ow Ow.m

The form Ow.m means the same as Ow on input. The octal digits are the digits 0 through 7.

On input, the field can contain a maximum of 20 octal digits. Blanks are allowed, and a plus or minus sign can precede the first octal digit. Blanks are interpreted as zeros, and an all blank field is interpreted as zero. A decimal point is not allowed. See figure 5-24 for examples of O input.

On output, if w is less than 20, the rightmost digits are output. For example, if location P contains

```
000000000373737373
```

and the output statements are

```
WRITE (6,100)P
100 FORMAT (1X,04)
```

the digits 3737 are output.

```
BOOLEAN P,Q,R
READ 10,P,Q,R
10 FORMAT (BZ,010,012,02)

Input record:

  3737373737666Δ6644Δ444 -0
      10          12      2

In storage (octal representation):

P: 00000000003737373737
Q: 00000000666066440444
R: 77777777777777777777
```

Figure 5-24. O Input Example

If w is greater than 20, the 20 total digits (20 octal digits = a 60-bit word) are right-justified with blanks on the left. For example, if the contents of location P shown above are output with the statements:

```
WRITE (6,200)P
200 FORMAT (1X,022)
```

output would appear as

```
ΔΔ00000000003737373737
```

A negative number is output in one's complement internal form. For example,

```
I = -11
WRITE (6,200)I
200 FORMAT (1X,022)
```

Output:

```
ΔΔ77777777777777777764
```

If m is specified, the number is printed with leading zeros so that at least m digits are printed, and with a minus sign for negative numbers. If the number cannot be output in w octal digits, all asterisks will fill the field.

The specification Ow produces a string of up to 20 octal digits. Two octal specifications must be used for variables whose type is complex or double precision.

P Descriptor

The P descriptor is used to change the position of a decimal point of a real number when it is input or output. The P descriptor has the form:

kP

where k is a signed or unsigned integer constant called the scale factor.

Scale factors can either precede D, E, F, and G format specifications or appear independently. Forms are as follows:

kPDw.d

kPEw.dEe

```

kPEw.d
kPFw.d
kPGw.d
kP

```

A scale factor of zero is established when each FORMAT specification is first referenced; it holds for all F, E, G, and D field descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G descriptors in that FORMAT specification until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G descriptors a zero scale factor (0P) must be specified. For example,

```
15 FORMAT(2P,E14.3,F10.2,G16.2,0P,4F13.2)
```

The 2P scale factor applies to the E14.3 format specification and also to the F10.2 and G16.2 format specifications. The 0P scale factor restores normal scaling ($10^0 = 1$) for the subsequent specification 4F13.2.

Example:

```
20 FORMAT(3P,5X,E12.6,F10.3,OPD18.7,-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).

On input, for F, E, D, and G editing, the number is divided by 10^k and stored, provided that the number in the input field does not have an exponent. 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.

On output, for F editing, the number in the output field is the internal number multiplied by 10^k . In the output representation, the decimal point is fixed; the number is adjusted to the left or right, depending on whether the scale factor is plus or minus. For example, the internal number -3.1415926536 can be represented on output under scaled F specifications as shown in figure 5-25.

```

.....
(-1PF13.6)    -3.14159
(  F13.6)     -3.141593
( 1PF13.6)    -31.415927
( 3PF13.6)    -3141.592654
.....

```

Figure 5-25. Scaled F Output

For E and D editing, the effect of the scale factor kP is to shift the output coefficient left k places and reduce the exponent by k. In addition, the scale factor controls the decimal normalization

between the coefficient and the exponent such that: if k is less than or equal to 0, there will be exactly -k leading zeros and d+k significant digits after the decimal point; if k is greater than 0, there will be exactly k significant digits to the left of the decimal point and d-k+1 significant digits to the right of the decimal point. For example, the number -3.1415926536 is represented on output under the indicated Ew.d scaling as shown in figure 5-26.

```

.....
(-3PE20.4)    -.0003E+04
(-1PE20.4)    -.0314E+02
(  E20.4)     -.3142E+01
( 1PE20.4)    -3.1416E+00
( 3PE20.4)    -314.16E-02
.....

```

Figure 5-26. Scaled E Output

For G editing, the effect of the scale factor is nullified unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number N is less than 10^{-1} or greater than or equal to 10^d). In these cases, the scale factor has the same effect as described for Ew.d and Dw.d scaling. For example, the numbers -3.1415926536 and -.00031415926536 are represented on output under the indicated Gw.d scaling as shown in figure 5-27.

```

.....
(-3PG20.6)    -3.14159
(-1PG20.6)    -3.14159
(  G20.6)     -3.14159
( 1PG20.6)    -3.14159
( 3PG20.6)    -3.14159
( 5PG20.6)    -3.14159
( 7PG20.6)    -3.14159
.....
.....
(-3PG20.6)    -.000314E+00
(-1PG20.6)    -.031416E-02
(  G20.6)     -.314159E-03
( 1PG20.6)    -3.141593E-04
( 3PG20.6)    -314.1593E-06
( 5PG20.6)    -31415.93E-08
( 7PG20.6)    -3141593.E-10
.....

```

Figure 5-27. Scaled G Output

R Descriptor

The R descriptor is used with noncharacter list items. This descriptor is used to transmit the rightmost characters of a word. The R descriptor has the form:

Rw

On both input and output, the R specification is identical to the A specification, unless w is less than 10.

On input, if w is less than 10, the rightmost w characters are read and stored right-justified with upper binary zero fill. See figure 5-28 for examples of R input.

```

      BOOLEAN HOO,RAY
      READ (5,600) HOO,RAY
      600 FORMAT (R10,R5)

Input record:
           RESULTS OF TEST
                                
           10          5

In storage:
      HOO: RESULTSΔOF
      RAY: 00000ΔTEST
  
```

Figure 5-28. R Input Example

On output, if w is less than 10, the rightmost w characters of the output item are written to the output record.

S, SP, SS Plus Sign Control

The nonrepeatable S, SP and SS edit descriptors can be used on output with the repeatable D, E, F, G, and I edit descriptors to control the printing of plus (+) characters. S, SP and SS have no effect on input.

Normally, FORTRAN does not precede positive numbers by a plus sign on output. If an SP descriptor is encountered in a format specification, all succeeding positive numeric fields will contain the plus sign (w must be of sufficient length to include the sign). If an SS or S descriptor is encountered, the optional plus signs will not appear.

S, SP, and SS have no effect on plus signs preceding exponents, since those signs are always provided. See figure 5-29 for examples of sign control.

```

A = 10.5
B = 7.3
C = 26.0
WRITE (2,'(1X,F6.2,SP,F6.2,SS,F6.2)') A,B,C

Output (before printing):
ΔΔ10.50Δ+7.30Δ26.00
  
```

Figure 5-29. Sign Control Example

T, TL, TR Descriptors

The T, TL, and TR descriptors provide for tabulation control. These descriptors have the forms:

Tn TLn TRn

n Is a nonzero unsigned decimal integer

When a Tn descriptor is encountered in a format specification, input or output control skips right or left to column n; the next edit descriptor is then processed.

When a TLn descriptor is encountered, control skips backward (left) n columns. If n is greater than or equal to the current character position, control skips to the first character position.

When a TRn descriptor is encountered, control skips forward (right) n characters. TRn is the same as nX.

On card input, control can be positioned beyond column 80, but a succeeding descriptor would read only blanks.

See figure 5-30 for tabulation control examples.

```

Example 1:
      READ 40, A,B,C
      40 FORMAT (T2,F5.2,TR5,F6.1,TR3,F5.2)

Input:
Δ684.73ΔΔΔΔΔ2436.2ΔΔΔΔ89.14

A is set to 684.7, B to 2436.0, and C to 89.0.

Example 2:
      WRITE (31,10)
      10 FORMAT (T20,'LABELS')

Control positions to column 20 of the output record and writes the characters LABELS.

Example 3:
      READ (2,'(F5.2,TL5,F5.2)') A,B

Input record:
76.05

The same information is read more than once.
Both A and B contain 76.05.
  
```

Figure 5-30. T, TL, and TR Descriptor Examples

With a T, TR, or TL specification, the order of a list need not be the same as that of the input or output record. The same information can be read more than once as shown in figure 5-33 example 3.

When a T, TR, TL specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; those already filled are left unchanged. An example is shown in figure 5-31.

```

PROGRAM TEST
1  FORMAT(12('0123456789'))
   PRINT 1
   PRINT 60
60  FORMAT(T80,'COMMENTS',T60,'HEADING4',T40,
+     'HEADING3',T20,'HEADING2',T2,'HEADING1')
   PRINT 10
10  FORMAT(20X,'THIS IS THE END OF THIS RUN',T52,'HONEST')
   PRINT 1
   STOP
   END

```

Line Printer Output:

```

12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
HEADING1          HEADING2          HEADING3          HEADING4          COMMENTS
                THIS IS THE END OF THIS RUN    HONEST

12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789

```

For the **FORMAT** statement labeled 60, control passes over the first print position (the one used for carriage control); therefore, it is automatically set to a blank, which causes the line to be single spaced.

Figure 5-31. T Output Example

The following example shows that it is possible to destroy a previously formed field:

```
WRITE (2,8)
8  FORMAT ('DISASTERS',T5,3H123)
```

Output record before printing:

```
DISA123RS
```

If the output record is printed, the first character is not printed. (See Carriage Control Character in this section.)

Termination of Format Control

A colon (:) in a format specification terminates format control if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list. This descriptor is useful in forms where nonlist item edit descriptors follow list item edit descriptors; when the iolist is exhausted, the subsequent edit descriptors are not processed. See figure 5-32 for examples of colon use.

```
A = 1.0
B = 2.2
C = 3.1
D = 5.7
PRINT 10, A, B, C, D
10 FORMAT (4(F4.1,:',',''))
```

Output (before printing):

```
Δ1.0,Δ2.2,Δ3.1,Δ5.7
```

Format control terminates after the value of D is printed, and the last comma is not printed.

Figure 5-32. Termination by Colon Example

X Descriptor

The X descriptor is used to skip character positions in an input line or output line. X is not associated with a variable in the input/output list. The X descriptor has the form:

nX

n Is the number of character positions to be skipped from the current character position; n is a nonzero unsigned integer

The specification nX indicates that transmission of the next character to or from a record is to occur at the position n characters forward from the current position.

When an X specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; however, positions already filled are left unchanged. See figure 5-33 for examples.

Example 1:

Source statements:

```
A = -342.743
B = 1.53190
J = 22
WRITE (6, '(1X,F9.4,4X,F7.5,4X,I3)') A,B,J
```

Output (before printing):

```
Δ-342.7430ΔΔΔΔ1.53190ΔΔΔΔΔ22
```

Example 2:

Source statement:

```
READ (3, '(F5.2,3X,F5.2,6X,F5.2)') R,S,T
```

Input:

```
14.62ΔΔ$13.78 ΔCOST Δ15.97
```

In storage:

```
R 14.62
S 13.78
T 15.97
```

Figure 5-33. X Descriptor Example

Z Descriptor

The Z descriptor is used for hexadecimal conversion. This descriptor has the forms:

Zw Zw.m

The form Zw.m is meaningful for output only. Hexadecimal digits include the digits 0 through 9 and the letters A through F. A hexadecimal digit is represented by 4 bits.

On input, the input string can contain up to 15 hexadecimal digits. Embedded blanks are interpreted as zero, and an all blank field is equivalent to zero. A plus or minus sign can precede the first digit. The string is stored right-justified with zeros on the left. See figure 5-34 for examples of Z input.

```
INTEGER R,S
READ (10, '(BZ,Z10,Z4)')R,S
```

Input record:

```
A309FFFFCC4ΔD1
```

10 4

In storage (hexadecimal representation):

```
R: 0000A309FFFFCC
S: 000000000040D1
```

Figure 5-34. Z Input Example

On output, if *w* is less than 15, the rightmost *w**4 bits are converted to hexadecimal and written. For example, if location *I* contains:

```
0000000000FB26C      (hexadecimal format)
```

then the output statement:

```
WRITE(6, '(IX,Z3)')I
```

writes the digits 26C.

If *w* is greater than 15, the 15 hexadecimal digits are right-justified with blanks on the left.

If *m* is specified, the number is printed with leading zeros so that at least *m* digits are output, and with a minus sign for negative numbers. If the number of hexadecimal digits exceeds *w*, a field of asterisks is written.

Repetition Factors

The repeatable edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. For example,

```
100 FORMAT (3I4,2E7.3)
```

is equivalent to:

```
100 FORMAT (I4,I4,I4,E7.3,E7.3)
```

Also,

```
50 FORMAT (4G12.6)
```

is equivalent to:

```
50 FORMAT (G12.6,G12.6,G12.6,G12.6)
```

A group of descriptors can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor. If no integer precedes the left parenthesis, the repetition factor is 1. For example,

```
1 FORMAT (I3,2(E15.3,F6.1,2I4))
```

is equivalent to the following specification if the number of items in the input/output list does not exceed the number of format conversion codes:

```
1 FORMAT(I3,E15.3,F6.1,I4,I4,E15.3,  
+ F6.1,I4,I4)
```

A maximum of nine levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If there are fewer items in the input/output list than indicated by the format conversions in the FORMAT specification, the excess conversions are ignored.

If the number of items in the input/output list exceeds the number of format conversions when the final right parenthesis in the FORMAT statement is reached, the line formed internally is output. The format control then scans to the left looking for a

right parenthesis within the FORMAT statement. If none is found, the scan stops when it reaches the beginning of the format specification. If a right parenthesis is found, however, the scan continues to the left until it reaches the field separator which precedes the left parenthesis pairing the right parenthesis. Output resumes with the format control moving right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered.

If *n* slashes are indicated, a repetition factor can be used to indicate multiple slashes; *n*-1 lines are skipped on output.

See figure 5-35 for repetition factor examples.

EXECUTION TIME FORMAT SPECIFICATION

Variable format specifications can be read in as part of the data at execution time and used wherever a normal format can be used. The format can be read in under the A specification and stored in a character array, variable, or array element; or it can be included in a DATA statement. Formats can also be generated by the program at execution time.

If an array or array element is used, its type can be other than character, although character is the preferred type. In either case, the format must consist of a list of descriptors and editing characters enclosed in parentheses, but without the keyword FORMAT and the statement label.

The name of the entity containing the specifications is used in place of the FORMAT statement number in the associated input/output statement. The name specifies the location of the first word of the format information.

The run time library checks execution time format specifications as it processes the I/O list. Once the I/O list is complete and format processing is finished, the library ignores the rest of the format specification.

See figure 5-36 for examples of execution time format specifications.

INPUT/OUTPUT STATEMENTS

The following information discusses keyword=value forms in input/output statements, formatted input/output statements, unformatted input/output statements, list directed input/output statements, namelist input/output statements, buffered input/output statements, file status statements, memory-to-memory input/output statements, and file positioning statements.

KEYWORD=VALUE FORMS IN INPUT/OUTPUT STATEMENTS

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

```
UNIT=u
```


Example 1:

```
DIMENSION A(9)
DATA A/3.62,-4.03,-9.78,-6.33,7.12,3.49,6.21,
+6.74,-1.18/
.
.
WRITE (3,15)(A(I),I=1,9)
15 FORMAT (8H RESULTS,4(/),(1X,3F8.2))
```

Format statement 15 is equivalent to:

```
15 FORMAT (8H RESULTS,//// (1X,3F8.2))
```

Output (before printing):

```
RESULTS
[blank line]
[blank line]
[blank line]
      3.62      -4.03      -9.78
     -6.33       7.12       3.49
      6.21      -6.74      -1.18
```

The 4(/) results in 3 blank lines.

Example 2:

```
READ (5,300) I,J,E,K,F,L,M,G,N,R
300 FORMAT (I3,2(I4,F7.3),I7)
```

Data is stored in I with format I3, J with I4, E with F7.3, K with I4, F with F7.3, and L with I7. A new record is then read; data is stored in M with the format I4, G with F7.3, N with I4, and R with F7.3.

Example 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, and KAT is input with I3. The FORMAT statement is exhausted (the right parenthesis has been reached), a new record is read, and the statement is rescanned from the group (F12.7,I3). WAY is input with the format F12.7, NAT with I3, and from a third record, RAY with F12.7, and MAT with I3.

Figure 5-35. Repetition Factor Examples

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

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

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

An integer or Boolean expression having the following characteristics:

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

or

BOOL(u) is a display code name in L format (left-justified with binary zero fill). This is the unit name. If this value is of the form TAPEk, where k is an integer in the range 0 through 999, with no leading zero, it is equivalent to the integer k for the purpose of identifying external units. A valid unit name consists of 1 through 7 letters or digits beginning with a letter.

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

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

Example 1:

Input record:

(E7.2,G20.5,F7.4,I3)

This specification can be read and subsequently referenced as follows:

```
CHARACTER F*30
READ (2,'(A)') F
WRITE (3,F) A,B,C,N
```

Example 2:

Input record:

(E12.2,F8.2,I7,2E20.3,F9.3,I4)

This specification can be read by the statements:

```
CHARACTER VAR*40
READ (2,'(A)') VAR
```

A subsequent output statement in the same program can refer to these format specifications as:

```
WRITE (2,VAR) A,B,I,C,D,E,J
```

which produces the same result as the statements:

```
WRITE (2,10) A,B,I,C,D,E,J
10 FORMAT (E12.2,F8.2,I7,2E20.3,F9.3,I4)
```

Example 3:

```
CHARACTER FMT*9
DATA FMT/'(1X,3I10)'/
C REMOVE CARRIAGE CONTROL CHARACTER
C IF NOT GOING TO PRINTER
IF (PRTFLG .EQ. 0) FMT (2:4)=' '
WRITE (2,FMT) I,J,K
```

If PRTFLG is zero, the program produces the same result as WRITE (2,'(3I10)') I,J,K.

Figure 5-36. Execution Time Format Specification Examples

FMT=fn

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

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

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

A noncharacter array containing the format specification.

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

An asterisk, indicating list directed input/output.

A namelist group name

A character constant whose value is a format specification.

When fn is specified as a constant or symbolic constant, it is not checked for correctness during compilation; it is passed, as is, to the run time library.

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

REC=rn

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

END=s1

Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation. END= is ignored for direct access input operations.

Records following an end-of-file can be read by either issuing a CLOSE statement followed by an OPEN statement on the file, or by using the EOF function. CLOSE/OPEN is the preferred method.

ERR=s1

Specifies the label of an executable statement to which control transfers if a parity error condition is encountered during input/output processing. ERR= has no effect on the processing of errors 78 and 79 in data input from a connected (terminal) file.

IOSTAT=ios

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

- <0 End-of-file
- =0 Operation completed normally
- >0 Number of error condition detected (see table B-5 in appendix B.)
- >1000 CRM error; the rightmost 3 digits correspond to an octal error code in the CYBER Record Manager reference manual. For example, error number 1355 corresponds to CRM error number 355.

Records following an end-of-file can be read by either issuing a CLOSE statement followed by an OPEN statement on the file, or by using the EOF function. CLOSE/OPEN is the preferred method.

iolist

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

FORMATTED INPUT/OUTPUT STATEMENTS

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

The formatted input/output statements are READ, WRITE, PRINT, and PUNCH. They are discussed below.

Formatted READ

The formatted READ statement (figure 5-37 and examples in figure 5-38) transmits data from either

unit *u* or the unit INPUT to storage locations named in *iolist* according to FORMAT specification *fn*.

The number of items in the *iolist* and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, one or more input records will be bypassed. The number of records bypassed is one plus the number of slashes interpreted in the FORMAT statement.

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

Formatted WRITE

The formatted WRITE statement (figure 5-39 and example in figure 5-40) transfers information from the storage locations named in the input/output list to the unit specified by *u*, according to the FORMAT specification *fn*.

```

READ ( { u,fn
        { u,FMT=fn
        { UNIT=u,FMT=fn } [ ,IOSTAT=ios][ ,ERR=s][ ,END=s[ ] ] [iolist]

READ fn[,iolist]

```

Figure 5-37. Formatted READ Statement

Example 1:

```

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

```

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

Example 2:

```

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

```

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

Example 3:

```

PROGRAM READ
OPEN (2, FILE='INPUT')
OPEN (3, FILE='OUTPUT')
READ (2,100,ERR=16,END=18) A,B
100 FORMAT (2F10.4)
C=A+B
PRINT *,A,B,C
STOP
16 PRINT 101
101 FORMAT ('ΔI/O ERROR')
STOP
18 PRINT 102
102 FORMAT ('ΔEND OF FILE')
STOP
END

```

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

Example 4:

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

```

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

```

Figure 5-38. Formatted READ Statement Examples

$$\text{WRITE} \left(\left(\begin{array}{l} u, fn \\ u, \text{FMT}=fn \\ \text{UNIT}=u, \text{FMT}=fn \end{array} \right) \left[\text{C}, \text{IOSTAT}=\text{ios} \right] \left[\text{ERR}=\text{sl} \right] \right) \left[\text{iolist} \right]$$

Figure 5-39. Formatted WRITE Statement

```

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

```

Figure 5-40. Formatted WRITE Statement Example

Examples:

```

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

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

```

The format specification can appear in the WRITE statement. For example,

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

Formatted PRINT

The formatted PRINT statement (figure 5-41 and example in figure 5-42) transfers information from the storage locations named in the input/output list to the file named **OUTPUT** according to the specified format. At the end of a batch job, file **OUTPUT** is normally sent to the printer.

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

Figure 5-41. Formatted PRINT Statement

```

PROGRAM PRINT
CHARACTER B*3
A=1.2
B='YES'
N=19
PRINT 4,A,B,N
4 FORMAT (G20.6,A,I5)
STOP
END

```

Figure 5-42. Formatted PRINT Example

Examples:

```

PRINT 4,A,B,N
4 FORMAT(G20.6,A,I5)

PRINT 50
50 FORMAT (' END OF FILE')

```

Formatted PUNCH

The formatted PUNCH statement (figure 5-43 and example in figure 5-44) transfers data from the storage locations specified by iolist to the file **PUNCH**. At the end of a batch job the file **PUNCH** is output on the standard punch unit as Hollerith codes, 80 characters or fewer per card in accordance with format specification *fn*. If the card image is longer than 80 characters, additional cards are punched with the remaining characters.

```
PUNCH fn [,iolist]
```

Figure 5-43. Formatted PUNCH Statement

```

PROGRAM PUNCH
C = 5.3
D = 'NO'
K = 15
PUNCH 10 C, D, K
10 FORMAT(G10.2, A, I6)
STOP
END

```

Figure 5-44. Formatted PUNCH Example

Examples:

```

PUNCH 5,A,B,C,ANSWER
5 FORMAT (3G12.6,G20.6)

PUNCH 30
30 FORMAT ('LAST CARD')

```

UNFORMATTED INPUT/OUTPUT STATEMENTS

Unformatted READ and WRITE statements do not use format specifications and do not convert data in any way on input or output. Instead, data is transferred as is between memory and the external device. Each unformatted input/output statement transfers exactly one record. If data is written by an unformatted WRITE and subsequently read by an unformatted READ, exactly what was written is read; no precision is lost since no conversion is performed.

Unformatted READ

The unformatted READ statement (figure 5-45 and example in figure 5-46) transmits one record from the specified unit *u* to the storage locations named in *iolist*. Records are not converted; no FORMAT statement is used. The information is transmitted from the designated file in the form in which it exists on the file without any conversion. If the number of words in the list exceeds the number of words in the record, an execution diagnostic results. If the number of locations specified in *iolist* is less than the number of words in the record, the excess data is ignored. If *iolist* is omitted, the unformatted READ skips one record.

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

Figure 5-45. Unformatted READ Statement

```
PROGRAM AREAD
READ (2,END=30,ERR=40) X,Y,Z
SUM = X+Y+Z/2.
.
.
.
END
```

Figure 5-46. Unformatted READ Example

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

Unformatted WRITE

The unformatted WRITE statement (figure 5-47 and example in figure 5-48) is used to output binary records. Information is transferred from the items *iolist* to the specified output unit *u* with no format conversion. One record is created by an unformatted WRITE statement. If the list is omitted, the statement writes a null record on the output device. A null record has no data but contains all other properties of a legitimate record.

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

Figure 5-47. Unformatted WRITE Statement

```
PROGRAM OUT
DIMENSION A(260), B(4000)
.
.
.
WRITE (10,ERR=16) A,B
END
```

The 4260 words of arrays A and B are written as one record on unit 10.

Figure 5-48. Unformatted WRITE Example

LIST DIRECTED INPUT/OUTPUT STATEMENTS

List directed input/output involves the processing of coded records without a FORMAT statement. Each record consists of a list of values in a freer format than is used for formatted input/output. This type of input/output is particularly convenient when the exact form of data is not important.

List Directed Input

The list directed READ statement (figure 5-49) transmits data from unit *u* or the file INPUT (if *u* is omitted or if UNIT= * specified) to the storage locations named in *iolist*. The input data items are free-form with separators rather than in fixed-size fields.

A list directed READ following a list directed READ that terminated in the middle of a record starts with the next data record.

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

See figure 5-50 for an example of list directed READ input.

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

Figure 5-49. List Directed READ Statement

Example 1:	
	<pre> PROGRAM LDR READ *, CAT, BIRD, DOG PRINT 10, CAT, BIRD, DOG 10 FORMAT(' CAT = ', E9.3,/' BIRD = ', E9.3,/' DOG = ',E9.3) STOP END </pre>
Input:	Output (after printing by a line printer):
13.3, -5.2, .01	<pre> CAT = .133E+02 BIRD = -.520E+01 DOG = .100E-01 </pre>
Example 2:	
	<pre> PROGRAM LDIN 100 READ(*,*,END=99) J,K PRINT 1, J,K 1 FORMAT (T20,2I5) GO TO 100 99 STOP END </pre>
Input:	Output (after printing by a line printer):
1 2	1 2
,3	1 3
,,	1 3
2*8	8 8
/9	8 8

Figure 5-50. List Directed Input Examples

Input data consists of a string of values separated by one or more blanks, or by a comma or slash, either of which can be preceded or followed by any number of blanks. Also, a line boundary, such as end-of-record or end-of-card, serves as a value separator; however, a separator adjacent to a line boundary does not indicate a null value.

Embedded blanks are not allowed in input values, except character values and complex numbers. The format of values in the input record is as follows:

Integers

Same format as for integer constants.

Real numbers

Any valid FORTRAN format for real or double precision numbers. In addition, the decimal point can be omitted; it is assumed to be to the right of the mantissa.

Complex numbers

Two real values, separated by a comma, and enclosed by parentheses. The parentheses are not considered to be a separator. The decimal point can be omitted from either of the real constants. Each of the real values can be preceded or followed by blanks.

Character values

A string of characters (which can include blanks) enclosed by apostrophes. A delimiting apostrophe can be represented within a string by two successive occurrences. Character values can only be read into character arrays, variables and substrings. If the string length exceeds the length of the list item, the string is truncated. If the string is shorter than the list item, the string is left-justified and remaining character positions are blank filled.

Logical values

An optional period, followed by a T or F, followed by optional characters which do not include separators (slashes or commas).

A Boolean constant can be input only if the corresponding list item is of type Boolean. These include:

Octal constants.

Hexadecimal constants.

Hollerith constants containing one through 10 characters and delimited by quotes. Constants of less than 10 characters are left-justified with blank fill on the right. Strings of greater than 10 characters are truncated to 10 characters.

In addition, real and integer values can be read into Boolean variables.

To repeat a value, an integer repeat constant is followed by an asterisk and the constant to be repeated. Blanks cannot be embedded in the repeat part of the specification.

A null can be input in place of a constant when the value of the corresponding list entity is not to be changed. A null is indicated by the first character in the input string being a comma or by two commas separated by an arbitrary number of blanks. Nulls can be repeated by specifying an integer repeat count followed by an asterisk and any value separator. The next value begins immediately after a repeated null. A null cannot be used for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, remaining list elements are treated as nulls and the remainder of the current record is discarded.

Input values must correspond in type to variables in the input/output list. Note that the form of a real value can be the same as that of an integer value.

List Directed Output

The list directed output statements are a WRITE (figure 5-51), a PRINT (figure 5-52), and a PUNCH (figure 5-53) statement. See figure 5-54 for examples.

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

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

Figure 5-51. List Directed WRITE Statement

```
PRINT*[ ,iolist]
```

Figure 5-52. List Directed PRINT Statement

```
PUNCH*[ ,iolist]
```

Figure 5-53. List Directed PUNCH Statement

```
Example 1:
PROGRAM LDW
INTEGER J(4)
COMPLEX Z(2)
DOUBLE PRECISION Q
DATA J,Z,Q /1,-2,3,-4,(7.,-1.),(-3.,2.),1.D-5/
PRINT *, J
PRINT *, Z,Q
STOP
END

Output (after printing by a line printer):
1 -2 3 -4
(7.,-1.) (-3.,2.) .00001

Example 2:
PROGRAM K
PRINT *, 'TYPE IN X'
READ *, X
PRINT *, 'TYPE IN Y'
READ *, Y
END

Terminal listing under NOS:
TYPE IN X
? 1.234
TYPE IN Y
? 5.678
```

Figure 5-54. List Directed Output Examples

PRINT outputs data to the unit OUTPUT. PUNCH outputs to the unit PUNCH.

List directed output is consistent with the input; however, null values, slashes, repeated constants, and the apostrophes used to indicate character values are not produced. For real or double precision variables with absolute values in the range of 10^{-6} to 10^9 , an F format type of conversion is used; otherwise, output is of the LPE type. Trailing zeros in the mantissa and leading zeros in the exponent are suppressed. Values are separated by blanks.

List directed output statements always produce a blank for carriage control as the first character of the output record.

Logical values are output as T or F. Complex values are enclosed in parentheses with a comma separating the real and imaginary parts.

Boolean values are output in the form $0^n[n]...$, where n is an octal digit. Leading zeros are suppressed.

On a connected file under NOS, if the iolist of a list directed output statement ends with a comma, no carriage control or line feed takes place after the line is output. Under NOS/BE and SCOPE 2, a comma as the last character of an iolist is ignored.

(Use of P-type records is not supported with list directed output.)

NAMelist INPUT/OUTPUT STATEMENTS

The NAMelist statement (figure 5-55 and example in figure 5-56) permits input and output of groups of variables and arrays with an identifying name. No format specification is used. The NAMelist statement is a nonexecutable statement that appears in the program following the declarative portion.

NAMelist/name/a[a]...[/name/a[a]...]...	
name	Symbolic group name which must be enclosed in slashes and must be unique within the program unit.
a	Variable or array name.

Figure 5-55. NAMelist Statement

The namelist group name identifies the succeeding list of variables or array names.

A namelist group name must be declared in a NAMelist statement before it is used in an input/output statement. The group name can be declared only once, and it cannot be used for any purpose other than a namelist name in the program unit. It can appear in READ, WRITE, PRINT, and PUNCH statements in place of the format designator. When a namelist group name is used, the list must be omitted from the input/output statement.

A variable or array name can belong to one or more namelist groups. Assumed size arrays cannot appear in a namelist group.

Example:

```
PROGRAM NMLIST
NAMELIST /SHIP/ A,B,C,I1,I2
READ(*, SHIP,END=10)
IF(C .GT. 0.0) THEN
A=B+C
I1=I1+I2
WRITE(*, SHIP)
ENDIF
STOP
10 PRINT *, 'NO DATA FOUND'
STOP
END
```

Input record:

Δ\$SHIP A=14.7,B=12.3,C=3.4,I1=58,I2=8\$

Output:

```
Δ$SHIP
ΔA      = .157E+02,
ΔB      = .123E+02,
ΔC      = .34E+01,
ΔI1     = 66,
ΔI2     = 8,
Δ$END
```

Figure 5-56. NAMelist Example

Data read by a single namelist name READ statement must contain only names listed in the referenced namelist group. All items in the namelist group, or any subset of the group, can be input. Values are unchanged for items not input. Variables need not be in the order in which they appear in the defining NAMelist statement.

The following paragraphs discuss namelist input and output and arrays in namelist.

Namelist Input

The namelist READ statement (figure 5-57) reads input data from a designated file. When a READ statement references a namelist group name, input data in the format described below is read from the designated file. If the specified group name is not found before end-of-file, a fatal error occurs. If the file is empty an end-of-file condition results. This must be detected by an END= or IOSTAT= parameter or a fatal error will result. A subsequent read on the same file without an intervening positioning, CLOSE/OPEN, or EOF function test, results in a fatal error. The format of a namelist input group is shown in figure 5-58 and an example is shown in figure 5-59.

In each record of a namelist group, column one is reserved for carriage control and must be left blank. Data items following \$name (or &name) are read until another \$ (or &) is encountered.


```

READ ( { u,name
        u,FMT=name
        UNIT=u,FMT=name } [,IOSTAT=ios][,ERR=sl][,END=sl] )
READ name

```

Figure 5-57. NAMELIST READ Statement

```

{ $
 & } name { v=c
            v(i1:i2)=c
            array[(s)]=[r*]c[, [r*]c] . . .
            array (s)(i1:i2)=[r*]c[, [r*]c] } [ , { $
 & } { v=c
            v(i1:i2)=c
            array[(s)]=[r*]c[, [r*]c] . . .
            array (s)(i1:i2)=[r*]c[, [r*]c] } . . . [, ] { $
 & } [END]

```

name Is the name of the namelist group.

v Is a variable name.

c Is a constant.

array Is an array name.

s Is an array subscript in which each subscript expression is an integer constant. The number of subscript expressions in s must be equal to the number of dimensions of the array.

r Is an unsigned, nonzero, integer repetition factor.

i1, i2 Are integer constants.

The form r*c is equivalent to r successive appearances of the constant c.

Figure 5-58. NAMELIST Group Format

```

Δ$GRP          Group name
ΔXVAL=5.0,
ΔARR=5*(1.7,-2.4),  Five complex numbers
ΔCHAR='HI THERE',
Δ$END          Group terminator

```

Figure 5-59. NAMELIST Group Input Example

Blanks must not appear in the following locations:

Between \$ (or &) and namelist group name

Between \$ (or &) and END

Within array names and variable names

Blanks can be used freely elsewhere.

Complex constants can be broken across records between the real part and the comma, and between the comma and the imaginary part.

More than one record can be used as input data in a namelist group. The first column of each input record is ignored. All input records containing data should end with a constant followed by a comma; however, the last record can be terminated by a \$ (or &) without the final comma. Constants can be preceded by a repetition factor followed by an asterisk. Omitting a constant constitutes a fatal error.

Constants can be integer, real, double precision, complex, logical, Boolean, or character. Each constant must agree with the type of the corresponding input list item as follows:

A logical, character, or complex constant must be of the same type as the corresponding input list item. A character constant is truncated from the right, or extended on the right with blanks, if necessary, to yield a constant of the same length as the variable, array element, or substring.

An integer, real, or double precision constant can be used for an integer, real, double precision, or Boolean input list item. The constant is converted to the type of the list item. A Boolean constant cannot be used for a non-Boolean list item.

Logical constants have the following forms:

```

.TRUE.      .FALSE.
.T.         .F.
T           F

```

A character constant must have delimiting apostrophes. If a character constant occupies more than one record, each continuation of the constant must begin in column two; a complex constant has the form (real constant, real constant). A character constant must extend to the end of a record preceding a continuation record. A Boolean constant must be an octal constant, a hexadecimal constant, or a Hollerith constant delimited by quotes.

Blank characters appearing within noncharacter constants are ignored. The BLANK= parameter in an OPEN statement has no effect on namelist. If a constant, other than a character constant, contains no characters other than blanks, a fatal error results.

Namelist Output

The namelist output statements are a WRITE (figure 5-60), a PRINT (figure 5-61), and a PUNCH (figure 5-62) statement.

```
WRITE ( { u,name
         u,FMT=name
         UNIT=u,FMT=name } [,IOSTAT=ios][,ERR=sl])
```

Figure 5-60. NAMELIST WRITE Statement

```
PRINT name
```

Figure 5-61. NAMELIST PRINT Statement

```
PUNCH name
```

Figure 5-62. NAMELIST PUNCH Statement

All variables and arrays and their values in the list associated with the namelist group name are output on the file associated with unit u, OUTPUT, or PUNCH. They are output in the order of specification in the NAMELIST statement. Output consists of at least three records. The first record is a \$ in column 2 followed by the group name; the last record is a \$ in column 2 followed by the characters END. Each group begins with triple spacing (a - is inserted in the carriage control position of each record).

See figure 5-63 for an example of namelist output.

```
PROGRAM NAME
NAMELIST /VALUES/ TOTAL,QUANT,COST
DATA QUANT,COST /15.,3.02/
TOTAL = QUANT*COST*1.3
WRITE (6,VALUES)
STOP
END
```

Output:

```
Δ$VALUES
ΔTOTAL   = .5888999999999999E+02,
ΔQUANT   = .15E+02,
ΔCOST    = .302E+01,
Δ$END
```

Figure 5-63. NAMELIST Output Example

No data appears in column 1 of any record. If a noncharacter constant would cross column 80, the columns up to and including 80 are filled with

blanks instead and the constant begins in column 82; therefore, card boundaries will not be crossed if data is punched. The maximum length of any record is 136 characters (unless a smaller maximum record length has been specified in the PROGRAM or OPEN statement). Logical constants appear as T or F. Elements of an array are output in the order in which they are stored.

Character constants are written with delimiting apostrophes. Boolean constants are written in the form O'n [n]...", where n is an octal digit; leading zeros are suppressed.

If a character constant crosses a record boundary and the file is punched, a record length of 80 must be specified to correctly read the cards with namelist input.

Records output by a namelist WRITE statement can be read later in the same program by a namelist READ statement specifying the same group name. See figure 5-64 for example. (Use of F-type records is not supported with namelist output.)

```
NAMELIST /ITEMS/ X,Y,Z
```

```
.
```

```
.
```

```
WRITE (6, ITEMS)
```

Output record:

```
Δ$ITEMS
ΔX      = .7342E+03,
ΔY      = .23749E+04,
ΔZ      = .2225E+02,
Δ$END
```

Subsequent READ statement:

```
READ (5,ITEMS)
```

Figure 5-64. NAMELIST WRITE and Subsequent NAMELIST READ Example

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. See figure 5-65 for examples of array use.

BUFFER INPUT/OUTPUT STATEMENTS

NOTE

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

Example 1:

```
INTEGER BAT(10)
NAMELIST /HAT/ BAT,DOT
READ (5,HAT)
```

Input record:

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

The value of DOT becomes 1.05; the array BAT is as follows:

BAT(1)	2
BAT(2)	3
BAT(3)	4
BAT(4)	4
BAT(5)	4
BAT(6)	4
BAT(7)	4
BAT(8)	4
BAT(9)	4
BAT(10)	4

Example 2:

```
DIMENSION GAY(5)
NAMELIST /DAY/ GAY,BAY,RAY
READ (5,DAY)
```

Input record:

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

array element=constant,...,constant

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants need not equal, but must not exceed, the remaining number of elements in the array.

Example 3:

```
DIMENSION ALPHA (6)
NAMELIST /BETA/ ALPHA,DELTA,X,Y
READ (5,BETA)
```

Input record:

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

In memory:

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

Figure 5-65. Arrays in NAMELIST Examples (Sheet 1 of 2)

Example 4:

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

Input record:

```
Δ$HURRY I1=1,L=.TRUE.,I2=2,I3=3.5,Y(3,5)=26,Y(1,1)=11,
Δ12.0E1,13,4*14,Z=(1.,2.),K=16,M=17$
```

Values stored:

```
I1=1          Y(1,2)=14.0
I2=2          Y(2,2)=14.0
I3=3          Y(3,2)=14.0
Y(3,5)=26.0  Y(1,3)=14.0
Y(1,1)=11.0  K=16
Y(2,1)=120.0 M=17
Y(3,1)=13.0  Z=(1.,2.)
              L=.TRUE.
```

The rest of Y is unchanged.

Figure 5-65. Arrays in NAMELIST Examples (Sheet 2 of 2)

Buffer input/output statements (BUFFER IN and BUFFER OUT) allow input/output operations to occur simultaneously with other processing. They differ from formatted and unformatted READ and WRITE statements in the following ways:

A buffer statement initiates data transmission and then returns control to the program so that it can perform other tasks while data transmission is in progress. A READ or WRITE completes data transmission before returning control to the program.

In a buffer statement, parity must be specified by a parity indicator. In a READ or WRITE statement, parity is implied by the form of the statement: an unformatted READ or WRITE implies binary mode, and a formatted READ or WRITE implies coded mode.

READ and WRITE are associated with an input/output list. Buffer statements are not associated with a list; data is transmitted to or from a block of storage.

ENDFILE, REWIND, and BACKSPACE are valid for files processed by buffer statements. However, a file processed by buffer statements cannot be processed in the same program by formatted or unformatted input/output statements, or by mass storage or CYBER Record Manager subroutines unless the file has been rewound before changing the type of input/output used.

Each buffer statement defines the location of the first and last words of the block of memory to or from which data is to be transmitted. The address of the last word must be greater than or equal to the address of the first word. The relative locations of the first and last word are defined only if they are the same variable or are in the same array, common block, or equivalence class. If the first and last words do not satisfy one of these relationships, their relative position is undefined and a fatal error might result at execution time.

If the first word and the last word are in the same common block but not in the same array or equivalence class, optimization might be degraded.

After execution of a buffer statement has been initiated, and before referencing the same file or any of the contents of the block of memory to or from which data is transferred, the status of the buffer operation must be checked by a reference to the UNIT function. This status check ensures that the data has actually been transferred and the buffer parameters for the file have been restored. If a second buffer operation is attempted on the same file without an intervening reference to UNIT, the results are undefined.

On a CYBER 170 Model 176 or a CYBER 170 Model 800 Series, a FILE control statement (appendix F) specifying SBF=NO must be provided if a level 2 or 3 extended memory variable is used in a buffer statement.

BUFFER IN

The BUFFER IN statement (figure 5-66 and example in figure 5-67) transfers one record from the file indicated by the unit designator u to the block of memory beginning at a and ending at b. If the record is shorter than the block of memory, excess locations are not changed. If the record is longer than the block of memory, excess words in the record are ignored, except when the record type is fixed (RT=F on FILE statement), in which case an error occurs.

The UNIT function can be used to test for end-of-file after BUFFER IN. After UNIT has been referenced, the number of words transferred to memory can be obtained by a call to the function LENGTH. If records do not terminate on a word boundary (in a file not written by BUFFER OUT), the exact length of the record is returned by LENGTHX in terms of words and excess bits.

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

- u** Is a unit specifier.
- p** Is an integer constant or simple integer variable. Designates parity on 7-track magnetic tape; 0 designates even parity (coded); 1 designates odd parity (binary). This parameter is irrelevant for mass storage and 9-track SI tapes. For 9-track S and L tapes, 0 indicates conversion from ASCII or EBCDIC to display code (depending on the REQUEST control statement specification), while 1 indicates no conversion. The parameter does not affect parity on 9-track tapes.
- a** Is the first variable or array element of the block of memory to which data is to be transmitted; cannot be type character.
- b** Is the last variable or array element of the block of memory to which data is to be transmitted; cannot be type character.

If u is a unit designator for a tape or mass storage device, the block of memory to which data is to be transmitted should be one word larger than logically required. The additional word is needed to receive an error status from the operating system if an input/output error occurs. Under SCOPE 2, the additional word is not needed because no error status is written.

Figure 5-66. BUFFER IN Statement

On NOS and NOS/BE, for files with record type S, if the end of a system-logical-record (end-of-section) is encountered on a file other than INPUT, no data is transferred and the length returned by LENGTH is zero. The next BUFFER IN begins reading after the end-of-section. (On file INPUT, end-of-section is treated as end-of-file). End-of-section is transparent to files with other record types.

The UNIT function should be used to test for end-of-file after BUFFER IN. For example,

```
DIMENSION CALC(51)
BUFFER IN (1,0) (CALC(1),CALC(50))
IF(UNIT(1,CALC(1),CALC(50)).GE.0) GO TO 20
```

causes coded information to be transferred from logical unit 1 into storage, beginning at the first word of the array CALC(1) and extending through CALC(50). An error or end-of-file will transfer control to statement 20.

BUFFER OUT

The BUFFER OUT statement (figure 5-68) writes one record by transferring the contents of the block of memory beginning at a and ending at b to the file indicated by the unit designator u at the parity (even or odd) indicated by p. The length of the record:

$$LWA - FWA + 1$$

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

- u** Is a unit specifier.
- p** Is an integer constant or simple integer variable. Designates parity on 7-track magnetic tape; 0 designates even parity (coded); 1 designates odd parity (binary). This parameter is irrelevant for mass storage and 9-track SI tapes. For 9-track S and L tapes, zero indicates conversion from display code to ASCII or EBCDIC (depending on the REQUEST control statement specification), while 1 indicates no conversion. The parameter does not affect parity on 9-track tapes.
- a** Is the first variable or array element of the block of memory from which data is to be transmitted; cannot be type character.
- b** Is the last variable or array element of the block of memory from which data is to be transmitted; cannot be type character.

Figure 5-68. BUFFER OUT Statement

```
PROGRAM TP
INTEGER REC(513),RNUMB
REWIND 1
DO 4 RNUMB = 1,1000
1  BUFFER IN (1,1) (REC(1),REC(513))
2  IF(UNIT(1),REC(1),REC(513))3,5,5
3  K = LENGTH(1)
4  LENGTH RETURNS NUMBER OF WORDS TRANSFERRED BY BUFFER IN.
5  PRINT 100, RNUMB, (REC(I), I = 1,K)
100 FORMAT ('ORECORD', I5, /, (1X,A10))
5  STOP
END
```

Binary information is transferred from logical unit 1 into storage beginning at the first word of the array, REC(1), and extending through REC(513). The UNIT function tests the status of the buffer operation. If the buffer operation is completed without error, statement 3 is executed. If an end-of-file or a parity error is encountered, control transfers to statement 5 and the program stops.

Figure 5-67. BUFFER IN Example

where LWA is the terminal address of the record and FWA is the starting address. For fixed-length records (RT=F on FILE statement), the record length is the length (characters) specified on the FILE statement (FL parameter). If FL is greater than (LWA - FWA + 1) X 10, an error occurs.

The UNIT function must be referenced before another reference is made to the file or to the contents of the block of memory.

INPUT/OUTPUT FILE STATUS STATEMENTS

FORTRAN provides three statements that can be used to establish, examine, or alter certain attributes of files used for input or output. These are the OPEN, INQUIRE, and CLOSE statements. They are discussed separately below.

OPEN

The OPEN statement (figure 5-69) can be used to associate an existing file with a unit number, to create a new file and associate it with a unit number, or to change certain attributes of an existing file.

The UNIT= parameter is required; all other parameters are optional, except for the RECL parameter, which must be specified if a file is being opened for direct access. If a STATUS of OLD or NEW is specified, the FILE= parameter must be given. If a STATUS of SCRATCH is specified, the FILE= parameter must be omitted, and if a STATUS of UNKNOWN is specified, the FILE= parameter is optional.

If the FILE= parameter is omitted, the file is assumed to be the one associated with the specified

```
OPEN ([UNIT=]u[, IOSTAT=ios][, ERR=s][, FILE=fin][, STATUS=sta][, ACCESS=acc][, FORM=fm][, RECL=r][
[, BLANK=blnk][, BUFL=b[ ]])
```

- u Specifies the unit number of the file to be opened. (See File Usage.)
- ios Is an integer variable that contains an error number if an error occurs during the open, or zero if no errors occur.
- sl Is the label of an executable statement to which control transfers if an error occurs during the open.
- fin Is a character expression (seven characters or fewer; first character must be a letter) whose value is the name of the file to be opened. Trailing blanks are removed. This file becomes associated with unit u.
- sta Is a character expression specifying file status. Valid values are:
 - 'OLD' File fin currently exists.
 - 'NEW' File fin does not currently exist.
 - 'SCRATCH' Delete the file associated with unit u on program termination or execution of CLOSE that specifies unit u; must not appear if FILE parameter is specified.
 - 'UNKNOWN' File status is unknown.
 Default is STATUS='UNKNOWN'.
- acc Is a character expression specifying the access method of the file. Valid values are:
 - 'SEQUENTIAL' File is to be opened for sequential access.
 - 'DIRECT' File is to be opened for direct access.
 Default is ACCESS='SEQUENTIAL'.

If the file exists, the access method must be valid for the existing file.
- fm Is a character expression having one of the following values:
 - 'FORMATTED' File is being opened for formatted input/output.
 - 'UNFORMATTED' File is being opened for unformatted input/output.
 - 'BUFFERED' File is being opened for buffered input/output.
 Default is FORM='FORMATTED' for sequential access files, FORM='UNFORMATTED' for direct access files.

For an existing file, the specified form must be valid for that file.

Figure 5-69. OPEN Statement (Sheet 1 of 2)

rl Is an integer variable or positive integer constant specifying the record length for a direct or sequential access file. RECL is required for a direct access file; if omitted for a sequential access file, it defaults to 150 characters or 15 words. Record length is specified in characters for a formatted file and in words for an unformatted file.

blnk Is a character expression having one of the following values:

'NULL' Blank values in numeric formatted input fields are ignored, except that a field of all blanks is treated as zeros.

'ZERO' Blanks, other than leading blanks, are treated as zeros.

Default is BLANK='NULL'.

bl Is a positive (nonzero) integer or Boolean expression specifying the file buffer length in decimal words. bl must be at least 129 words on coded tape, 513 words on binary tape, and 64 words on disk. Default is system selected based on device type.

Figure 5-69. OPEN Statement (Sheet 2 of 2)

unit in the PROGRAM statement (described in section 6). If the file is not specified on the PROGRAM statement, the file name is derived from the unit number. For unit numbers in the range 0 through 999, the file name is TAPE n where n is the unit number; for unit numbers having the form of a logical file name, the file name will be the same as the unit number.

A declaration in an OPEN statement overrides a declaration in a preceding PROGRAM statement provided no input/output operations have been performed on the file. See figure 5-70 for example.

```
PROGRAM XX (TAPE2=500)
.
.
.
OPEN (2,BUFL=1000,FILE='FILEY')
READ (2,100) A,B,C
```

The PROGRAM statement declares a 500 word buffer for unit 2. The OPEN statement specifies a 1000 word buffer for unit 2, which is used. The READ statement reads data from FILEY.

Figure 5-70. OPEN Overriding PROGRAM Declaration Example

Declarations of file properties specified on a FILE control statement override any conflicting OPEN statement parameters for a unit associated with that file; this applies to all OPEN statements for that unit. For example, an MRL or FL specification on a FILE control statement always overrides the RECL parameter value specified in an OPEN statement. (Under SCOPE 2, this is valid only if the file was opened in a previous job step following the FILE control statement.) However, the user should not use the FILE and OPEN statements together to implement two levels of file equivalencing. For example, FILE, OUTPUT, LFN=A combined with OPEN (6,FILE='OUTPUT') might produce unexpected results if the user attempts both a WRITE to tape 6 and a PRINT to OUTPUT.

Once properties of a file have been established in an OPEN statement, only the BLANK= parameter can be changed in a subsequent OPEN statement for that file, unless the file is first closed in a CLOSE statement.

Once a file has been associated with a particular unit, the file can be associated with another unit in a subsequent OPEN statement. The file is then associated with more than one unit. In this case the unit numbers refer to the same file. Actions taken on one unit also affect the other unit. For example, closing a unit closes all other units associated with the same file.

See figure 5-71 for examples of the OPEN statement.

Example 1:

```
OPEN (2,FILE='INFIL')
.
.
OPEN (3,FILE='INFIL')
READ (2,100) A,B
READ (3,100) X,Y
```

Both READ statements read from file INFIL.

Example 2:

```
OPEN (3,FILE='XXX',STATUS='OLD',BLANK='ZERO')
```

When data is read from the existing file XXX, blanks will be interpreted as zeros.

Example 3:

```
OPEN (2,STATUS='NEW',ERR=12,FILE='NEWFL',
+ACCESS='SEQUENTIAL')
```

A new file NEWFL is associated with unit 2 and is to be a sequential access file.

Example 4:

```
OPEN (2,FILE='MYFILE')
WRITE (2,'(A)')A,B,C
OPEN (2,FILE='PART2')
```

The second OPEN statement closes MYFILE before opening PART2.

Figure 5-71. OPEN Statement Examples

If a file is associated with a unit and a succeeding OPEN statement associates a different file with the same unit, the effect is the same as performing a CLOSE without a STATUS= parameter on the currently associated file before associating the new file with the unit. See figure 5-71 example 4.

CLOSE

The CLOSE statement (figure 5-72) disconnects a file from a specified unit and specifies whether the file connected to that unit is to be kept or released.

CLOSE ([UNIT=]u[,Iostat=ios] [,ERR=sl] [,STATUS=sta])	
u	Is the unit designator of the file to be closed.
ios	Is an integer variable which, upon completion of the CLOSE, contains the error number; a value of 0 indicates no errors occurred.
sl	Is the label of an executable statement to which control transfers if an error occurs during the close.
sta	Is a character expression that determines the disposition of the file associated with the specified unit. Valid values are:
'KEEP'	The file is kept after execution of the CLOSE statement.
'DELETE'	The file is unloaded after execution of the CLOSE statement.
Default is STATUS='DELETE' if file status is 'SCRATCH'; otherwise, the default is STATUS='KEEP'.	
'KEEP' is not valid for a file whose status is 'SCRATCH'.	

Figure 5-72. CLOSE Statement

A CLOSE statement can appear in any program unit in the program; it need not appear in the same program unit as the OPEN statement specifying the same unit.

A CLOSE statement that references a unit that does not have a file connected to it has no effect.

After a unit has been disconnected by a CLOSE statement, it can be connected again within the same program to the same file or to a different file. A file connected to a unit specified in a CLOSE statement can be connected again to the same or to another unit, provided the file still exists.

File equivalence established on the PROGRAM statement or on the execution control statement is no longer in effect after the CLOSE statement is executed.

When a program terminates normally, an implicit CLOSE(u,STATUS='KEEP') occurs for each connected unit unless the status of the file was SCRATCH; in this case, a CLOSE(u,STATUS='DELETE') occurs.

Example:

```
CLOSE (2,ERR=25,STATUS='DELETE')
```

In static mode, once a file is closed in a CLOSE statement, it must be explicitly reopened if it is to be used again.

INQUIRE

There are two forms of the INQUIRE statement (figure 5-73 and example in figure 5-74). Inquire by unit is used to obtain information about the current status of a specified unit. Inquire by file is used to obtain information about the current status of a file. Either a file name (inquire by file) or a unit specifier (inquire by unit), but not both, must be specified in an INQUIRE statement. The file or unit need not exist when INQUIRE is executed.

Following execution of an INQUIRE statement, the specified parameters contain values that are current at the time the statement is executed.

If a unit number is specified and the unit is opened, the NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, OPENED, EXIST, NUMBER, ACCESS, and BLANK variables will contain information about the file associated with the unit. EXIST returns a TRUE value only if the unit has been opened by a reference on the PROGRAM statement or the OPEN statement; it does not indicate whether a file by this name is local or not.

If a file name is specified, the NAMED, NAME, SEQUENTIAL, DIRECT, FORMATTED, UNFORMATTED, OPENED, EXIST, NUMBER, ACCESS, FORM, RECL, NEXTREC, and BLANK variables will contain information about the file and the unit it is associated with. EXIST returns a TRUE value only if a nonempty local file by this name exists or if an empty local file by this name is currently open.

When EXIST returns a FALSE value, the NUMBER, NAMED, NAME, ACCESS, SEQUENTIAL, DIRECT, FORM, FORMATTED, UNFORMATTED, RECL, NEXTREC, and BLANK variables will contain undefined values. This does not result in an error.

If a file is specified that is associated with more than one unit, the NUMBER variable will contain one of the unit numbers or names.

Note that if a unit that is not associated with a file is specified, only the IOSTAT and EXIST variables contain values.

If an error occurs during an INQUIRE, only IOSTAT contains a value.

MEMORY-TO-MEMORY INPUT/OUTPUT STATEMENTS

Internal files provide a means of reformatting and transferring data from one area of memory to another, without the need to write the data and reread it under a different format specification. Internal files also allow numeric conversion to or

from character data type. Input and output on internal files are performed by formatted READ and WRITE statements and the ENCODE and DECODE statements. However, no input/output devices are involved.

The two types of internal files are standard internal files and extended internal files. They are discussed below along with the ENCODE and DECODE statements.

```
INQUIRE ( { [UNIT=u]
            { FILE=fin }
            [, IOSTAT=ios] [, ERR=sl] [, EXIST=ex] [, OPENED=od] [, NUMBER=num] [, NAMED=nmd] [, NAME=fn]
            [, ACCESS=acc] [, SEQUENTIAL=seq] [, DIRECT=dir] [, FORM=fm] [, FORMATTED=fmt] [, UNFORMATTED=unf]
            [, RECL=rcl] [, NEXTREC=nr] [, BLANK=blnk] )
```

- u** Is the external unit for which information is to be returned; if the unit is associated with a file, information about the file is returned. (The format of this parameter is described under File Usage.)
- fin** Is a character expression specifying the name of the file for which information is to be returned.
- ios** Is an integer variable which, upon completion of the INQUIRE, contains an error number; the value is 0 if no errors occurred.
- sl** Is a user-specified statement label of an executable statement to which control passes if an error occurs during an inquire.
- ex** Is a logical variable:
 - .TRUE. The file (unit) exists.
 - .FALSE. The file (unit) does not exist.
- od** Is a logical variable:
 - .TRUE. The file (unit) is connected to a unit (file).
 - .FALSE. The file (unit) is not connected to a unit (file).
- num** Is an integer variable containing the external unit number of the unit currently associated with the file; undefined if the file is not associated with a unit.
- nmd** Is a logical variable:
 - .TRUE. The file has a name.
 - .FALSE. The file does not have a name.
- fn** Is a character variable containing the name of the file associated with unit u.
- acc** Is a character variable indicating the access method of the file:
 - 'SEQUENTIAL' The file is opened for sequential access input/output.
 - 'DIRECT' The file is opened for direct access input/output.

If the file is not opened, acc is undefined.
- seq** Is a character variable indicating whether the file can be opened for sequential access input/output:
 - 'YES' The file can be opened for sequential access input/output.
 - 'NO' The file cannot be opened for sequential access input/output.
 - 'UNKNOWN' Cannot be determined.
- dir** Is a character variable indicating whether the file can be opened for direct access input/output:
 - 'YES' The file can be opened for direct access input/output.
 - 'NO' The file cannot be opened for direct access input/output.
 - 'UNKNOWN' Cannot be determined.

Figure 5-73. INQUIRE Statement (Sheet 1 of 2)

fm	Is a character variable indicating formatted or unformatted input/output: 'FORMATTED' The file is opened for formatted input/output. 'UNFORMATTED' The file is opened for unformatted input/output. If the file has not been opened, fm is undefined.
fmt	Is a character variable specifying whether the file can be opened for formatted input/output: 'YES' The file can be opened for formatted input/output. 'NO' The file cannot be opened for formatted input/output. 'UNKNOWN' It cannot be determined if the file can be opened for formatted input/output.
unf	Is a character variable specifying whether the file can be opened for unformatted input/output: 'YES' The file can be opened for unformatted input/output. 'NO' The file cannot be opened for unformatted input/output. 'UNKNOWN' It cannot be determined if the file can be opened for unformatted input/output.
rcl	Is an integer variable containing the record length of a file opened for direct access. If the file is 'FORMATTED', rcl contains the record length in characters; if 'UNFORMATTED', the record length is in words; undefined if the file is not opened for direct access.
nr	Is an integer variable; for a direct access file, nr contains the record number of the last record read or written plus one. If no records have been read or written, nr contains 1. Undefined for sequential files.
blnk	Is a character variable: 'NULL' Null blank control is in effect for a file opened for formatted input/output. 'ZERO' Zero blank control is in effect for a file opened for formatted input/output. Undefined if the file is not opened for formatted input/output.

Figure 5-73. INQUIRE Statement (Sheet 2 of 2)

```

LOGICAL EX
CHARACTER*10 AC
.
.
.
INQUIRE (FILE='AFILE', ERR=100, EXIST=EX,
+ACCESS=AC)

```

Figure 5-74. INQUIRE Statement Example

Standard Internal Files

A standard internal file can be any character variable, array, or substring. If the file is a variable or substring, it consists of a single record whose length is the length of the variable or substring. If the file is an array, each array element constitutes a single record. For example,

```
CHARACTER *20 A(100)
```

creates an internal file A containing 100 records of 20 characters each.

LCM resident internal files are restricted to a maximum length of 150 characters when used in READ or WRITE statements.

Records of an internal file are defined by storing data into the records, either with an output statement or an assignment statement. The record is blank filled before the characters are moved in. So, if the number of characters written in a record is less than the length of the record, the unused portion contains blanks.

It is not necessary to declare internal files in the same manner as external files. Only formatted input/output can be used; unformatted, list directed, **namelist**, and **buffer input/output** are not valid for internal files. In addition, file manipulation and file status statements cannot be used with internal files. Some sample programs using internal files are included in section 12.

For input, data is read from a standard internal file using a formatted READ statement in which the internal unit identifier is a character variable, array, or substring. Data is transferred from consecutive locations starting at the first character position of u, converted under format specification, and stored in the variables specified in iolist.

Some examples of internal files used for input are shown in figure 5-75.

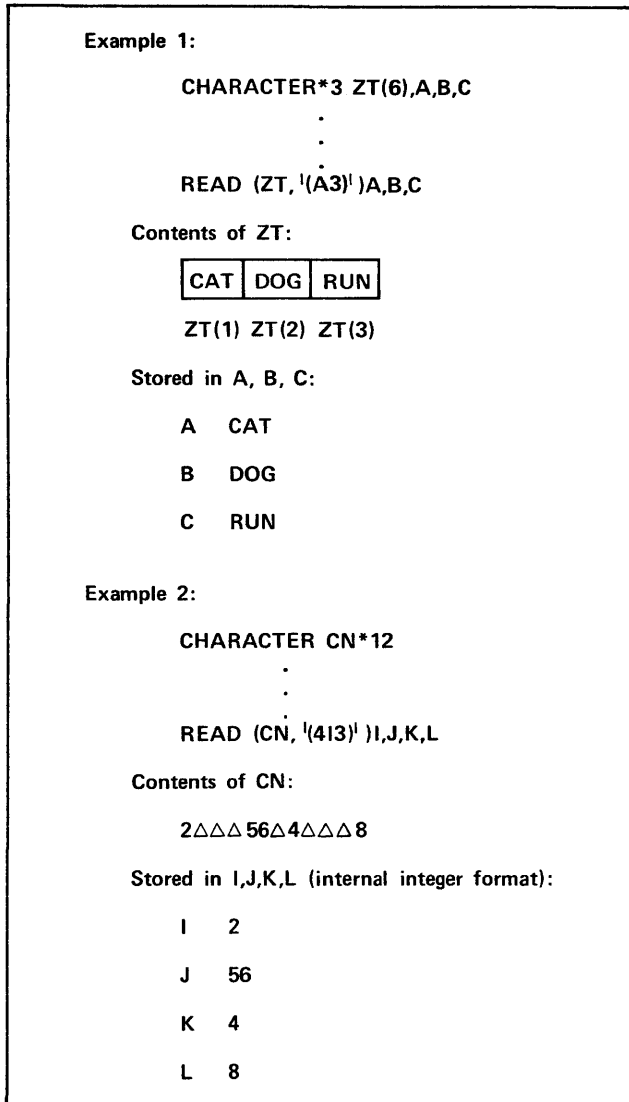


Figure 5-75. Internal File Input Examples

For output, data is written to standard internal files using a formatted WRITE statement in which the internal unit u is a character variable, array, or substring name. The WRITE statement transmits data from the variables specified in iolist to consecutive locations starting with the leftmost character of the location specified by u; data is converted from internal to character format according to the format specification. The number of characters transmitted is determined by the record length. If the iolist exceeds the number of edit descriptors, the format statement is processed again and variables are written to the next record in the internal file. (See the discussion under Repetition Factors.) If there are fewer records in the internal file than iolist variables, the output is terminated.

Figure 5-76 shows some examples of internal files used for output.

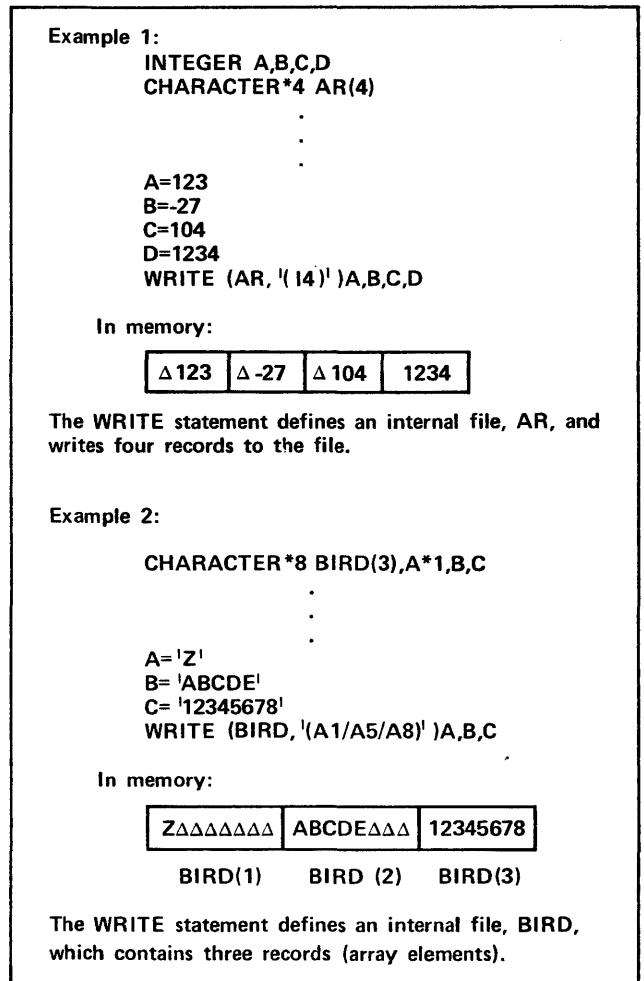


Figure 5-76. Internal File Output Examples

Extended Internal Files

NOTE

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

An extended internal file can be any noncharacter variable, array, or array element. A record of an extended internal file is defined by writing the record. The record length is measured in characters. Since one word contains 10 characters, the record length of an extended internal file is given by:

$$10*a$$

where a is the number of words in the record.

An extended internal file residing in LCM cannot consist of more than 15 words.

ENCODE Statement

The ENCODE statement (figure 5-77 and examples in figure 5-78) is the extended internal file output statement.

ENCODE (c, fn, u) iolist

- c** Is an unsigned integer constant or variable having a value greater than zero; c specifies the number of characters to be transferred per record. The record length is calculated from c. Must not exceed 150 if the file resides in level 2 storage on a CYBER 170/Model 176, CYBER 70/Model 76, or 7600 computer.
- fn** Is a statement label of a FORMAT statement, or a character expression whose value is a format specification; fn must not specify NAMELIST or list directed formatting.
- u** Is an extended internal file (noncharacter variable, array element or array name) in which the record is to be encoded.
- iolist** Is a list of noncharacter variables, arrays or array elements to be transmitted to the location specified by u.

Figure 5-77. ENCODE Statement

ENCODE is similar to an internal file formatted WRITE. Values are transferred to the receiving storage area from the variables specified in iolist under the specified format. The first record starts with the leftmost character of the location specified by u. The length in characters of each record is:

$$\text{INT}((c+9)/10)*10$$

where INT(a) is the largest integer less than or equal to a. If c is less than the record length, the remainder of the word is blank filled.

The internal file must be large enough to contain the total number of characters transmitted by the ENCODE statement. For example, if 70 characters are generated by the ENCODE statement, the array starting at location v must be at least 70 characters (7 words) in length. If A is the receiving array the declaration `BOOLEAN A(7)` would be sufficient. If 27 characters are generated, the declaration `BOOLEAN A(3)` is sufficient.

If the list and the format specification transmit more than the number of characters specified per record, an execution error message is printed. If the number of characters transmitted is less than the record length, remaining characters in the record are blank filled.

ENCODE can be used to calculate a field definition in a format specification at execution time.

An area in memory should not be encoded or decoded upon itself, as this gives unpredictable results.

Example 1:

```
BOOLEAN SPECMAT(1)
.
.
IF(M.GE.10.OR.M.LE.1) GO TO 2
ENCODE (10,100,SPECMAT)M
100 FORMAT ('(2A10,I',I1,1H)')
.
.
PRINT SPECMAT,A,B,J
```

In this example, the value m in the FORMAT statement (2A10,Im) is to be specified at some point in the program. The variable M is permitted to vary in the range 2 through 9. M is tested to ensure it is within limits; if it is not, control goes to statement 2 which could be an error routine. If M is within limits, ENCODE packs the integer value of M with the characters (2A10,IA) in SPECMAT. A and B will be printed under specification A10, and the quantity J under specifications I2 through I9, according to the value of m.

Example 2:

```
PROGRAM IGEN
INTEGER FMT(1)
DO 9 J=1,50
ENCODE (10,7,FMT)J
7 FORMAT ('(I',I2,')')
9 WRITE (6,FMT)J
STOP
END
```

In memory, FMT is first (I1), then (I2), then (I3), and so forth.

Figure 5-78. ENCODE Statement Examples

DECODE Statement

The DECODE statement (figure 5-79 and examples in figure 5-80) is the extended internal file input statement.

DECODE (c, fn, u) iolist

- c, fn, and u are as described for ENCODE.
- iolist** Is a list of noncharacter variables, arrays, or array elements to receive data from the extended internal file specified by u.

Figure 5-79. DECODE Statement

DECODE performs a memory-to-memory transfer of data similar to an internal file formatted READ. Starting at location u, display code characters in memory are converted according to the specified format and stored in the variables specified in iolist.

Example 1:

In memory:

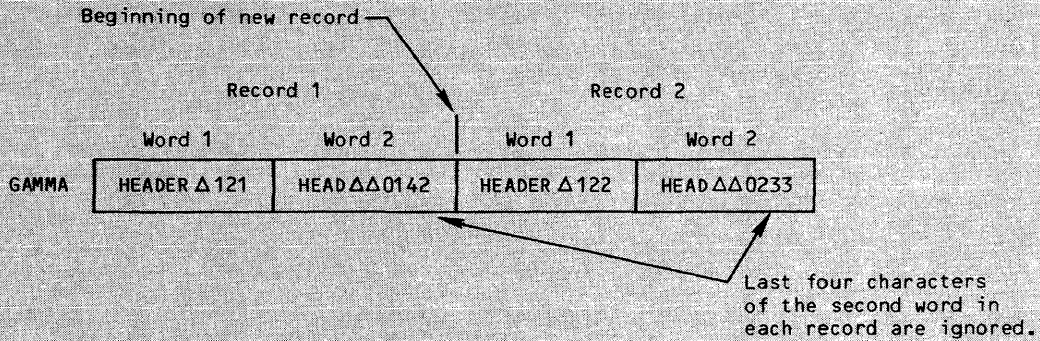
```
LOC1  SSSSxxxxx
LOC2  xxxxxDDDDD
```

The following statements store SSSSDDDDD in location NAME:

```
BOOLEAN LOC1,LOC2,TEMP,NAME
.
.
.
DECODE (10,1,LOC2)TEMP
1 FORMAT (5X,A5)
ENCODE (10,2,NAME)LOC1,TEMP
2 FORMAT (2A5)
```

Example 2:

```
INTEGER GAMMA (4)
DECODE (16,1,GAMMA) X,B,C,D
1 FORMAT (2A8)
```



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

```
X=HEADER Δ1
B=21HEADΔΔ
C=HEADER Δ1
D=22HEADΔΔ
```

Figure 5-80. DECODE Statement Examples

DECODE processing of an illegal character for a given conversion specification produces a fatal error. If DECODE is processing an A or R format specification and encounters a zero character (6 bits of binary zero), the character is treated as a colon under 64-character set or as a blank under 63-character set.

DECODE can be used to pack the partial contents of two words into one.

FILE POSITIONING STATEMENTS

Three statements can be used to position files connected for sequential access: REWIND, BACKSPACE, and ENDFILE. They are discussed separately below.

REWIND

The REWIND statement (figure 5-81) positions a file at beginning-of-information so that the next input/output operation references the first record in the file, even though several ENDFILE statements may have been issued to that unit since the last REWIND. If the file is already at beginning-of-information, no action is taken. (Refer to BACKSPACE/REWIND, appendix F, for further information.)

```
REWIND ([UNIT]=]u[,IOSTAT=ios][,ERR=s[]])
REWIND u
```

Figure 5-81. REWIND Statement

Example:

```
REWIND 3
```

BACKSPACE

The BACKSPACE statement (figure 5-82) backspaces unit *u* one record. When the file is positioned at beginning-of-information, this statement acts as a do-nothing statement. Backspace operations should not be used on direct access files or on records created by list directed or namelist output.

```
BACKSPACE ([UNIT=u][,IOSTAT=ios][,ERR=s])  
BACKSPACE u
```

Figure 5-82. BACKSPACE Statement

Example:

```
DO 1 LUN = 1,4  
1 BACKSPACE LUN
```

The files associated with units 1 through 4 are backspaced one record.

ENDFILE

The ENDFILE statement (figure 5-83) writes an end-of-partition (end-of-file) on the designated unit. ENDFILE is not permitted on units opened for direct access. The end-of-partition boundary can be detected by the END= and IOSTAT= variables.

```
ENDFILE ([UNIT=u][,IOSTAT=ios][,ERR=s])  
ENDFILE u
```

Figure 5-83. ENDFILE Statement

Because the file mode (formatted or unformatted) cannot be detected, ENDFILE should not be the first operation on a file.

Meaningful results are not guaranteed if ENDFILE is used on a file processed by mass storage sub-routines.

Example:

```
IOUT = 7  
ENDFILE (UNIT=IOUT, ERR=100)
```

End-of-partition is written on unit 7.

An executable program consists of one main program and optional subprograms.

Both main programs and subprograms are known as program units. A program unit contains a group of FORTRAN statements, including optional comments; it is terminated by an END statement. Program units can be compiled independently of each other, but a subprogram cannot be executed except through a main program.

There are two types of subprograms: the specification subprogram and the procedure subprogram.

A subprogram that begins with a BLOCK DATA statement is a specification subprogram. It is used to enter initial values for variables and array elements in named common blocks.

A subprogram that begins with a SUBROUTINE statement or a FUNCTION statement is a procedure subprogram known as a subroutine subprogram or a function subprogram, respectively. It can accept one or more values through a list of arguments, common blocks, or both.

A procedure is a function or a subroutine subprogram that can be executed many times. A subroutine subprogram begins with a SUBROUTINE statement and terminates with an END statement; it can return one or more values to the referencing program unit.

A function is used only in expressions to supply a value to the expression. Functions can occur in three forms:

As a user-written function subprogram beginning with a FUNCTION statement, terminating with an END statement, and containing other statements.

As a single statement written by the user.

As an intrinsic function supplied by the FORTRAN library. Intrinsic functions are discussed in section 7 along with other procedures supplied by the FORTRAN library.

These functions differ in the way they are written and referenced, although they have many characteristics in common.

A function written by the user is sometimes known as an external function.

Table 6-1 summarizes the characteristics of these overlapping entities.

Functions written in FORTRAN that have names identical to the mathematical intrinsic function names described in section 7, such as AMAX1 or SQRT, must be declared EXTERNAL in the calling program unit. This declaration causes the compiler to reference the user-defined function, using a pass by reference sequence.

MAIN PROGRAM

A main program is a program unit that does not begin with a FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY statement. The main program should have a PROGRAM statement (optional) and at least one executable statement followed by an END statement.

The execution of any program begins with the main program unit. No executable program can have more than one main program unit, except that an overlay-structured program has one main program unit in each overlay.

The main program can be compiled independently of any subprograms. However, when a main program is loaded into memory for execution, all the required subprograms must be loaded with it prior to its execution.

Although the PROGRAM statement is optional, it is useful in defining the program name and declaring certain properties of files to be used by the program. The format and use of the PROGRAM statement are discussed below.

PROGRAM STATEMENT

The PROGRAM statement (figure 6-1 and example in figure 6-2) defines the program name that is used as the entry point name and as the object deck name for the loader. The PROGRAM statement also declares certain properties of files to be used by the program.

In a program structured for overlays, the fpar list is used only in the PROGRAM statement for the main overlay. The fpar list cannot be used for the PROGRAM statements of primary or secondary overlays.

PROGRAM STATEMENT USAGE

The PROGRAM statement can declare files that are used in the program and in any subprograms that are called. If this statement is omitted from the main program, the program is assumed to have the 6-character name START, and two files named INPUT and OUTPUT.

Files referenced in input/output statements need not be declared in a PROGRAM statement. If a file is not declared on the PROGRAM statement, a buffer will be created on the first reference to the file.

FORTRAN input/output routines add the characters TAPE as a prefix to the unit number to form the file name. For instance, TAPE3 is the file name assigned to unit number 3 and TAPE5 is the file name assigned to unit number 5. TAPE5 and TAPE05 do not specify the same file name. TAPEu refers to a file located on rotating mass storage unless specified otherwise in the job control statements

TABLE 6-1. CHARACTERISTICS OF PROGRAM COMPONENTS

Characteristics	Main Program	Block Data	Subroutine	(External) Function	Statement Function	Intrinsic Function
Program unit?	Yes	Yes	Yes	Yes	No	†
Subprogram?	---	---	Yes	Yes	No	†
Procedure?	---	No	Yes	Yes	Yes	Yes
User-written?	Yes	Yes	Yes	Yes	Yes	No
Reference from	---	External statement in any program unit	Any executable program unit	Any executable program unit	Defining program unit	Any executable program unit
Multiple entry possible?	---	---	Yes	Yes	---	---
RETURN statement possible?	Same as END	---	Alternate, multiple	Multiple	No	---
How referenced	Control statement	Not executable	CALL statement	Name used in expression	Name used in expression	Name used in expression
How typed	---	---	---	Implicitly or explicitly	Implicitly or explicitly	Function name or generic
How values accepted	---	---	Through arguments or common blocks	Through arguments or common blocks	Through arguments	Through arguments
How values returned	---	---	Through arguments or common	Function name	Function name	Function name

†An intrinsic function, along with other supplied subprograms, can be external; and in terms of an executable program, is considered to be a program unit. Some intrinsic functions are compiled in-line; therefore, they are neither external nor procedures since the same code is not re-executed.

before the program is executed. The file is temporary unless it is explicitly declared to be permanent through the appropriate operating system control statement.

If a buffer length is specified on the PROGRAM statement, FORTRAN input/output statements will use a buffer with the specified length. The buffer length can appear only with the first reference to the file in the PROGRAM statement. A buffer length of zero should be specified for a file referenced by a BUFFER IN or BUFFER OUT statement, unless the file is a connected file or the file description has been changed by a FILE control statement. Since buffered records are transmitted directly into and out of central memory, the field length of the program is reduced for each file declared with zero buffer length in the PROGRAM statement. The following values of n are minimal if default record and block types are used:

For terminals, n is ignored unless a record, which exceeds the default size of the internal buffer, is written. If a record that exceeds

the default size of the internal buffer is written, and n also exceeds the default size of the internal buffer, CMM provides a block of n words that will be used for subsequent writes to the terminal. For efficiency, set the buffer length to zero.

For mass storage input/output files, n must be at least 64. Large records and sequential reading/writing execute faster with a larger buffer.

For sequential files, the format controls the minimum value of n in the following way:

SI tape

129 for formatted, 513 for unformatted

I,X tape

513 for unformatted (NOS only)

PROGRAM name[(fpar[,fpar]...)]	
name	Is the program name that cannot be used elsewhere in the program as a user-defined name.
fpar	Can declare the file in any of the following forms: file file=n file=/r file=n/r altunit=file
file	Is the file name of a file required by the main program or its subprograms; the maximum number of file names is 49.
file=n	Specifies an integer or octal constant for the buffer length; default length is 2003 octal words. n is ignored if specified in a program run under SCOPE 2.
file=/r	Specifies the maximum length in characters for list directed, formatted, and NAMELIST records; default length is 150 characters.
file=n/r	Specifies both the buffer length and record length for the file.
altunit=file	Specifies that two file names are equivalent. The buffer length and record length for altunit are the same as previously specified (or defaulted) for file. The two files share the same FIT. Usually, altunit has the form TAPEu where u is an integer in the range 0 through 999.

Figure 6-1. PROGRAM Statement

S tape	513 for formatted or unformatted
L tape	must be at least maximum block length plus 1
mass storage	64 for formatted, 513 for unformatted

The appearance of a symbolic file name in the PROGRAM statement has a similar effect as the execution of an OPEN statement (see section 5) with that file name. Any later attempt to use the OPEN statement to change the buffer length results in an error, unless the file has been closed first.

Example 1:	<pre>PROGRAM PROGA(AFILE,TAPE2=AFILE)</pre> <p>Assigns the name PROGA to the program, and equivalences the name TAPE2 to AFILE. When any input/output statement in the program references unit 2, the reference applies to local file AFILE.</p>
Example 2:	<pre>PROGRAM ABC(INDAT=1000/200,TAPE6=INDAT)</pre> <p>Assigns the name ABC to the program. File INDAT is declared to have a buffer length of 1000 words and a maximum record length of 200 characters. Any reference to unit 6 references file INDAT.</p>
Example 3:	<pre>PROGRAM FIRST</pre> <p>Assigns the name FIRST to the program. The file list is legally omitted.</p>

Figure 6-2. Examples of PROGRAM Statement

Record length should always be specified for files referenced in list directed input/output statements. This specification creates a separate working storage area for the file, which is different from the default area. If the default area is used, input/output to other files destroys any data remaining after a list directed read.

When file names are made equivalent, the buffer length and record length specified for the first file also apply to the specified altunit. Therefore, any attempt to specify buffer length or record length for altunit results in an error. An example of equivalent file names is shown in figure 6-3.

PROCEDURES, SUBPROGRAMS, AND STATEMENT FUNCTIONS

Subroutine subprograms and function subprograms are both procedures. They are discussed below, along with the ENTRY, RETURN, and END statements that are applicable to both of these types of procedures.

A statement function is also a procedure, although it is not a subprogram. It is discussed separately below.

SUBROUTINE SUBPROGRAM

A subroutine subprogram is a procedure that communicates with the calling program unit either through a list of arguments passed with the CALL statement or through common blocks. A subroutine subprogram is executed when a CALL statement naming the subroutine is encountered in a program unit.

Subroutines begin with a SUBROUTINE statement and end with an END statement. Control is returned to the calling program unit when a RETURN statement is encountered. If control flows into the END statement, then a RETURN is implied.

```

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

```

The READ statement reads from logical unit 5. Since the PROGRAM statement declares file TAPES as equivalent to file INPUT, records are read from file INPUT. The WRITE statement writes to logical unit 6. Since the PROGRAM statement declares file TAPE6 as equivalent to file OUTPUT, records are written to file OUTPUT. The equivalent OPEN statements to the PROGRAM statement are:

```

OPEN(5,FILE='INPUT')
OPEN(6,FILE='OUTPUT')

```

Figure 6-3. File Equivalencing Example

SUBROUTINE Statement

The SUBROUTINE statement (figure 6-4 and example in figure 6-5) must appear as the first statement of the subroutine subprogram. The statement contains the symbolic name that is the main entry point of the subprogram. The name must not be the same as any other program unit or entry name. Also, the name cannot be the same as any name in the subroutine.

```
SUBROUTINE sub[[d[,d]...]]
```

- sub Is the name of the subroutine subprogram. If there are no dummy arguments, either sub or sub() can be used.
- d Is a dummy argument that can be a variable name, array name, dummy procedure name, or *.

Figure 6-4. Subroutine Statement

The subprogram name is not used to return results to the calling program.

Subroutines can contain any statement except a PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. A subroutine subprogram must not directly or indirectly call itself.

In a subroutine subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a subroutine call. The SUBROUTINE statement can also have dummy arguments for statement labels; these arguments are represented by asterisks.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in subroutine subprograms. More details can be found below.

```

PROGRAM MAIN
INTEGER A,B
OPEN(5,FILE='INPUT')
OPEN(6,FILE='OUTPUT')
100 READ(5,100) A,B
    FORMAT(2I2)
    IF (A-B) 10,20,20
10 CALL ERROR1
20 RESULT = (A*100) + 375.2
    STOP
    END

C
SUBROUTINE ERROR1
WRITE(6,1)
1 FORMAT(5X, 'NUMBER IS OUT OF RANGE')
RETURN
END

```

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

Figure 6-5. Subroutine Call Example

CALL Statement

A subroutine subprogram is brought into execution by a CALL statement (figure 6-6).

CALL sub([[a[,a] . . .]])	
sub	Is the name of subroutine or dummy procedure.
a	Is an actual argument that can be one of the following: An expression (except a character expression involving concatenation of a dummy argument with length (*)) An array name An intrinsic function name An external procedure name A dummy procedure name An alternate return specifier of the form *s
s	Is the statement label of an executable statement that appears in the same program unit as the CALL statement.

Figure 6-6. CALL Statement

The CALL statement can contain actual arguments and statement labels which must correspond in order, number, and type to those in the subroutine definition. An actual argument of type Boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type Boolean.

An actual argument in a CALL statement can be a dummy argument name that appears in the dummy argument list of the subprogram containing the CALL statement. An asterisk dummy argument cannot be used as an actual argument.

FUNCTION SUBPROGRAM

A function subprogram is a procedure that communicates with the referencing program unit through a list of arguments or common blocks. It is usually referred to as an external function. A function subprogram performs a set of calculations when the name appears in an expression in the referencing program unit.

(In addition to the function subprogram, there are functions that are procedures but are not subprograms. These functions are the statement function discussed later in this section and the intrinsic function discussed in the next section.)

A function subprogram begins with a FUNCTION statement and ends with an END statement. Control is returned to the referencing program unit when a RETURN or END is encountered; a RETURN statement of the form RETURN exp is not allowed in a function subprogram.

A function subprogram can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, or another FUNCTION statement. A function must not directly or indirectly reference itself.

FUNCTION Statement

The FUNCTION statement (figure 6-7 and example in figure 6-8) must appear as the first statement of the function subprogram. The FUNCTION statement specifies the symbolic name that is used as the main entry point of the subprogram.

[typ] FUNCTION fun([d[,d] . . .])	
typ	Is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BOOLEAN, or CHARACTER*len. The len value specifies the length of the result of the character function.
fun	Is the name of the function subprogram; fun is an external function name.
d	Is a dummy argument that can be a variable name, array name, or dummy procedure name.

Figure 6-7. FUNCTION Statement

	PROGRAM MAIN
	INTEGER Z
	Z = JOR(5,3)
	ZZ = JAM(5,3)
	PRINT *, Z,ZZ
	STOP
	END
c	
	FUNCTION JOR (X,Y)
	INTEGER X,Y
	JOR = X-Y
	RETURN
	ENTRY JAM(X,Y)
	JAM=X+Y
	RETURN
	END
	Function subprogram JOR is executed when the name appears in the calling program unit. The alternate entry point is entry JAM in function JOR.

Figure 6-8. Function Reference Example

The symbolic name of a function subprogram, or an associated entry name of the same type, is a variable name in the function. The symbolic name specified in a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type statement. If the type of a function is specified in a FUNCTION statement, then the function name cannot appear in a type statement. In an executable statement, the symbolic name can appear only as a variable. During execution, this variable becomes defined and can be referenced or redefined. The value of the function is the value of this variable when control returns to the referencing program unit.

The type of the function name must be the same in the referencing program unit and the referenced function subprogram. When type is omitted, the type of the function is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when the function name is not explicitly typed. The name cannot have its type explicitly specified more than once.

If the name of a function subprogram is of type character, then each entry name must be type character and vice versa. The length of the function subprogram symbolic name and any entry names in the function subprogram must be specified with the same length. For example, if the function name has a length of (*), all entry names must have a length of (*).

The symbolic name of a function subprogram must not be the same as any other name, except a variable name or common block name.

The function subprogram can have more than one entry point, although alternate returns are prohibited. See figure 6-8 for example. Multiple entry points are established through the ENTRY statement.

In a function subprogram, the symbolic name of a dummy argument is unique to the program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA, or COMMON statement, except as a common block name. The dummy arguments are replaced with the actual arguments during a function reference.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. Adjustable dimensions are permitted in function subprograms, as described below.

Function Subprogram Reference

References to function subprograms are not the same as references to statement functions.

Function subprograms and intrinsic functions are referenced through a function name (figure 6-9). When the function name is referenced in an expression, a function is executed. The function name can appear anywhere in an expression where an operand of the same type can be used.

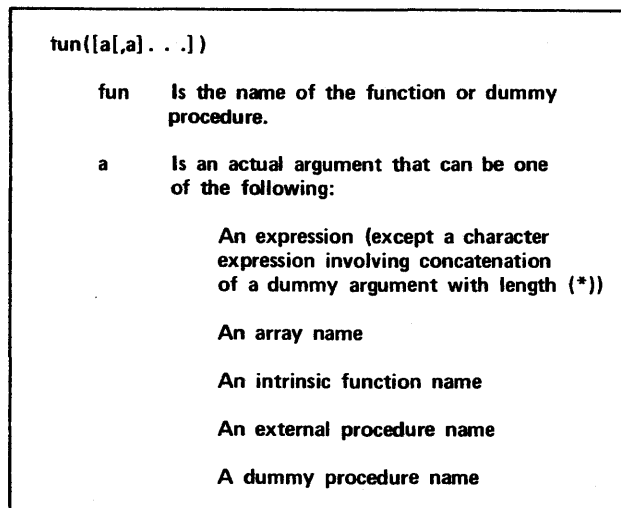


Figure 6-9. Function Reference Name

The type of the function result is the type of the function name. The arguments must agree in order, number, and type with the corresponding dummy arguments. An actual argument of type Boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type Boolean.

Function subprograms and intrinsic functions can be referenced in any procedure subprogram.

ENTRY STATEMENT

The ENTRY statement (figure 6-10 and example in figure 6-11) can be used to define additional entry points for a procedure established by the SUBROUTINE or FUNCTION statement.

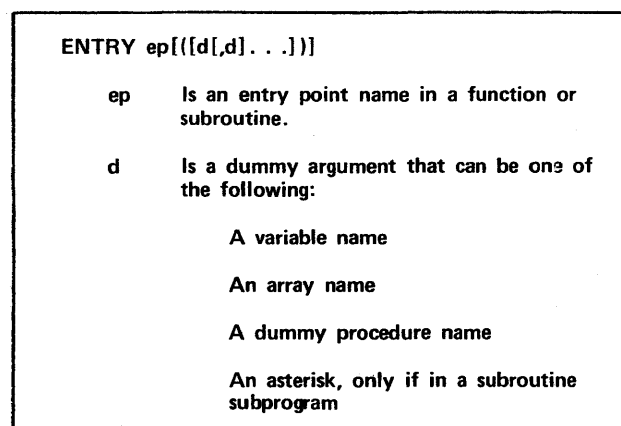


Figure 6-10. ENTRY Statement

```

PROGRAM MAIN
DIMENSION SET1(25)
1 READ 5,N
5 FORMAT(I1)
IF(N .EQ. 0) GO TO 900
IF(N. EQ. 1) CALL CLEAR(SET1)
IF(N. EQ. 2) CALL FILL(SET1)
DO 99 I=1,25
PRINT 6, SET1(I)
6 FORMAT (F5.2)
99 CONTINUE
GO TO 1
900 STOP
END

C
SUBROUTINE CLEAR(ARRAY)
DIMENSION ARRAY(25)
20 DO 10 I= 1,25
ARRAY(I) = 0.0
10 CONTINUE
ENTRY FILL(ARRAY)
READ *, VALUE,IPLACE
IF(IPLACE .GT. 25) RETURN
ARRAY(IPLACE) = VALUE
RETURN
END

```

Figure 6-11. Examples of ENTRY Statements

Each procedure subprogram has a primary entry point established by the statement that begins the program unit. Usually, a subroutine call or function reference invokes the procedure at the primary entry point, and the first statement executed is the first executable statement in the program unit. ENTRY statements can be used to define other entry points.

A procedure that contains one or more ENTRY statements has multiple entry points.

An ENTRY statement can appear anywhere after the SUBROUTINE or FUNCTION statement in the subprogram. An ENTRY statement cannot appear between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminal statement of the DO loop.

When an entry name is used to reference a procedure, execution begins with the first executable statement that follows the referenced entry point. An entry name is available for reference in any program unit, except in the procedure that contains the entry name. The entry name can appear in an EXTERNAL statement and (for a function entry name) in a type statement.

Each reference to a procedure must use an actual argument list that corresponds in number of arguments and type of arguments with the dummy argument list in the corresponding SUBROUTINE, FUNCTION, or ENTRY statement. Type agreement is not required for actual arguments that have no type, such as a dummy subroutine name. The dummy arguments for an entry point can therefore be different from the dummy arguments for the primary entry point or another entry point. No dummy argument can be used in an executable statement of a procedure unless it has already appeared in a FUNCTION, SUBROUTINE, or ENTRY statement.

The order, number, type, and names of the dummy arguments in an ENTRY statement may be different from the order, number, type, and names of the dummy arguments in the FUNCTION statement or SUBROUTINE statement and other ENTRY statements in the same subprogram.

RETURN AND END STATEMENTS

Execution of a END or RETURN statement returns program control to the next executable statement in the referencing program unit. These statements are often used together at the end of a subroutine subprogram or a function subprogram.

The END statement (figure 6-12) ends each subroutine subprogram or function subprogram. Execution of the statement terminates the subroutine or function.

```

END

```

Figure 6-12. END Statement

Single and Multiple Returns

The RETURN statement (figure 6-13) can be used wherever appropriate to terminate the subroutine or function subprogram. The END statement must be at the end of the subprogram.

```

RETURN[exp]

exp Is an arithmetic or Boolean expression. If exp is not of type integer, the value INT(exp) is used. The optional expression exp can only be used in a subroutine subprogram.

```

Figure 6-13. RETURN Statement

Both subroutine subprograms and function subprograms can have multiple returns. A multiple return exists: when the subprogram has more than one RETURN statement, and when a single RETURN statement is separated from the END statement by other statements.

The optional expression exp is not applicable to a single or multiple return. See figure 6-14 for example.

```

PROGRAM MAIN
READ *, X,Y,Z
CALL VAL(X,Y,Z)
END

C
SUBROUTINE VAL(A,B,C)
IF(A .LT. 0.0) GO TO 12
PRINT *, 'A IS POSITIVE'
RETURN
12 CONTINUE
PRINT *, 'A IS NEGATIVE'
RETURN
END

```

Figure 6-14. MULTIPLE Return Example

Alternate Return

An alternate return is used only within a subroutine subprogram. It returns control to the referencing program unit at a place other than the next executable statement after the CALL statement. The RETURN statement in the form RETURN exp is used for an alternate return.

Control is returned to a specified point in the referencing program unit. The specified point is a statement label in the referencing program unit. The statement labels must be included in the actual argument list, each preceded by an asterisk. Control returns to the statement label determined by the integer value of the alternate return expression. If the value of the expression is less than one, or greater than the number of asterisks in the SUBROUTINE statement or ENTRY statement that is the current entry point, control returns to the statement following the CALL statement. For example, if a CALL statement contains five statement labels and if the alternate return expression evaluates to three, control returns to the third statement label specified in the actual argument in the alternate return list.

Alternate returns can be used only in subroutine subprograms. See figure 6-15 for example.

```
PROGRAM MAIN
READ *, A,B,C
CALL XCOMP(A,B,C,*20,*30,*40)
20 CONTINUE
PRINT *, 'RETURNED TO STMT 20'
GO TO 10
30 CONTINUE
PRINT *, 'RETURNED TO STMT 30'
GO TO 10
40 CONTINUE
PRINT *, 'RETURNED TO STMT 40'
10 END
C
SUBROUTINE XCOMP(B1,B2,G,*,*,*)
IF(B1*B2 - 4.159) 11,12,13
11 CONTINUE
RETURN 2
12 CONTINUE
RETURN 1
13 CONTINUE
IF(B1 .GT. 32.) RETURN 3
RETURN
END
```

RETURN 1 is a return to statement 20 in the calling program; RETURN 2 is a return to statement 30; RETURN 3 is a return to statement 40. The subroutine contains both the normal RETURN statement and alternate RETURN.

Figure 6-15. Alternate Return Example

STATEMENT FUNCTIONS

A statement function is a user-defined procedure. It is a nonexecutable, single-statement computation that applies only to the program unit containing the definition.

Within a program unit, a statement function must appear after the specification statements and before the first executable statement in the unit. A statement function must not directly or indirectly reference itself.

Statement Function Definition

A statement function is specified by a single statement (figure 6-16 and example in figure 6-17). It is similar to an assignment statement.

```
fun([d[,d]. . .]) = expr
```

fun Is the function name.

d Is a statement function dummy argument.

expr Is an expression in which each primary is an expression expr enclosed in parentheses, or is one of the following:

- Constant
- Symbolic constant
- Variable reference
- Array element reference
- Intrinsic function reference
- Reference to a statement function which appears in the same program unit, before or after this statement
- External function reference
- Dummy procedure reference
- Substring reference

Figure 6-16. Statement Function

The symbolic name of the function is a variable and contains the value of the expression after execution. During execution, the actual argument expressions are evaluated, converted if necessary to the types of the corresponding dummy arguments according to the rules for assignment, and passed to the function. Thus, an actual argument cannot be an array name or a function name. In addition, if a character variable or array element is used as an actual argument, a substring reference to the corresponding dummy argument must not be specified in the statement function expression. The expression of the function is evaluated, and the resulting value is converted as necessary to the data type of the function.

```

PROGRAM SFUNC
INTEGER SN
DIMENSION AVG(25)
ADD(A,B,C,D) = A+B+C+D
AVRG(T1,T2,T3,T4) = ADD(T1,T2,T3,T4)/4
GRADE(SCORE,HSCORE) = (SCORE/HSCORE) * 100
SN=1
1 READ(*,100,END=20) S1,S2,S3,S4
100 FORMAT(4F5.1)
    AVG(SN)=AVRG(S1,S2,S3,S4)
    NS=SN
    SN = SN + 1
    GO TO 1
20 HIGH = AVG(1)
    DO 30 SN = 1, NS
    IF(AVG(SN) .GT. HIGH) HIGH = AVG(SN)
30 CONTINUE
    DO 40 SN=1, NS
    CRVEDG = GRADE(AVG(SN),HIGH)
    PRINT *, CRVEDG
40 CONTINUE
STOP
END

```

Figure 6-17. Examples of Statement Functions

The symbolic name of a statement function is local and must not be the same as any other local name in the program unit, except a common block name. The name of a statement function cannot be an actual argument and must not appear in an INTRINSIC or EXTERNAL statement. If the statement function is used in a function subprogram, then the statement function can contain a reference to the name of the function subprogram or any of its entry names as a variable, but not as a function.

Each variable reference in the expression can be either a reference to a variable within the same program unit or to a dummy argument of the statement function. Statement functions can reference dummy variables that appear in a SUBROUTINE, FUNCTION, or ENTRY statement, but that statement must precede the statement function. Statement function dummy arguments can have the same names as variables defined elsewhere in the same program unit without conflict. Any reference to the name inside the function refers to the dummy argument, and any reference to the name outside the function definition refers to the variable.

Statement Function Reference

A statement function is referenced through a statement function name (figure 6-18). When the statement function name is referenced in an expression, the statement function is evaluated. The actual arguments are evaluated and converted to the type of the corresponding dummy argument; the resulting values are used in place of the corresponding dummy arguments in evaluation of the statement function expression. The definition of a statement function must not directly or indirectly reference itself. The statement function name can appear anywhere in an expression where an operand of the same type can be used.

The type of the statement function result is the type of the statement function name. The arguments must agree in order and number with the corresponding dummy arguments.

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

fun	Is the name of the statement function.
a	Is an actual argument that must be an expression (except a character expression involving concatenation of a dummy argument with length (*)).

Figure 6-18. Statement Function Reference

A statement function can be referenced only in the program unit where the statement function appears.

PROGRAM UNIT AND PROCEDURE COMMUNICATION

Communication between the referencing program unit and the referenced procedure is accomplished by passing actual arguments and by using common blocks.

Common blocks can be used to pass data to a subprogram, but not to an intrinsic function or a statement function. Data must be passed to these functions through an argument list.

Passing arguments and use of common blocks is discussed below. The BLOCK DATA subprogram, which is used to initialize named common, is then discussed.

PASSING ARGUMENTS

Actual arguments in the referencing program unit are associated with the referenced procedure through dummy arguments.

Actual Arguments

Actual arguments appear in the argument list of the referencing program unit. The referencing program unit passes actual arguments to the referenced procedure. The procedure receives values from the actual arguments and returns values to the referencing program unit. Actual arguments can be constants, symbolic names of constants, variables, array names, array elements, function references, and expressions. An actual argument cannot be the name of a statement function within the referencing program unit.

Dummy Arguments

Dummy arguments appear in the argument list of the referenced procedure. Within the referenced procedure, the dummy arguments are associated with the actual arguments passed. Procedures use dummy arguments to indicate the types of actual arguments, the number of arguments, and whether each argument is a variable, array, procedure, or statement label. Dummy arguments for statement functions can only be variables. Since all names are local to the program unit, the same dummy argument name can be used in more than one procedure. A dummy argument appearing in a SUBROUTINE, FUNCTION, or ENTRY statement must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements except as a common block name. Dummy arguments used in array declarations for adjustable dimensions must be type integer. Dummy arguments representing array names must be dimensioned.

Argument Association

When a procedure is executed, the actual arguments and dummy arguments are matched up and each actual argument replaces each dummy argument. The type of the actual argument and the dummy argument must be the same. The actual arguments must be in the same order and there must be the same number as the dummy arguments in the referenced procedure. The actual arguments that are evaluated before the association of arguments include: expressions, substring expressions, and array subscripts. If the actual argument is a procedure name, the procedure must be available for execution at the time of the reference to the procedure.

A dummy argument is undefined unless it is associated with an actual argument. Argument association can exist at more than one level of procedure reference, and terminates within a program unit at the execution of a RETURN or END statement.

A subprogram reference can cause a dummy argument to be associated with another dummy argument in the referenced procedure. Any dummy arguments that become associated with each other can be referenced but must not be stored into during the execution of the procedure. For example, if a procedure is defined as:

```
SUBROUTINE ALPHA(X,Y)
```

and referenced with:

```
CALL ALPHA(A,A)
```

then the dummy arguments X and Y would each be associated with the actual argument A. X and Y would be associated with each other and therefore must not be stored into.

A subroutine reference can cause a dummy argument to become associated with an entity in a common block. For example, if a procedure contains the statements:

```
SUBROUTINE ALPHA(X)  
COMMON Y
```

and the referencing program unit contains:

```
COMMON A  
CALL ALPHA(A)
```

then the actual argument A causes the dummy argument X to become associated with Y, which is in blank common. In this case, X and Y cannot be stored into during execution of the subroutine.

The information below discusses argument association with reference to character length, variables, arrays, procedure arguments, asterisk arguments, and adjustable dimensions.

Character Length

For type character, both the dummy and actual arguments must be of type character, and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument of type character is greater than the length of the dummy argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are used as the dummy argument.

If a dummy argument is an array name, length applies to the entire array and not to each array element. Length of array elements in the dummy argument can be different from length of array elements in the actual argument. The total length of the actual argument array must be greater than or equal to the total length of the dummy argument array.

When an actual argument is a character substring, the length of the actual argument is the length of the substring. If the actual argument expression involves concatenation, the sum of the lengths of the operands is the length of the actual argument.

Variables

A variable in a dummy argument can be associated with a variable, array element, substring, or expression in the actual argument. A procedure can define or redefine the associated dummy argument if the actual argument is a variable name, array element name, or substring reference. The procedure cannot redefine the dummy argument if the actual argument is a constant, a symbolic constant, a function reference, an expression using operators, or an expression enclosed in parentheses.

Arrays

The array declaration in a type, COMMON, or DIMENSION statement provides the information needed for the array during the execution of the program unit. The actual argument array and the dummy argument array can differ in the number of the dimension and size of the array. A dummy argument array can be associated with an actual argument that is an array, array element, or array element substring.

If the actual argument is a noncharacter array name, the size of the actual argument array cannot be less than the size of the dummy argument array. Each actual argument array element is associated with the dummy argument array element that has the corresponding subscript value.

An association exists for array elements in a character array. Note that unless the lengths of the elements in the dummy and actual argument agree, the dummy and actual argument array elements might consist of different characters. For example, if a program unit has the following statements:

```
DIMENSION A(2)
CHARACTER A*2
.
.
.
CALL SUB(A)
```

and the subroutine has the following statements:

```
SUBROUTINE SUB(B)
DIMENSION B(2)
CHARACTER B*1
```

then the first character of A(1) corresponds to B(1) and the second character of A(1) corresponds to B(2).

If the actual argument is a noncharacter array element name, the size of the dummy argument cannot exceed (as+1-av), where as is the size of the actual argument array and av is the subscript value of the array element. For example, if the program unit has the following statements:

```
DIMENSION ARRAY(20)
.
.
.
CALL CHECK(ARRAY(3))
```

then the value of as is 20, and av is 3. The maximum dummy array size is 18 for the subroutine:

```
SUBROUTINE CHECK (DUMMY)
DIMENSION DUMMY(18)
.
.
.
SWAP= DUMMY(2)
```

Actual argument array elements are associated with dummy argument array elements, starting with the first element passed. In the example, DUMMY(2) is associated with ARRAY(4), and DUMMY(18) is associated with ARRAY(20).

The association for characters is basically the same as for noncharacter array elements. The actual argument for characters can be an array name, array element name, or array element substring name. If the actual argument begins at character storage position acu of an array, then the first character storage position of the dummy argument array becomes associated with character storage position acu of the actual argument array, and so forth to the end of the dummy argument array.

An actual argument that is an extended Hollerith constant is treated as a one-dimensional array with ten characters per word. For extended Hollerith constants of nH or "f" form, any unused characters in the last word are blank filled; for R"f" forms, the last word is right justified with left zero fill; for L"f" forms, the last word is left justified with right zero fill.

Procedure Arguments

A dummy argument that is a dummy procedure can be associated only with an actual argument that is an intrinsic function, external function, subroutine, or another dummy procedure. If the dummy argument is used as an external function, the actual argument that is passed must be a function or dummy procedure. The type of the dummy argument must agree with the type of result of all specific actual arguments that become associated with the dummy argument. When a dummy argument is used as an external function and is the name of an intrinsic function, the intrinsic function name corresponding to the dummy argument name is not available. If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine or dummy procedure, and the dummy argument must not appear in a type statement or be referenced as a function.

Asterisk Arguments

A dummy argument that is an asterisk can only appear in the argument list of a SUBROUTINE or ENTRY statement in a subroutine subprogram. The actual argument is an alternate return specifier in the CALL statement.

Adjustable Dimensions

Adjustable dimensions enable creation of a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine SUM can accept is 10 elements.

If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A, N)
DIMENSION A(N)
```

Value N is passed as an actual argument.

Adjustable dimensions can be passed through common variables. For example,

```
SUBROUTINE SUB(A)
COMMON/B/M,N
DIMENSION A(M,N)
```

Dimension of array A, in subroutine SUB, is specified by the values M and N passed through the common block B.

Character strings and arrays can also be adjustable. For example,

```
SUBROUTINE MESSAG(X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares X with a length of (*) to accept strings of varying size. Note that the length of the string is not passed explicitly as an actual argument.

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument, but is determined by the number of elements stored into the array. If an array is dimensioned *, the array in the calling program must be large enough to contain all the elements stored into it in the subroutine. See figure 6-19 for example.

```
SUBROUTINE CAT (A,M,N,B,C)
REAL A(M), B(N), C(*)
DO 10 I=1, M
10 C(I)=A(I)
DO 20 I=1, N
20 C(I+M)=B(I)
RETURN
END
```

Subroutine CAT places the contents of array A followed by the contents of array B into array C. The dimension of C in the calling program must be greater than or equal to M+N.

Figure 6-19. Example of Adjustable Dimensions in Subroutine

Use of the asterisk form of the adjustable dimension prevents subscript checking for the array, so the user must be careful not to reference outside the array bounds. Use of this form is preferable to the common practice of declaring arrays to have dimension 1.

USING COMMON BLOCKS

Common blocks can be used to transfer values between a referencing program unit and a subprogram. Common blocks can reduce the number of storage units required for a program by enabling two or more subprograms to share some of the same

storage units. The variables and arrays in a common block can be defined and referenced in all subprograms that contain a declaration of that common block. The names of the variables and arrays in the common block can be different for each subprogram. The association is by storage and not by name.

Common blocks cannot be used to pass data to intrinsic functions or statement functions; the method used to pass data to these procedures is through an argument list.

A reference to data in a common block is valid if the data is defined and is the same type as the type of the name used in the main program or subprogram. The exceptions to agreement between the type in common and the type of the reference are:

Either part of a complex entity can be referenced as real.

A Boolean entity can be referenced as integer.

In a subprogram, entities declared in a labeled common block can remain defined or become undefined at execution of an END or RETURN statement. If a labeled common block with the same name has been declared in a program unit that is directly or indirectly referencing the subprogram, the entities remain defined. Entities specified in a SAVE statement remain defined. Entities that are initially defined by DATA statements, and have neither been redefined nor become undefined, remain defined. Execution of a RETURN or END statement does not cause entities in blank common, or entities in any labeled common block that appears in the main program, to become undefined.

An example using common blocks in a subroutine is shown in figure 6-20.

```
PROGRAM AVRGE
COMMON NUMBR(10), STORE
REAL NUMBR, STORE
READ *, NUMBR
CALL SUM
STORE = STORE/10
PRINT *, 'AVERAGE= ', STORE
END

C
SUBROUTINE SUM
COMMON A(10), B
REAL A,B
B = 0.
DO 10 I= 1, 10
10 B = B + A(I)
CONTINUE
RETURN
END
```

Array NUMBR in program AVRGE and array A in subroutine SUM share the same locations in common. The values read into locations NUMBR(1) through NUMBR(10) are available to subroutine SUM.

Figure 6-20. Using Common

BLOCK DATA SUBPROGRAMS

The block data subprogram is a nonexecutable specification subprogram that can be used to enter initial values for variables and array elements in named common blocks. A program can have more than one block data subprogram. Only one block data subprogram can be unnamed; the 7-character name `BLKDAT` is assigned to the unnamed block data subprogram.

The `BLOCK DATA` statement (figure 6-21 and example in figure 6-22) must appear as the first statement of the block data subprogram. The name used for the block data subprogram must not be the same as any local variables in the subprogram. The name must not be the same as any other program unit or entry name in the program.

Block data subprograms can contain `IMPLICIT`, `PARAMETER`, `DIMENSION`, type, `COMMON`, `SAVE`, `EQUIVALENCE`, `LEVEL`, or `DATA` statements. A block data subprogram ends with an `END` statement. Data can be entered into more than one common block in a block data program. All variables having storage in the named common must be specified even if they are not all initially defined.

A `BLOCK DATA` subprogram can be referenced by an external statement in any program unit. If it is loaded from a library, the control statement `LDSET(USEP=blockdataname)` should be specified.

BLOCK DATA [sub]

sub Is the name of the block data subprogram.

Figure 6-21. `BLOCK DATA` Statement

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

The block data subprogram `ANAME` enters data into common blocks `CAT` and `DOG`. Initial values are defined for variables `X` and `Y` in block `CAT` and variable `R` in block `DOG`. No initial values are defined for variables `Z`, `S`, or `T`.

Figure 6-22. Example of `BLOCK DATA`

Certain procedures that are of general utility or that are difficult to express in FORTRAN statements are contained within the FORTRAN library. In general, these include the intrinsic functions and various subprograms that interface with different aspects of the operating system.

The intrinsic functions are used in expressions to calculate values.

The utility subprograms and functions include several different types of operations:

Accessing parameters on an LGO or EXECUTE control statement through the GETPARM subroutine

Accessing the seed for the random number generator function

Duplicating COMPASS macros that access operating system functions

Checking status and other input/output operations

Connecting files to interactive terminals

Implementing random access files through the MSIO routines when direct access files are not suitable

Debugging operations that dump memory, trace subroutine calls, affect error messages, and check data errors

Changing the collating sequence for character data evaluations

Static loading for programs that are not using the Common Memory Manager.

Utility subprograms are always called by name. A user-supplied subprogram with the same name overrides the utility supplied by the FORTRAN library.

INTRINSIC FUNCTIONS

Intrinsic functions are procedures that perform a set of calculations when the name appears in an expression in the referencing program unit. Intrinsic functions communicate with the referencing program unit through a single value associated with the function symbolic name. See section 6 for a general discussion of functions.

When the name of an intrinsic function duplicates another element in a program, the result depends on the element and the references. If a variable, array, or statement function is defined with the same name as an intrinsic function, the name is a local name that no longer refers to the intrinsic function. If an external function subprogram is written with the same name as an intrinsic function, use of the name references the intrinsic function, unless the name is declared as the name of an external function with the EXTERNAL statement described in section 2.

Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name. If an intrinsic function is typed something other than the default for that function, the compiler does not honor the type statement and generates an error.

If DB=TB is specified on the FTN5 control statement, many intrinsic functions perform argument range checking and issue an informative diagnostic at execution time.

GENERIC AND SPECIFIC FUNCTIONS

Certain intrinsic functions are generic, but have related specific functions. For example, the generic function name LOG computes the natural logarithm of an argument. Its argument can be real, double precision, or complex, and the type of the result is the same as the type of the argument. Specific function names ALOG, DLOG, and CLOG also compute the natural logarithm. The specific function name ALOG computes the log of a real argument and returns a real result. Likewise, the specific name DLOG is for double precision arguments and results, and the specific name CLOG is for complex arguments and results.

If a generic name and specific names exist, a generic name can be used in place of a specific name and is more flexible than a specific name. Except for type conversion generic functions, the type of the argument determines the type of the result.

Only a specific name can be used as an actual argument when passing the function name to a user-defined procedure or function.

BOOLEAN ARGUMENTS PASSED TO INTRINSIC FUNCTIONS

Boolean arguments passed to the intrinsic functions are treated differently, depending on the type of intrinsic function. If the type of intrinsic function is Boolean, the Boolean argument(s) are treated as Boolean. If the type of the intrinsic function is non-Boolean and it accepts one argument, and the argument is Boolean, the following rules apply:

For the generic function INT, the argument is treated as integer.

For the generic functions REAL, DBLE, and CMPLX, the argument is treated as real.

For other generic functions, if type integer is allowed, the argument is treated as integer; otherwise, the argument is treated as real.

For specific functions, the argument is treated as the required type, unless such treatment is invalid (required type is character).

For nonBoolean intrinsic functions that accept more than one argument, and at least one argument is Boolean, the following rules apply:

For generic function CMPLX, the argument(s) are treated as real.

For other generic functions, if all of the arguments are Boolean they are treated as type integer. Otherwise, the arguments are treated as the type of the non-Boolean argument, unless such treatment is invalid (non-Boolean argument(s) are type character).

For specific functions, Boolean arguments are treated as the required type, unless such treatment is invalid (required type is character).

Treating a Boolean argument as another type implies no change of bit pattern. When the required type is double precision or complex, the second word will be all zeros.

FUNCTION DESCRIPTIONS

Table 7-1 lists the intrinsic functions in alphabetical order by generic or specific name. The table shows the generic name, specific name, type of argument, type of function, number of arguments, intrinsic function, and definition. Table 7-2 lists the mathematical intrinsic functions in alphabetical order by function. This table shows the function, syntax, type of name, domain, definition, and range.

TABLE 7-1. INTRINSIC FUNCTIONS

Generic Name	Specific Name	Type of Argument	Type of Function	Number of Arguments	Intrinsic Function	Definition
ABS	IABS ABS DABS CABS	Integer Real Double Complex	Integer Real Double Real	1	Absolute Value	Defined as $ a $; if a is complex, square root of $((\text{real } a)^2 + (\text{imaginary } a)^2)$
ACOS	ACOS DACOS	Real Double	Real Double	1	Arccosine	Defined as $\arccos(a)$
-	AIMAG	Complex	Real	1	Imaginary part of complex argument	Imaginary part of $(ar, ai) = ai$
AINT	AINT DINT	Real Double	Real Double	1	Truncation	Defined as $\text{int}(a)$
-	AMAXO MAXI	Integer Real	Real Integer	2 thru 500	Choosing largest value	Defined as $\max(a_1, a_2, \dots, a_n)$
-	AMINO MINI	Integer Real	Real Integer	2 thru 500	Choosing smallest value	Defined as $\min(a_1, a_2, \dots, a_n)$
-	AND	Any type but character	Boolean	2 thru 500	Boolean product	Boolean result of .AND. operator
ANINT	ANINT DNINT	Real Double	Real Double	1	Nearest whole number	Defined as $\text{int}(a+.5)$ if a is positive or zero; $\text{int}(a-.5)$ if a is negative
ASIN	ASIN DASIN	Real Double	Real Double	1	Arcsine	Defined as $\arcsin(a)$
ATAN	ATAN DATAN	Real Double	Real Double	1	Arctangent	Defined as $\arctan(a)$
-	ATANH	Real	Real	1	Hyperbolic arctangent	Defined as $\text{arctanh}(a)$
ATAN2	ATAN2 DATAN2	Real Double	Real Double	2	Arctangent	Defined $\arctan(a_1/a_2)$
BOOL	-	Any type except logical	Boolean	1	Type conversion	Conversion to Boolean
-	CHAR	Integer	Character	1	Type conversion	Integer conversion to character
CMPLX	- - - -	Integer Real Double Complex	Complex Complex Complex Complex	1 or 2	Type conversion	Conversion to complex
-	COMPL	Any type but character	Boolean	1	Complement	Boolean result of .NOT. operator
-	CONJG	Complex	Complex	1	Conjugate of complex argument	Negation of imaginary part of $(ar, -ai)$
COS	COS DCOS CCOS	Real Double Complex	Real Double Complex	1	Cosine	Defined as $\cos(a)$, where a is in radians

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Generic Name	Specific Name	Type of Argument	Type of Function	Number of Arguments	Intrinsic Function	Definition
-	COSD	Real	Real	1	Cosine	Defined as $\cos(a)$, where a is in degrees
COSH	COSH DCOSH	Real Double	Real Double	1	Hyperbolic cosine	Defined as $\cosh(a)$
DBLE	- - - -	Integer Real Double Complex	Double Double Double Double	1	Type conversion	Conversion to double
DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double	2	Positive difference	Defined as $a_1 - a_2$ if a_1 is greater than a_2 ; 0 if a_1 is not greater than a_2
-	DPROD	Real	Double	2	Double precision product	Defined as $a_1 * a_2$
-	EQV	Any type but character	Boolean	2 thru 500	Equivalence	Boolean result of .EQV. operator
-	ERF	Real	Real	1	Error function	Defined as $\text{erf}(a)$
-	ERFC	Real	Real	1	Complementary error function	Defined as $1 - \text{erf}(a)$
EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex	1	Exponential	Defined as $e^{**}a$
-	ICHAR	Character	Integer	1	Type conversion	Character conversion to integer
-	INDEX	Character	Integer	2	Index of a substring	Location of substring a_2 in string a_1
INT	- INT IFIX IDINT -	Integer Real Real Double Complex	Integer Integer Integer Integer Integer	1	Type conversion	Conversion to integer, $\text{int}(a)$
-	LEN	Character	Integer	1	Length	Length of character string
-	LGE	Character	Logical	2	Lexically greater than or equal	True if a_1 follows a_2 , or $a_1 = a_2$, in ASCII collating sequence
-	LGT	Character	Logical	2	Lexically greater than	True if a_1 follows a_2 in ASCII collating sequence
-	LLE	Character	Logical	2	Lexically less than or equal	True if a_1 precedes a_2 , or $a_1 = a_2$, in ASCII collating sequence
-	LLT	Character	Logical	2	Lexically less than	True if a_1 precedes a_2 in ASCII collating sequence
-	LOCf	Any type	Integer	1	Location	Address of variable, array element, substring, subroutine, or external function
LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex	1	Natural logarithm	Defined as $\log_e(a)$

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Generic Name	Specific Name	Type of Argument	Type of Function	Number of Arguments	Intrinsic Function	Definition
LOG10	ALOG10 DLOG10	Real Double	Real Double	1	Common logarithm	Defined as $\log_{10}(a)$
-	MASK	Integer	Boolean	1	Mask	Boolean result of a1 left-justified l bits
MAX	MAX0 AMAX1 DMAX1	Integer Real Double	Integer Real Double	2 thru 500	Choosing largest value	Defined as max (a1,a2[,an]...)
MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double	2 thru 500	Choosing smallest value	Defined as min (a1,a2[,an]...)
MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double	2	Remaindering	Defined as $a1 - \text{int}(a1/a2)*a2$
-	NEQV	Any type but character	Boolean	2 thru 500	Non-equivalence	Same as exclusive OR
NINT	NINT IDNINT	Real Double	Integer Integer	1	Nearest integer	Defined as $\text{int}(a+.5)$ if a is positive or zero; $\text{int}(a-.5)$ if a is negative
-	OR	Any type but character	Boolean	2 thru 500	Boolean sum	Boolean result of .OR. operator
-	RANF	None	Real	0	Random number generator	Random number in range (0,1)
REAL	FLOAT REAL - - SNGL -	Integer Integer Real Complex Double Complex	Real Real Real Real Real Real	1	Type conversion	Conversion to real
-	SECOND	None	Real	0	Second	CPU time in seconds from start of job
-	SHIFT	Any type but character for a1; integer for a2	Boolean	2	Shift	Boolean result of a shifted a2 bit positions
SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double	2	Transfer of sign	Defined as $ a1 $ if $a2$ is ≥ 0 ; $- a1 $ if $a2$ is < 0
SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex	1	Sine	Defined as $\sin(a)$, where a is in radians
-	SIND	Real	Real	1	Sine	Defined as $\sin(a)$, where a is in degrees
SINH	SINH DSINH	Real Double	Real Double	1	Hyperbolic sine	Defined as $\sinh(a)$
SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex	1	Square root	Square root of (a)

TABLE 7-1. INTRINSIC FUNCTIONS (Contd)

Generic Name	Specific Name	Type of Argument	Type of Function	Number of Arguments	Intrinsic Function	Definition
TAN	TAN DTAN	Real Double	Real Double	1	Tangent	Defined as $\tan(a)$, where a is in radians
-	TAND	Real	Real	1	Tangent	Defined as $\tan(a)$, where a is in degrees
TANH	TANH DTANH	Real Double	Real Double	1	Hyperbolic tangent	Defined as $\tanh(a)$
-	XOR	Any type but character	Boolean	2 thru 500	Exclusive OR	Boolean result of .XOR. operator

TABLE 7-2. SUMMARY OF MATHEMATICAL INTRINSIC FUNCTIONS

Function	Syntax	Type of Name	Domain	Definition	Range
Arccosine (result in radians)	ACOS(y) ACOS(y) DACOS(y)	Generic Real Double	$ y \leq 1$	$\cos^{-1}(y)$	$0 \leq \text{ACOS}(y) \leq \pi$
Arcsin (result in radians)	ASIN(y) ASIN(y) DASIN(y)	Generic Real Double	$ y \leq 1$	$\sin^{-1}(y)$	$-\pi/2 \leq \text{ASIN}(y) \leq \pi/2$
Arctangent (result in radians)	ATAN(y) ATAN(y) DATAN(y)	Generic Real Double		$\tan^{-1}(y)$	$-\pi/2 \leq \text{ATAN}(y) \leq \pi/2$
Arctangent (2 arguments, result in radians)	ATAN2(y,x) ATAN2(y,x) DATAN2(y,x)	Generic Real Double	$x < 0, y < 0$ $x = 0, y < 0$ $x > 0$ $x = 0, y > 0$ $x < 0, y > 0$ $x = 0, y = 0$	$-\pi + \tan^{-1}(y/x)$ $-\pi/2$ $\tan^{-1}(y/x)$ $\pi/2$ $\pi + \tan^{-1}(y,x)$ error	$-\pi < \text{ATAN2}(y,x) < -\pi/2$ $-\pi/2 < \text{ATAN2}(y,x) < \pi/2$ $\pi/2 < \text{ATAN2}(y,x) \leq \pi$
Inverse hyperbolic tangent	ATANH(y) ATANH(y)	Generic Real	$ y < 1$	$\tanh^{-1}(y)$	$-16.98 < \text{ATANH}(y) < 16.98$
Trigonometric cosine (argument in radians)	COS(x) COS(x) DCOS(x)	Generic Real Double	$ x < 2^{47}$	$\cos(x)$	$-1 \leq \text{COS}(x) \leq 1$
	CCOS(x)	Complex	$ x \leq \pi * 2^{46}$ $ y \leq 741.66$	$\cos(x+iy)$	
Trigonometric cosine (argument in degrees)	COSD(x) COSD(x)	Generic Real	$ x < 2^{47}$	$\cos(x)$	$-1 \leq \text{COSD}(x) \leq 1$
Hyperbolic cosine	COSH(x) COSH(x) DCOSH(x)	Generic Real Double	$ x \leq 742.36$	$\cosh(x)$	$1 \leq \text{COSH}(x)$

TABLE 7-2. SUMMARY OF MATHEMATICAL INTRINSIC FUNCTIONS (Contd)

Function	Syntax	Type of Name	Domain	Definition	Range
Error function	ERF(x) ERF(x)	Generic Real		$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$	$-1 < \text{ERF}(x) < 1$
Complementary error function	ERFC(x) ERFC(x)	Generic Real	$x < 25.923$	$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$	$0 < \text{ERFC}(x) < 2$
Exponential	EXP(x) EXP(x) DEXP(x)	Generic Real Double	$-675.81 < x < 741.66$	e^x	$0 < \text{EXP}(x)$
	CEXP(x)	Complex	$-675.81 < x < 741.66$ $ y < *2^{46}$	$e^{(x+iy)}$	
Natural logarithm	LOG(x) ALOG(x) DLOG(x)	Generic Real Double	$x > 0$	$\log_e(x)$	
	CLOG(x)	Complex	$x^2 + y^2 \neq 0$	$\log_e(x+iy)$	$-\pi < \text{imaginary part} < \pi$
Common logarithm (base 10)	LOG10(x) ALOG10(x) DLOG10(x)	Generic Real Double	$x > 0$	$\log_{10}(x)$	
Trigonometric sine (argument in radians)	SIN(x) SIN(x) DSIN(x)	Generic Real Double	$ x < 2^{47}$	$\sin(x)$	$-1 < \text{SIN}(x) < 1$
	CSIN(x)	Complex	$ x < \pi * 2^{46}$ $ y < 741.66$	$\sin(x+iy)$	
Trigonometric sine (argument in degrees)	SIND(x) SIND(x)	Generic Real	$ x < 2^{47}$	$\sin(x)$	$-1 < \text{SIND}(x) < 1$
Hyperbolic sine	SINH(x) SINH(x) DSINH(x)	Generic Real Double	$ x < 742.36$	$\sinh(x)$	
Square root	SQRT(x) SQRT(x) DSQRT(x)	Generic Real Double	$x \geq 0$	$x^{0.5}$	$\text{SQRT}(x) \geq 0$
	CSQRT(x)	Complex	$x \geq 0, x < 0$		value in right half plane
Trigonometric tangent (argument in radians)	TAN(x) TAN(x) DTAN(x)	Generic Real Double	$ x < 2^{47}$	$\tan(x)$	
Trigonometric tangent (argument in degrees)	TAND(x) TAND(x)	Generic Real	$ x < 2^{47}$	$\tan(x)$	
Hyperbolic tangent	TANH(x) TANH(x)	Generic Real		$\tanh(x)$	$-1 < \text{TANH}(x) < 1$

The description of each intrinsic function is given below in alphabetical order.

ABS

ABS(a) is a generic function that returns an absolute value. The result is integer, real, or double precision, depending on the argument type. For an integer, real, or double precision argument, the result is $|a|$. For a complex argument, the result is the square root of (ar^2+ai^2) . The specific names are IABS, ABS, DABS, and CABS.

ACOS

ACOS(a) is a generic function that returns an arccosine. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ACOS and DACOS.

AIMAG

AIMAG(a) is a specific function that returns the imaginary part of a complex argument. The real result is ai , where the complex argument is (ar,ai) . AIMAG does not have a generic name.

AINT

AINT(a) is a generic function that returns an integer after truncation. The result is real. For a real or double precision argument, the result is 0 if $|a|$ is less than 1. If $|a|$ is greater than or equal to 1, the result is the largest integer with the same sign as argument a that does not exceed the magnitude of a . The specific names are AINT and DINT.

ALOG

ALOG(a) is a specific function that returns the natural logarithm of the argument. The argument is real and the result is real. The generic name is LOG.

ALOG10

ALOG10(a) is a specific function that returns the logarithm base 10 of the argument. The argument is real and the result is real. The generic name is LOG10.

AMAXO

AMAXO(a1,a2 [,an]...) is a specific function that returns the value of the largest argument. The 2 through 500 arguments are integer, and the result is real. AMAXO does not have a generic name.

AMAX1

AMAX1(a1,a2 [,an]...) is a specific function that returns the value of the largest argument. The 2 through 500 arguments are real, and the result is real. The generic name is MAX.

AMINO

AMINO(a1,a2 [,an]...) is a specific function that returns the value of the smallest argument. The 2 through 500 arguments are integer, and the result is real. AMINO does not have a generic name.

AMIN1

AMIN1(a1,a2 [,an]...) is a specific function that returns the value of the smallest argument. The 2 through 500 arguments are real, and the result is real. The generic name is MIN.

AMOD

AMOD(a1,a2) is a specific function that returns a modulus $a2$. The arguments are real, and the result is real. If $a2$ is zero, results are undefined. The $|a1/a2|$ must be less than 2^{48} . The generic name is MOD.

AND

AND(a1,a2 [,an]...) is a specific function that returns a Boolean product. The 2 through 500 arguments can be any type but character, and the result is Boolean. AND does not have a generic name.

ANINT

ANINT(a) is a generic function that returns the nearest whole number. The result is real or double precision, depending on the argument type. The specific names are ANINT and DNINT.

ASIN

ASIN(a) is a generic function that returns an arcsine. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ASIN and DASIN.

ATAN

ATAN(a) is a generic function that returns an arctangent. The result is expressed in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ATAN and DATAN.

ATANH

ATANH(a) is a specific function that returns a hyperbolic arctangent. The argument and result are real. See table 7-2. ATANH does not have a generic name.

ATAN2

ATAN2(a1,a2) is a generic function that returns an arctangent. The result is expressed in radians. The result is real or double precision, depending on the type of the arguments. The arguments must not both be 0. See table 7-2. The specific names are ATAN2 and DATAN2.

BOOL

BOOL(a) is a generic function that performs type conversion and returns a Boolean value. The argument can be integer, real, double precision, complex, character, or Boolean. For an integer, real, or Boolean argument, the result is the bit string constituting the data. For a double precision or complex argument, the result is the bit string after conversion of the argument to real with REAL(a). For a character argument, the result is the value of the Hollerith constant nff, where n is the length and f is the character value; if n is greater than 10, the rightmost characters are truncated. BOOL does not have specific names.

CABS

CABS(a) is a specific function that returns a real result from a complex argument. The generic name is ABS.

CCOS

CCOS(a) is a specific function that returns a complex result from a complex argument. The generic name is COS.

CEXP

CEXP(a) is a specific function that returns a complex result from a complex argument. The generic name is EXP.

CHAR

CHAR(a) is a specific function that returns the character value of an integer argument. The value returned depends on the collating sequence used. If the ASCII collating sequence is used, the argument must be in the range 0 through 63; the first character in the collating sequence corresponds to value 0, the second character to value 1, the third to value 2, and so forth. The result is the selection of a single character from the collating sequence. If, in a user-specified collating sequence, more than one character has weight a, the character returned can be any of them. CHAR does not have a generic name.

CLOG

CLOG(a) is a specific function that returns a complex result from a complex argument. The generic name is LOG.

CMPLX

CMPLX(a) or CMPLX(a1,a2) is a generic function that performs type conversion and returns a complex value. CMPLX can have one or two arguments. A single argument can be integer, real, double precision, or complex. If two arguments are used, the arguments must be of the same type and must both be integer, real, or double precision. For a single integer, real, or double precision argument, the result is complex, with the argument used as the real part and the imaginary part zero. For a single complex argument, the result is the same as the argument. For two arguments a1 and a2, the result is complex, with argument a1 used as the real part and argument a2 used as the imaginary part. CMPLX does not have specific names.

COMPL

COMPL(a) is a specific function that returns a complemented value. The argument can be any type except character, and the result is Boolean. If the argument is not Boolean, the argument is converted with BOOL(a). The result is the result of the logical operator .NOT. on a Boolean value. COMPL does not have a generic name.

CONJG

CONJG(a) is a specific function that returns a conjugate of a complex argument. The result is complex. For a complex argument (ar,ai), the result is (ar,-ai) with the imaginary part negated. CONJG does not have a generic name.

COS

COS(a) is a generic function that returns a cosine. The argument is assumed to be in radians. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are COS, CCOS, and DCOS.

COSD

COSD(a) is a specific function that returns a cosine. The argument is assumed to be in degrees. The argument and result are real. See table 7-2. COSD does not have a generic name.

COSH

COSH(a) is a generic function that returns a hyperbolic cosine. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are COSH and DCOSH.

CSIN

CSIN(a) is a specific function that returns the sine of the argument. The argument and result are complex. The generic name is SIN.

CSQRT

CSQRT(a) is a specific function that returns a complex result from a complex argument. The generic name is SQRT.

DABS

DABS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ABS.

DACOS

DACOS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ACOS.

DASIN

DASIN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ASIN.

DATAN

DATAN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ATAN.

DATAN2

DATAN2(a1,a2) is a specific function that returns a double precision result from a double precision argument. The generic name is ATAN2.

DBLE

DBLE(a) is a generic function that performs type conversion and returns a double precision result. The argument can be integer, real, double precision, or complex. For an integer or real argument, the result has as much precision as the double precision field can contain. For a double precision argument, the result is the argument. For a complex argument, the real part is used, and the result has as much precision as the double precision field can contain. DBLE does not have specific names.

DCOS

DCOS(a) is a specific function that returns a double precision result from a double precision argument. The generic name is COS.

DCOSH

DCOSH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is COSH.

DDIM

DDIM(a1,a2) is a specific function that returns a double precision result from double precision arguments. It returns the value of $a1-a2$; if $a1$ is less than $a2$, it returns 0. The generic name is DIM.

DEXP

DEXP(a) is a specific function that returns a double precision result from a double precision argument. The generic name is EXP.

DIM

DIM(a1,a2) is a generic function that returns a positive difference. The result is integer, real, or double precision, depending on the argument type. Both arguments must be the same type. The result is $a1-a2$ if $a1$ is greater than $a2$, and the result is 0 if $a1$ is less than or equal to $a2$. The specific names are DIM, IDIM, DDIM.

DINT

DINT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is AINT.

DLOG

DLOG(a) is a specific function that returns a double precision result from a double precision argument. The generic name is LOG.

DLOG10

DLOG10(a) is a specific function that returns a double precision result from a double precision argument. The generic name is LOG10.

DMAX1

DMAX1(a1,a2 [,an]...) is a specific function that returns a double precision result from 2 through 500 double precision arguments. The generic name is MAX.

DMIN1

DMIN1(a1,a2 [,an]...) is a specific function that returns a double precision result from 2 through 500 double precision arguments. The generic name is MIN.

DMOD

DMOD(a1,a2) is a specific function that returns a double precision result from two double precision arguments. If $a2$ is zero, results are undefined. The $|a1/a2|$ must be less than 2^{48} . The generic name is MOD.

DNINT

DNINT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is ANINT.

DPROD

DPROD(a1,a2) is a specific function that returns a double precision product. The arguments are real, and the result is double precision. The result is $a1*a2$. DPROD does not have a generic name.

DSIGN

DSIGN(a1,a2) is a specific function that returns a double precision result from two double precision arguments. The generic name is SIGN.

DSIN

DSIN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SIN.

DSINH

DSINH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SINH.

DSQRT

DSQRT(a) is a specific function that returns a double precision result from a double precision argument. The generic name is SQRT.

DTAN

DTAN(a) is a specific function that returns a double precision result from a double precision argument. The generic name is TAN.

DTANH

DTANH(a) is a specific function that returns a double precision result from a double precision argument. The generic name is TANH.

EQV

EQV(a1,a2 [,an]...) is a specific function that returns an equivalence result. The 2 through 500 arguments can be any type except character, and the result is Boolean. The result is the same as for the Boolean .EQV. operator. EQV does not have a generic name.

ERF

ERF(a) is a specific function that returns an error function result. The argument and result are real. See table 7-2. ERF does not have a generic name.

ERFC

ERFC(a) is a specific function that returns a complementary error function result. The argument and result are real. See table 7-2. ERFC does not have a generic name.

EXP

EXP(a) is a generic function that returns an exponential. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are EXP, DEXP, and CEXP.

FLOAT

FLOAT(a) is a specific function that returns a real result from an integer argument. The generic name is REAL.

IABS

IABS(a) is a specific function that returns an integer result from an integer argument. The generic name is ABS.

ICHAR

ICHAR(a) is a specific function that returns an integer value from a character argument. The value returned depends on the collating weight of the character in the collating sequence used. For the ASCII collating sequence, the first character in the collating sequence is at position 0, the second character at position 1, the third at position 2, and so forth. For a user-specified collating sequence, two or more characters can have the same value. The argument is a character value with a length of one character, and the value returned is the integer position of that character in the collating sequence. ICHAR does not have a generic name.

IDIM

IDIM(a1,a2) is a specific function that returns an integer result from integer arguments. It returns the value of $a1-a2$; if $a1$ is less than $a2$, it returns 0. The generic name is DIM.

IDINT

IDINT(a) is a specific function that returns an integer result from a double precision argument. The generic name is INT.

IDNINT

IDNINT(a) is a specific function that returns an integer result from a double precision argument. The generic name is NINT.

IFIX

IFIX(a) is a specific function that returns an integer result from a real argument. The generic name is INT.

INDEX

INDEX(a1,a2) is a specific function that returns the location of a substring within a string. Both arguments must be character string arguments. If string a2 occurs as a substring within string a1, the result is an integer indicating the starting position of the substring a2 within a1. If a2 does not occur as a substring within a1, the result is 0. If a2 occurs as a substring more than once within a1, only the starting position of the first occurrence is returned. INDEX does not have a generic name.

INT

INT(a) is a generic function that performs type conversion to integer. The result is integer, and the argument can be integer, Boolean, real, double precision, or complex. For an integer or Boolean argument, the result is the argument. For a real or double precision argument where |a| is less than 1, the result is 0. Where |a| is greater than or equal to 1, the result is the largest integer with the same sign as argument a that does not exceed the magnitude of a. For a complex argument, the real part is used, and the result is the same as for a real argument. The specific names are INT, IFIX and IDINT.

ISIGN

ISIGN(a1,a2) is a specific function that returns an integer result from two integer arguments. The generic name is SIGN.

LEN

LEN(a) is a specific function that returns the length of a character string. The argument is a character string, and the result is an integer indicating the length of the string. LEN does not have a generic name.

LGE

LGE(a1,a2) is a specific function that returns a result indicating lexically greater than or equal to. The arguments are character strings. The result is true only if a1 follows a2, or a1 is equal to a2, in the ASCII collating sequence. LGE does not have a generic name.

LGT

LGT(a1,a2) is a specific function that returns a result indicating lexically greater than. The arguments are character strings. The result is true only if a1 follows a2 in the ASCII collating sequence. LGT does not have a generic name.

LLE

LLE(a1,a2) is a specific function that returns a result indicating lexically less than or equal to. The arguments are character strings. The result is true only if a1 precedes a2, or a1 is equal to a2, in the ASCII collating sequence. LLE does not have a generic name.

LLT

LLT(a1,a2) is a specific function that returns a result indicating lexically less than. The arguments are character strings. The result is true only if a1 precedes a2 in the ASCII collating sequence. LLT does not have a generic name.

LOCF

LOCF(a) is a specific function that returns a location, that is, an address. The argument can be a variable, array element, substring, subroutine, or external function. For a noncharacter argument, the result is integer. For a character argument, LOCF returns the argument address, length, and beginning character position. A flag indicating extended memory or central memory residence is always returned. The format of the result for character arguments is shown in figure 7-1. LOCF does not have a generic name.

NOTE

Because of anticipated changes, use of this feature is not recommended. For guidelines, see appendix G.2.

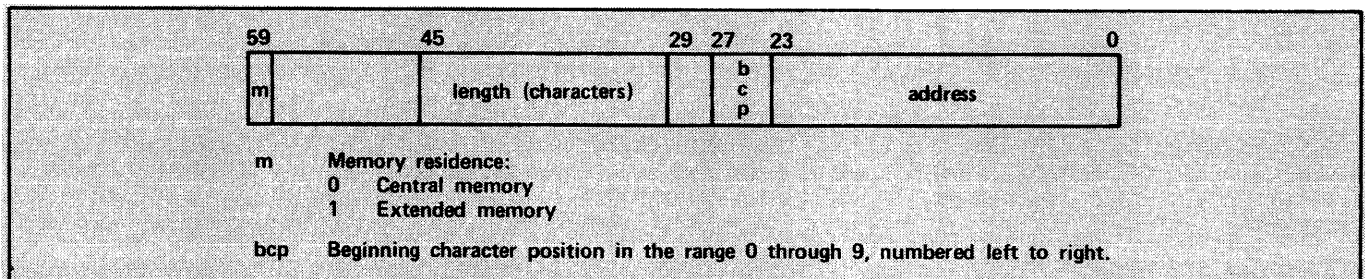


Figure 7-1. LOCF Result for Character Argument

LOG

LOG(a) is a generic function that returns a natural logarithm. The result is real, double precision, or complex, depending on the argument type. See table 7-2. For a complex argument (ar,ai), the range of the imaginary part of the result is $-\pi < ai < \pi$. The imaginary part of the result is only 0 when ar is greater than 0 and ai=0. The specific names are ALOG, DLOG, and CLOG.

LOG10

LOG10(a) is a generic function that returns a common logarithm. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are ALOG10 and DLOG10.

MASK

MASK(a) is a specific function that returns a Boolean result. The argument is integer or Boolean in the range 0 through 60. The result is a word of left-justified 1 bits followed by (60-a) 0 bits (a<0 → a=0). MASK does not have a generic name.

MAX

MAX(a1,a2 [,an]...) is a generic function that returns the largest value. The result is integer, real, or double precision, depending on the type of the 2 through 500 arguments. The specific names are MAXO, AMAX1, and DMAX1.

MAXO

MAXO(a1,a2 [,an]...) is a specific function that returns as an integer result the largest value from 2 through 500 integer arguments. The generic name is MAX.

MAX1

MAX1(a1,a2 [,an]...) is a specific function that returns as an integer result the largest value from the 2 through 500 real arguments. MAX1 does not have a generic name.

MIN

MIN(a1,a2 [,an]...) is a generic function that returns the smallest value from the 2 through 500 arguments. The result is integer, real, or double precision, depending on the type of arguments. The specific names are MINO, AMIN1, and DMIN1.

MINO

MINO(a1,a2 [,an]...) is a specific function that returns as an integer result the smallest value from the 2 through 500 integer arguments. The generic name is MIN.

MIN1

MIN1(a1,a2 [,an]...) is a specific function that returns as an integer result the smallest value from the 2 through 500 real arguments. MIN1 does not have a generic name.

MOD

MOD(a1,a2) is a generic function that returns the remainder of a1 divided by a2. The result is integer, real, or double precision, depending on the argument type. The result is $a1 - (\text{int}(a1/a2) * a2)$. If a2=0, results are undefined. The arguments a1 and a2 must not exceed 248-1. The specific names are MOD, AMOD, and DMOD.

NEQV

NEQV(a1,a2 [,an]...) is a specific function that returns a nonequivalence result. The result is Boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the Boolean exclusive or (.XOR.) operator. NEQV does not have a generic name.

NINT

NINT(a) is a generic function that returns the nearest integer. The result is integer, and the argument can be real or double precision. For a real or double precision argument where a is zero or positive, the result is $(\text{int}(a+.5))$. For an argument where a is negative, the result is $(\text{int}(a-.5))$. The specific names are NINT and IDNINT.

OR

OR(a1,a2 [,an]...) is a specific function that returns a Boolean sum. The result is Boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the Boolean .OR. operator. OR does not have a generic name.

RANF

RANF is a specific function that returns a random number. Since there is no argument, RANF is referenced as RANF(). The result is real and is in the exclusive range between 0 and 1. RANF does not have a generic name.

The seed for random number generation can be initialized and specified through the RANSET and RANGET subroutines described later in this section.

REAL

REAL(a) is a generic function that performs type conversion and returns a real result. The argument can be integer, real, double precision, or complex. For a complex argument (ar,ai), the result is real(ar). The specific names are REAL, FLOAT, and SNGL.

SECOND

SECOND is a specific function that returns the CPU time in seconds since the beginning of the job. Since there is no argument, SECOND is referenced as SECOND(). The result is real. SECOND does not have a generic name.

SHIFT

SHIFT(a1,a2) is a specific function that returns a shifted result. The argument a1 is any type but character, and argument a2 is integer or Boolean. The Boolean result is a1 shifted a2 bit positions. The shift is left circular if a2 is positive; right with sign extension and end off if a2 is negative. a2 is in the range -60 and +60. SHIFT does not have a generic name.

SIGN

SIGN(a1,a2) is a generic function that returns a value after a transfer of sign. The result is integer, real, or double precision, depending on the argument type. The result is |a1| if a2 is greater than or equal to 0. The result is -|a1| if a2 is less than 0. The specific names are SIGN, ISIGN, and DSIGN.

SIN

SIN(a) is a generic function that returns a sine. The argument is assumed to be in radians. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are SIN, DSIN, and CSIN.

SIND

SIND(a) is a specific function that returns a sine. The argument is assumed to be in degrees. The argument and result are real. See table 7-2. SIND does not have a generic name.

SINH

SINH(a) is a generic function that returns a hyperbolic sine. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are SINH and DSINH.

SNGL

SNGL(a) is a specific function that returns a real result from a double precision argument. The generic name is REAL.

SQRT

SQRT(a) is a generic function that returns a principal square root. The result is real, double precision, or complex, depending on the argument type. See table 7-2. The specific names are SQRT, DSQRT, and CSQRT.

TAN

TAN(a) is a generic function that returns a tangent. The argument is assumed to be in radians. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are TAN and DTAN.

TAND

TAND(a) is a specific function that returns a tangent. The argument is assumed to be in degrees. The argument and result are real. See table 7-2. TAND does not have a generic name.

TANH

TANH(a) is a generic function that returns a hyperbolic tangent. The result is real or double precision, depending on the argument type. See table 7-2. The specific names are TANH and DTANH.

XOR

XOR(a1,a2 [,an]...) is a specific function that returns an exclusive OR value. The 2 through 500 arguments can be any type but character, and the result is Boolean. The result is the same as for the Boolean exclusive or (.XOR.) operator. XOR does not have a generic name.

GETPARM SUBROUTINE

The GETPARM subroutine call (figure 7-2) is used to access parameters that have been declared on the execution control statement (see section 11). Each call to GETPARM returns the next parameter from the control statement.

CALL GETPARM (c1,c2,i)	
c1	Character variable substring or array element to receive the parameter name.
c2	Character variable substring or array element to receive the parameter value.
i	Integer return code. Possible values are: <ul style="list-style-type: none">-1 No user parameters remain on execution control statement (c1, c2 undefined).0 Normal return (values stored in c1 and c2).1 Parameter name only specified on execution control statement (c2 contains blanks).

Figure 7-2. GETPARM Call

RANDOM NUMBER ROUTINES

The following routines can be used in the generation of random numbers.

RANSET SUBROUTINE

The RANSET call (figure 7-3) initializes the seed of the intrinsic function RANF. Bit 0 is set to 1 (forced odd), and bits 59 through 48 are set to 1717 octal.

CALL RANSET(n)

n Is a 1-word bit pattern used to generate the seed for the function RANF.

Figure 7-3. RANSET Call

RANGET SUBROUTINE

The RANGET call (figure 7-4) obtains the current seed of the intrinsic function RANF between 0 and 1. The value returned to n is not necessarily normalized. The value returned can be passed to RANSET at a later time to regenerate the same sequence of random numbers.

CALL RANGET(n)

n Is the symbolic name to receive the random number seed.

Figure 7-4. RANGET Call

OPERATING SYSTEM INTERFACES

The following routines are used to communicate with the operating system. They are comparable to the operating system COMPASS macros.

NOTE

Refer to appendix G for recommendations on the use of these routines.

DATE FUNCTION

DATE (figure 7-5) returns the current date as the value of the function in the form Δ mm/dd/yy Δ (NOS/BE and SCOPE 2) or Δ yy/mm/dd. (NOS), where mm is the number of the month, dd is the day within the month, and yy is the year. The format might be different at a particular installation. The value returned is type character with a length of 10. DATE must be declared type character*10 in the calling program.

DATE()

Figure 7-5. DATE Function

JDATE FUNCTION

JDATE (figure 7-6) returns the current date as the value of the function in the form yyddd, where yy is the year and ddd is the number of the day within the year. The value returned is type character with a length of 5. JDATE must be declared type character*5 in the calling program. (JDATE is not available on SCOPE 2.)

JDATE()

Figure 7-6. JDATE function

TIME OR CLOCK FUNCTION

TIME (figure 7-7) or CLOCK (figure 7-8) returns the current reading of the system clock as the value of the function in the form Δ hh.mm.ss., where hh is hours from 0 to 23, mm is minutes, and ss is seconds. The value returned is type character with a length of 10; the first character in the value returned is system dependent and is not always a blank. TIME and CLOCK must be declared type character*10 in the calling program.

TIME()

Figure 7-7. TIME Function

CLOCK()

Figure 7-8. CLOCK Function

DISPLA SUBROUTINE

DISPLA (figure 7-9) places a name and a value in the dayfile. The character constant h cannot be more than 50 characters; k is a real or integer variable or expression and is displayed as an integer or real value. Characters with display codes greater than 57 octal are listed in the dayfile, but are replaced by blanks when displayed at the operator's console. If the first character is \$, the message will flash at the console (on NOS/BE).

CALL DISPLA(h,k)

- h Is a character expression to be displayed.
- k Is a real or integer variable or expression whose value is to be displayed.

Figure 7-9. DISPLA Call

REMARK SUBROUTINE

REMARK (figure 7-10) places a message in the dayfile. The maximum message length is 80 characters displayed 40 characters per line under NOS/BE, 90 characters displayed on one line under SCOPE 2, and one line of 80 characters under NOS. A message exceeding the maximum length is truncated. Characters with display codes greater than 57 octal are listed in the dayfile, but are replaced by blanks when displayed at the operator's console. If the first character is \$, the message will flash at the console (on NOS/BE).

CALL REMARK(h)

- h Is a character expression.

Figure 7-10. REMARK Call

SSWTCB SUBROUTINE

SSWTCB (figure 7-11) tests sense switches. If sense switch 1 is on, j is set to 1; if sense switch 1 is off, j is set to 2. The value i is 1 to 6. If i is out of range, an informative diagnostic is printed, and j is set to 2. The sense switches are set or reset by the computer operator or by the control statements SWITCH (NOS, NOS/BE, and SCOPE 2), ONSW (NOS only), and OFFSW (NOS only).

CALL SSWTCB(i,j)

- i Is a sense switch number.
- j Is an integer return variable.

Figure 7-11. SSWTCB Call

EXIT SUBROUTINE

EXIT (figure 7-12) terminates program execution and returns control to the operating system.

NOTE

Use of the STOP statement is preferable to CALL EXIT.

CALL EXIT

Figure 7-12. EXIT Call

CHEKPTX SUBROUTINE

CHEKPTX (figure 7-13 and example in figure 7-14) generates a checkpoint dump of the specified files.

For additional information about CHEKPTX, see the appropriate operating system reference manual.

RECOVR SUBROUTINE

RECOVR (figure 7-15) allows a user program to gain control at the time that normal or abnormal job termination procedures would otherwise occur. RECOVR is not available on SCOPE 2.

Initialization

Initialization of RECOVR at the beginning of a program establishes the conditions under which control is to be regained and specifies the address of user recovery code. If the stated condition occurs during program execution, control returns to the user code. If necessary, the system increases the CP time limit, input/output time limit, or mass storage limit to provide an installation-defined minimum of time and mass storage for RECOVR processing. No limit is increased more than once in a job. RECOVR can be called more than once during program initialization to reference different user recovery subroutines. These calls to RECOVR can use different combinations of conditions for the same or different user recovery subroutines.

In an overlay-structured program, calls to RECOVR as well as the user recovery subprograms should be in the (0,0) overlay.

A checksum of the user recovery code can be requested during initialization. If flagged conditions subsequently occur, RECOVR again checksums the code before returning control to it. This gives some assurance of user code integrity before it is executed. If the checksum parameter is zero, no checksum is done.

Specification

No more than ten routines can be specified by RECOVR in one program. If an error occurs and more than one routine has been established for that error, the routines are called successively, with the routine most recently specified called first. The second specification of a subroutine overrides its previous parameters. This override can be used to remove a subroutine from the RECOVR list by passing a mask of zero.

CALL CHEKPTX(filelist,SP)

filelist Is an array in the following format:

	59	17	11	0	
Word 1		n	0000		
Word 2	lfn ₁		f ₁		
Word 3	lfn ₂		f ₂		
	≈				≈
Word n+1	lfn _n		f _n		

- SP Is a flag indicating whether or not all files assigned to the job are to be checkpointed. If SP is zero, all local files are checkpointed. If SP is nonzero, the files specified by filelist are checkpointed.
- n Is the number of files in the following list, to a maximum of 42.
- lfn_i Is the name (in left-justified display code) of user storage files to be processed.
- f_n Is a number indicating specific manner in which lfn is to be processed (does not apply to SCOPE 2).
- 0 Mass storage file is copied from beginning-of-information to its position at checkpoint time, and only that portion will be available at restart. The file is positioned at the latter point.
 - 1 Mass storage file is copied from its position at check time to end-of-information, and only that portion will be available at restart. The file is positioned at the former point.
 - 2 Mass storage file is copied from beginning-of-information to end-of-information; the entire file will be available at restart time. The file is positioned at the point at which the checkpoint was taken.
 - 3 The last operation on the file determines how the mass storage file is copied.

Figure 7-13. CHEKPTX Call

```

DIMENSION IFILES(4)
IFILES(1) = 0"30000"
IFILES(2) = L"TAPE1" .OR. 0"10000"
IFILES(3) = L"TAPE2" .OR. 0"30000"
IFILES(4) = L"TAPE3"
.
.
.
CALL CHEKPTX(IFILES, 3)

```

Files TAPE1, TAPE2, and TAPE3 will be checkpointed, as these are the file names established in words 2-4 of array IFILES. The first word of the array establishes the number of files in the following list.

Figure 7-14. CHEKPTX Example

Error Conditions

If one of the user's selected error conditions occurs, RECOVR gains control, performs internal tasks, and then transfers control to the user's recovery subroutines. The following three arguments are passed to the user's recovery subroutine:

1. A 26-word integer array. The first 16 words are an image of the exchange package; the 17th word is the contents of RA+1. The 18th word contains a zero value unless it is changed by the user's recovery subroutine (discussed later in this section). The 19th through 26th words contain a copy of the 2nd through 9th words of the reprieve parameter block. Refer to the appropriate operating system reference manual for detailed information.

CALL RECOVER(name,flags,checksum,addr1,addr2)																	
name	Is the name of subroutine to be executed if flagged conditions occur (must be specified in an EXTERNAL statement).																
flags	In an octal value for conditions under which recovery code is to be executed, as outlined below. Conditions can be combined as desired, with octal values up to 377 allowed. <table border="0" style="margin-left: 20px;"> <tr><td>001</td><td>Arithmetic mode error.</td></tr> <tr><td>002</td><td>PP call or auto-recall error.</td></tr> <tr><td>004</td><td>Time or storage limit exceeded.</td></tr> <tr><td>010</td><td>Operator drop, kill, or rerun.</td></tr> <tr><td>020</td><td>System abort.</td></tr> <tr><td>040</td><td>CP abort.</td></tr> <tr><td>100</td><td>Normal termination.</td></tr> <tr><td>200</td><td>Terminal interrupt.</td></tr> </table>	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.	200	Terminal interrupt.
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.																
200	Terminal interrupt.																
checksum	Is the last word address of recovery code to be checksummed; 0 if no checksum is desired; a negative number if the address parameters follow, specifying the area where checksumming is desired.																
addr1	Is the first word address of recovery code to be checksummed (used only when checksum is a negative number).																
addr2	Is the last word address of recovery code to be checksummed (used only when checksum is a negative number).																

Figure 7-15. RECOVER Call

- A flag that, upon return, determines the type of program termination. This flag is initially set to zero. The possible types of program termination are discussed later in this section.
- An array, starting at RA+1, that allows a subroutine to access all of the user's field length.

If the recovery subroutine was called because of normal termination, the subroutine, before returning, should flush the buffers of all output files. Buffers can be flushed by an ENDFILE, REWIND, or CLOSE statement.

Upon return from a user's recovery subroutine, the 18th word of the reprieve package and the argument flag contain values set by the user's recovery

subroutine. If the 18th word of the reprieve package is set to 4, the program immediately resumes normal execution, and the argument flag is ignored. If the 18th word is not set to 4, the argument flag, which is set by the user's recovery subroutine, is checked. If the flag is set to nonzero, the job terminates normally, as if no errors had occurred. This happens after all the recovery subroutines have been processed. If the flag remains zero for all the recovery subroutines, the job continues as if RECOVER had not been called; that is, the original system error code is reset and processed.

If the flags are not set by any of the recovery subroutines, the job continues as if RECOVER had not been called; that is, the original system error code is reset and processed.

For additional information about RECOVER, see the appropriate operating system reference manual.

INPUT/OUTPUT SUBPROGRAMS

Utility subprograms involved in input/output are discussed below. These include subprograms for status checking, extended memory, interactive terminals, tape labels, and mass storage.

STATUS CHECKING

Status checking for input/output statements such as READ and WRITE should be performed with the optional parameters of these statements (section 5), but can also be performed with the functions UNIT, EOF, and IOCHECK. UNIT and EOF return an end-of-file status for any of the following conditions:

- End-of-section (for file INPUT only)
- End-of-partition
- Nondeleted W format flag record
- Embedded tape mark
- Terminating double tape mark
- Terminating end-of-file label
- Embedded zero length level 17 block

The functions UNIT and IOCHECK return a parity error indication for every record within or spanning a block containing a parity error; such an indication, however, does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the CYBER Record Manager input/output routines.

On SCOPE 2 only, parity status can be checked on write operations that access mass storage files when the write check option has been specified on the REQUEST statement for the file. (See the SCOPE 2 reference manual.) Write parity errors for other types of devices (such as staged/on-line tape) are detected by the operating system, and a message to this effect is written in the dayfile.

UNIT Function

UNIT (figure 7-16) checks the status of a BUFFER IN or BUFFER OUT operation for an end-of-file or parity error condition on logical unit u. When UNIT is referenced, the user program does not regain control until input/output operations on the unit are complete. The function returns the following values:

- 1. Unit ready, no end-of-file or parity error encountered on the previous operation.
- +0. Unit ready, end-of-file encountered on the previous operation.
- +1. Unit ready, parity error encountered on the previous operation.

UNIT(u,a,b)

- u Is the unit specifier.
- a Is the first variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.
- b Is the last variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.

Figure 7-16. UNIT Function

Example:

```
BUFFER IN (5,1) (B(1), B(100))
IF (UNIT(5,B(1), B(100)) 12,14,16
```

Transfers control to the statement labeled 12, 14, or 16 if the value returned is -1., 0., or +1., respectively.

Parameters a and b should always be included. If these parameters are omitted (as in most older programs), there is a slight chance that at OPT=2, execution errors will result or that the elements between a and b will not contain the expected values.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. UNIT should only be called for a file processed by BUFFER statements.

The UNIT function is of type real.

EOF Function

EOF (figure 7-17) tests for an end-of-file condition on unit u following a formatted, list directed, NAMELIST, or unformatted sequential read. Zero is returned if no end-of-file is encountered, or a nonzero value if end-of-file is encountered. If an end-of-file is encountered, EOF clears the indicator before returning control.

EOF(u)

- u Is the unit specifier.

Figure 7-17. EOF Function

Example:

```
IF (EOF(5) .NE. 0) GO TO 20
```

Transfers control to statement 20 if an end-of-file is encountered on unit 5.

If the IOSTAT= or END= parameter is not used in the READ statement that reads the end of file, the program will be terminated before EOF can be used to check for end of file.

The EOF function returns a zero value following read or write operations on random access files (files accessed by READMS/WRTIMS), 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.

Records following an end-of-file can be read by either issuing a CLOSE statement followed by an OPEN statement on the file or by using the EOF function. CLOSE/OPEN is the preferred method.

The EOF function does not replace the END= or the IOSTAT= parameters in the READ statement.

The EOF function is of type real.

IOCHEC Function

IOCHEC (figure 7-18) tests for a parity error on unit u following a formatted, list directed, NAMELIST, or unformatted read. The value zero is returned if no error has been detected. If a parity error occurs, IOCHEC clears the parity indicator before returning. Parity errors are handled in this way regardless of the type of the external device.

IOCHEC(u)

- u Is the unit specifier.

Figure 7-18. IOCHEC Function

Example:

```
J=IOCHEC(6)
IF(J .NE. 0) GO TO 25
```

Zero value is returned to J if no parity error occurs and nonzero if an error does occur; control transfers to the statement labeled 25 if an error occurs.

LENGTH Function and LENGTHX Subroutine

The LENGTH function (figure 7-19) or the LENGTHX subroutine (figure 7-20) returns information regarding the previous BUFFER IN or READMS call of the file designated by u. The argument nw of the value of LENGTHX is set to the number of words read. The argument ubc is set to the number of unused bits in the last word of the transfer. The arguments nw, ubc, and value returned are type integer.

```
LENGTH(u)

u    Is the unit specifier.
```

Figure 7-19. LENGTH Function

```
CALL LENGTHX(u,nw,ubc)

u    Is the unit specifier.

nw   Is the argument set to the number of
      words read.

ubc  Is the argument set to the number of
      unused bits in the last word of the
      transfer.
```

Figure 7-20. LENGTHX Call

After an unformatted BUFFER IN on a 9-track S or L tape, the unused bit count parameter of LENGTHX is rounded down so as to indicate a whole number of 6-bit characters. For example, a BUFFER IN of a 23-character record returns a length of four words with an unused bit count of 54, even though the actual unused bit count is 56.

If an odd number of words is written to a 9-track S or L tape by an unformatted BUFFER OUT, the record on the tape contains four additional zero bits at the right so as to be a whole number of 8-bit characters. If such a record is subsequently read by BUFFER IN, the length indication in LENGTH or LENGTHX is one word greater than the number of words originally written.

LENGTHX can be used after a formatted READ to determine the length of the previous record prior to blank padding by CYBER Record Manager. The argument nw is set to the number of words in the record, and ubc is set to the unused bits in the last word. Actual record length can then be computed by:

$$10*(nw-1) + (10-ubc/6)$$

For a file accessed by buffer statements, LENGTH or LENGTHX should be called only after a call to UNIT ensures that input/output activity is complete; otherwise, file integrity might be endangered.

EXTENDED MEMORY

The following subprograms are used to transfer data in extended memory.

MOVLEV Subroutine

MOVLEV (figure 7-21) transfers n consecutive words of data between a and b. MOVLEV can be used to transfer blocks of data between ECS, LCM, or UEM and central memory. The arguments a and b are variables or array elements; n is an integer value. The argument a is the starting address of the data to be moved, and b is the starting address of the receiving location.

```
CALL MOVLEV(a,b,n)

a    Specifies a variable that is the starting ad-
      dress of the data to be moved.

b    Specifies a variable that is the starting
      address of the receiving location.

n    Specifies the number of words to be
      moved.
```

Figure 7-21. MOVLEV Call

No conversion is performed by MOVLEV. If data from a real variable is moved to a type integer receiving field, the data remains real.

Example:

```
CALL MOVLEV (A, I, 1000)
```

After the move, I does not contain the integer equivalent of A.

Example:

```
DOUBLE PRECISION D1(500), D2(500)
CALL MOVLEV (D1, D2, 1000)
```

Since D1 is defined as double precision, n should be set to 1000 to move the entire D1 array.

NOTE

MOVLEV of character data is not allowed.

MOVLCH Subroutine

MOVLCH (figure 7-22) transfers n consecutive characters between a and b. MOVLCH can be used to transfer characters between ECS, LCM, or UEM and central memory. MOVLCH can also transfer characters between two blocks of storage resident in the same level of memory, LCM to LCM, ECS to ECS, or UEM to UEM. Both arguments a and b must be of type character; a diagnostic is issued if one, or both, is of any other type.

Example:

```
CHARACTER * 123, CH1(10), CH2(5)
CALL MOVLCH (CH1(8), CH2(3), 369)
```

The last three elements of character array CH1 are to be moved to the last three elements of character array CH2. Each element is 123 characters long; therefore, the total number of characters to move is $3*123=369$.

CALL MOVLCH (a, b, n)

- a Specifies a variable, array element, or substring that represents the starting location of the character string to be moved.
- b Specifies a variable, array element, or substring that represents the starting location of the receiving area.
- c Specifies the number of characters to move.

Figure 7-22. MOVLCH Call

INTERACTIVE TERMINALS

If a program to be run interactively calls for input/output operations through the user's remote terminal, all files to be accessed through the terminal must be formally associated with the terminal at the time of execution.

In particular, the file INPUT must be connected to the terminal if data is to be entered there and a numbered logical unit is not designated in the READ statement. The file OUTPUT must be connected to the terminal if execution diagnostics are to be displayed or printed at the terminal, or if data is to be displayed or printed there and a numbered unit is not designated in the WRITE statement. These files are automatically connected to the terminal when the program is executed under NOS/BE, using the RUN command of the EDITOR utility of INTERCOM, or under the NOS FORTRAN subsystem.

Under all operating systems, the user can connect any file from within the program by using the CONNEC subroutine call. Under INTERCOM, any file can be connected to the terminal by the CONNECT command. More information about INTERCOM is in the INTERCOM Version 5 Reference Manual and the INTERCOM Version 5 Interactive Guide for Users of FORTRAN. Under HELLO7, for SCOPE 2, any file can be connected by providing a FILE control statement specifying CNF=YES. More information about NOS is in the NOS time-sharing user's reference manual and the reference manuals describing the interactive facility.

Under any system, if a file specified in a CONNEC exists as a local file but is not connected at the time of the call, the file's buffer is flushed before the file is connected to the terminal.

The following subroutines are used to connect and disconnect files from interactive terminals.

CONNEC Subroutine

CONNEC (figure 7-23) connects files to the terminal.

Under NOS, if CONNEC specifies an existing local file, the buffers for the file are flushed (if it is an output file) and the file is returned. A subsequent DISCON for the file causes the connected file to be returned, but the preexisting file is not reassociated with the file name.

CALL CONNEC(u,cs)

- u Is the unit designator.
 - cs Is an optional character set designator (applicable to NOS/BE only).
-
- 0 Display code (default)
 - 1 ASCII-128
 - 2 ASCII-256

Figure 7-23. CONNEC Call

For a program run under NOS, any file can be connected to the terminal by the ASSIGN command. In addition, the user can connect any file from within the program by using CONNEC.

Data input or output through a terminal under NOS is represented ordinarily in a standard 64-character set. However, the user can elect to have data represented in an ASCII 128-character set (which provides the capability for recognizing control codes and lowercase, as well as uppercase, letters) by entering the ASCII command. Characters contained in the standard set are stored internally in 6-bit display code, whether or not the ASCII command has been entered. The additional characters which complete the ASCII 128-character set are stored internally in 6/12-bit display code if the ASCII command has been entered; otherwise, they are mapped into the standard 64-character set and stored internally in 6-bit display code. See appendix A.

Under NOS/BE, if cs is not specified, it is set to 0. If display code is selected, input/output operations must be formatted, list directed, NAMELIST, or buffered. If either of the ASCII codes is selected, input/output operations must be either formatted or buffered. When a CONNEC specifies a file already connected with the character set specified, the call is ignored. If the file specified is already connected with a character set other than that specified, cs is reset accordingly.

Data input or output through a terminal under INTERCOM is represented ordinarily in a CDC or ASCII 64-character set, depending on installation option. For these sets, ten characters in 6-bit display code are stored in each central memory word. As described above, a terminal user can specify from within a program that data represented in an ASCII 128-character set (providing the capability for recognizing lowercase letters and control codes) or an ASCII 256-character set (providing the capability for recognizing lowercase letters, control codes, and parity) be input or output through the terminal. For the ASCII 128-character and 256-character sets, characters are stored in five 12-bit bytes in each central memory word. Characters in the ASCII 128-character set are represented in 7-bit ASCII code right-justified in each byte with binary zero fill. Characters in the ASCII 256-character set are represented in 8-bit ASCII code right-justified in each byte with binary zero fill. See appendix A.

Some devices (plotters and other graphics devices) require data to be sent and received in full 8-bit ASCII mode. This can be done from FORTRAN by using the transparent mode capability of NOS and BUFFER IN/BUFFER OUT. Buffer input/output to a file of record type S should be used to send or receive full 8-bit data. See the NOS reference manual for details on transparent mode terminal input/output.

Example:

```

.
.
.
INTEGER I(2)
DATA I/0"0007 4007 4101 4141 4102",
0"4142 4007 0000 0000 0000"/
CALL CONNec (2)
BUFFER OUT (2,1) (I(1), I(2))
IF (UNIT(2) .GT. 0 ) STOP 'OUTPUT DEVICE ERROR'
.
.
.

```

This program segment outputs the first 12 bits of 0007 to select transparent mode output. The data for the output comes next. Each 8-bit quantity is right-aligned in a 12-bit field, which must have the top bit set. The data is terminated by 12 to 60 bits of zeros right-aligned in an output word.

When data represented in either ASCII character set code is transferred with a formatted input/output statement, the maximum record length should be specified in the PROGRAM statement as twice the number of characters to be transferred (section 6). Allowance should also be made in input/output operations for the fact that internal characters require twice as much space as external characters.

DISCON Subroutine

DISCON (figure 7-24) disconnects a file from within a program. This request is ignored if the specified file is not connected. After execution of

CALL DISCON(u)

u Is the unit specifier.

Figure 7-24. DISCON Call

this statement under NOS/BE, the specified file remains local to the terminal. In addition, if the file existed prior to connection, the file name is reassocated with the information contained on the device where the file resided prior to connection. Data written to a connected file is not contained in the file after it is disconnected. Under NOS, DISCON causes the connected file to be returned; the disconnected file name is not reassocated with the preexisting information.

TAPE LABEL SUBROUTINE

LABEL (figure 7-25) passes tape label information to the operating system (LABEL is recognized, but ignored on SCOPE 2.) Before LABEL can be used, the control statement that requests the tape for the job must specify that the tape has labels. LABEL should not be used with files accessed with CYBER Record Manager interface routines.

CALL LABEL(u,labinfo)

u Is the unit specifier.

labinfo Specifies the array for label information.

Figure 7-25. LABEL Call

The value labinfo is the name of a 4 word array containing label information in the format shown in table 7-3.

TABLE 7-3. LABINFO BLOCK CONTENT

Word	Bit Positions	Character Length	Contents	Default for Output
1	59 thru 0	10	File identifier	None. Entire field must be supplied by the user.
2	59 thru 18	7	File identifier continued	None. Entire field must be supplied by the user.
	17 thru 0	3	File sequence number	001†
3	59 thru 48	2	Generation version number	00†
	47 thru 30	3	Retention cycle	000†
	29 thru 0	5	Creation date (YYDDD)	Today's Julian date.
4	59 thru 24	6	Set identifier (VSN)	Blanks (field can be blank or zero filled).
	23 thru 0	4	File section number	0001†

† Field must be display code zero filled.

Default output values are specified for a field by filling it with binary zeros. If any of the fields are improperly initialized, NOS generates an LX BUFFER/FET PARAMETER ERROR and aborts the job on output. For further information concerning the fields, consult the appropriate operating system reference manuals concerning words 9-12 of the file environment table (FET).

The actions that occur at the time of the LABEL call are:

Passing the address and length of the user label block to CYBER Record Manager.

Forcing a rewind on any succeeding open of the logical unit.

The LABEL reference occurs before any access is made to the logical unit. If a WRITE subsequently occurs on the logical unit, the Record Manager routines write a label formatted from the LABINFO area onto the tape as a HDR1 label. Or, when a close or endfile operation is performed at job termination, the routines write an EOF1 label formatted from LABINFO. If a READ subsequently occurs on the logical unit, and a label is encountered on the opening read or an EOF condition, the label information is reformatted and placed into LABINFO.

MASS STORAGE

Mass storage input/output (MSIO) subroutines allow the user to create, access, and modify files on a random basis without regard for their physical positioning. Each record in the file can be read or written at random without logically affecting the remaining file contents. The length and content of each record are determined by the user. A random file can reside on any mass storage device. CYBER Record Manager word addressable file organization is used to implement MSIO files. The CYBER Record Manager reference manual contains details of word-addressable implementation.

A file processed by mass storage subroutines should not be processed by any other form of input/output.

The following paragraphs discuss random file access, OPENMS, WRITMS, READMS, CLOSMS, and STINDX.

Random File Access

A randomly accessible file capability is provided by the mass storage input/output subroutines. Random files offer the same advantages as direct access files (described in section 5). In a random file, as in a direct access file, any record can be read, written, or rewritten directly, because the file resides on a random access mass storage device that can be positioned to any portion of a file.

NOTE

Direct access files should be used where possible because they are ANSI standard. However, applications requiring variable length randomly accessible records must use the random file subroutines since the standard direct access file capability allows only fixed length records. Direct access files do not use an index; the disk addresses of the records are calculated using the record length and record number. Thus, an arbitrarily large record number, such as a social security number, could cause a file to be excessively large. For these files, the record ordinal should be used as the record number.

To permit random accessing, each record in a random file is uniquely and permanently identified by a record key. A key is an 18- or 60-bit quantity, selected by the user and included as a parameter on the call to read or write a record. When a record is first written, the key in the call becomes the permanent identifier for that record. The record can be retrieved later by a read call that includes the same key, and it can be updated by a write call with the same key.

When a random file is in active use, the record key information is kept in an array in the user's field length. The user is responsible for allocating the array space by a DIMENSION, type, or similar array declaration statement, but must not attempt to manipulate the array contents. The array becomes the directory or index to the file contents. In addition to the key data, it contains the word address and length of each record in the file. The index is the logical link that enables the mass storage subroutines to associate a user call key with the hardware address of the required record.

The index is maintained automatically by the mass storage subroutines. The user must not alter the contents of the array containing the index in any manner: to do so might result in destruction of the file contents. (In the case of a subindex, the user must clear the array before using it as a subindex, and read the subindex into the array if an existing file is being reopened and manipulated. However, individual index entries should not be altered.)

When a permanent file that was created by mass storage input/output routines is to be modified it must be attached with modify and extend permissions (append permission under NOS). Under NOS/BE and SCOPE 2, the EXTEND control statement should be used after the file is modified. Failure to extend the file can render it unusable.

In response to a call to open the file, the mass storage subroutine automatically clears the assigned index array. The index array should be noncharacter to insure that it begins on a word boundary. If an existing file is being reopened,

the mass storage subroutines locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to the mass storage device. When the file is reopened, by the same job or another job, the index is again read into the index array space provided, so that file manipulation can continue.

Object time input/output subroutines control the transfer of records between central memory and mass storage.

NOTE

The ARG=FIXED parameter cannot be specified on the FTN5 control statement if any default parameters are used in the following routines.

OPENMS Subroutine

OPENMS (figure 7-26) opens the mass storage file and informs the system that it is a random (word addressable) file.

The array (ix) specified in the call is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

Example:

```
DIMENSION I(11)
CALL OPENMS (5,I,11,0)
```

These statements prepare for random input/output on the file TAPE5 using an 11-word (10 entry) master index of the number type. If the file already exists, the master index is read into memory starting at address I.

WRITMS Subroutine

WRITMS (figure 7-27) transmits data from central memory to the file.

CALL OPENMS(u,ix,lngth,t)

- u Is the unit specifier.
- ix Is the name of the array containing the master index.
- lngth Is the length of the master index; for a number index $lngth \geq (\text{number of entries in master index})+1$; for a name index $lngth \geq 2 * (\text{number of entries in master index})+1$.
- t Is the type of index
 - 0 File has a number master index.
 - 1 File has a name master index.

Figure 7-26. OPENMS Call

The end-of-data (for $r=-1$ and $r=0$) is defined to be immediately after the end of the data record which is closest to end-of-information. The first record written at end-of-data overwrites the old index.

CYBER Record Manager operates more efficiently if n is always a multiple of 64. The r parameter can be omitted if the s parameter is also omitted. The s parameter marks a subindex record which may aid user utilities to distinguish subindex records from data records.

Example:

```
CALL WRITMS (3,DATA,25,6,1)
```

Unconditionally rewrites in place of file TAPE3, starting at the address of the array named DATA, a 25-word record with an index number key of 6. The default value is taken for the s parameter.

READMS Subroutine

READMS (figure 7-28) transmits data from the file to central memory. CYBER Record Manager operates more efficiently if n is always a multiple of 64.

CALL WRITMS(u,fwa,n,k,r,s)

- u** Is the unit specifier.
- fwa** Is the name of an array in central memory or LCM (address of first word).
- n** Is the number of 60-bit words to be transferred.
- k** Is the record key; for number index, $1 \leq k \leq \text{lgth}-1$; for name index, $k = \text{any } 60\text{-bit quantity except } 0$.
- r** Specifies rewrite
 - 1 Rewrite in place if new record length does not exceed old record length; otherwise write at end-of-data.
 - 0 No rewrite; write at end-of-data (default value).
 - 1 Rewrite in place. Unconditional request; fatal error occurs if new record length exceeds old record length.
- s** Specifies subindex flag
 - 0 Do not write subindex marker flag in index control word (default value).
 - 1 Write subindex marker flag in index control word for this record.

Figure 7-27. WRITMS Call

CALL READMS(u,fwa,n,k)

- u** Is the unit specifier.
- fwa** Is the name of an array in central memory or LCM (address of first word).
- n** Is the number of 60-bit words to be transferred. If n is less than the record length, n words are transferred without diagnostic.
- k** Is the record key; for a number index, $1 \leq k \leq \text{lgth}-1$; for name index, $k = \text{any } 60\text{-bit quantity except } \pm 0$.

Figure 7-28. READMS Call

Example:

```
CALL READMS (3,MORDAT,25,2)
```

Reads the first 25 words of record 2 from unit 3 (TAPE3) into central memory starting at the address of the array MORDAT.

CLOSMS Subroutine

CLOSMS (figure 7-29) 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 run, to preserve a file created by an experimental job that might subsequently abort, or to perform other special functions.

CALL CLOSMS(u)

- u** Is the unit specifier.

Figure 7-29. CLOSMS Call

Since new data records can overwrite the old index, a file which has had new data records added is invalid unless the file is closed. (Under NOS/BE and SCOPE 2, permanent files must also be extended.) Jobs which might abort before closing the files should use RECOVER to recover and terminate normally (that is, STOP) to cause the files to be closed.

When using mass storage input/output subroutines in overlays or segments, care should be taken to close a file before program termination. If this is not possible, the mass storage input/output routines must reside in the (0,0) overlay or root segment. This can be done by including a call to an MSIO routine in the (0,0) overlay or root segment (the call need not be executed), or by using the LIBLOAD control statement.

Example:

```
CALL CLOSMS (2)
```

Closes the file TAPE2.

STINDX Subroutine

STINDX (figure 7-30) 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 subindex instead of the master index, STINDX is called to select the subindex as the current index. The STINDX call does not cause the subindex to be read or written; that task must be carried out by explicit READMS or WRITMS calls. It merely updates the internal description of the current index to the file.

Example:

```
DIMENSION SUBIX (10)  
CALL STINDX (3,SUBIX,10,0)
```

Selects a new index, SUBIX, for file TAPE3 with an index length of 10 (up to nine entries). The records referenced via this subindex use number keys.

Example:

```
DIMENSION MASTER (5)  
CALL STINDX (2,MASTER,5)
```

CALL STINDX(u,ix,lngth,t)

u	Is the unit specifier.
ix	Is the name of the array in central memory containing the subindex (first word address).
lngth	Is the length of subindex; for a number index $\text{lngth} \geq (\text{number of entries in subindex})+1$; for a name index $\text{lngth} \geq 2 * (\text{number of entries in subindex})+1$.
t	Is the type of subindex; if omitted, t is the same as the current index: 0 File has a number subindex. 1 File has a name subindex.

Figure 7-30. STINDX Call

Selects a new index, MASTER, from file TAPEZ with an index length of 5 and index type unchanged from the last index used.

Index Key Types

There are two types of index key, name and number. A name key can be any 60-bit quantity except +0 or -0. A number key must be a simple positive integer, greater than 0 and less than or equal to the length of the index in words, minus 1 word. The user selects the type of key by the t parameter of the OPENMS call. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. If the user should inadvertently attempt to reopen an existing file with an incorrect index type parameter, the job will be aborted. (This does not apply to subindexes chosen by STINDX calls; proper index type specification is the sole responsibility of the user.) In addition, key types cannot be mixed within a file. Violation of this restriction might result in destruction of a file.

The choice between name and number keys is left entirely to the user. The nature of the application may clearly dictate one type or the other. However, where possible, the number key type is preferable. Job execution will be faster and less central memory space will be required. Faster execution occurs because it is not necessary to search the index for a matching key entry (as is necessary when a name key is used). Space is saved due to the smaller index array length requirement.

Master Index

The master index type for a given file is selected by the t parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created; attempts to do so by reopening the file with the opposite type index are treated as fatal errors.

Subindex

The subindex type can be specified independently for each subindex. A different subindex name/number type can be specified by including the t parameter in the STINDX call. If t is omitted, the index type remains the same as the current index. Intervening calls which omit the t parameter do not change the most recent explicit type specification. The type remains in effect until changed by another STINDX call.

STINDX cannot change the type of an index which already exists on a file. The user must ensure that the t parameter in a call to an existing index agrees with the type of the index in the file. Correct subindex type specification is the responsibility of the user; no error message is issued.

Multilevel File Indexing

When a file is opened by an OPENMS call, the mass storage routines clear the array specified as the index area, and if the call is to an existing file, locates the file index and reads it into the array. This creates the initial or master index.

The user can create additional indexes (subindexes) by allocating additional index array areas, preparing the area for use as described below, and calling the STINDX subroutine to indicate to the mass storage routine the location, length and type of the subindex array. This process can be chained as many times as required, limited only by the amount of central memory space available. (Each active subindex requires an index array area.) The mass storage routine uses the subindex just as it uses the master index; no distinction is made.

A separate array space must be declared for each subindex that will be in active use. Inactive subindexes can, of course, be stored in the random file as additional data records.

The subindex is read from and written to the file by the standard READMS and WRITMS calls, since it is indistinguishable from any other data record. Although the master index array area is cleared by OPENMS when the file is opened, STINDX does not clear the subindex array area. The user must clear the subindex array to zeros. If an existing file is being manipulated and the subindex already exists on the file, the user must read the subindex from the file into the subindex array by a call to READMS before STINDX is called. STINDX then informs the mass storage routine to use this subindex as the current index. The first WRITMS to an existing file using a subindex must be preceded by a call to STINDX to inform the mass storage routine where to place the index control word entry before the write takes place.

If the user wishes to retain the subindex, it must be written to the file after the current index designation has been changed back to the master index, or to a higher level subindex by a call to STINDX.

The following examples illustrate the use of index key type. In figure 7-31, program MS1 creates a random file with a number index. The program MS2 adds two new records to the file created by MS1.


```

PROGRAM MS1(TAPE3)
C
C CREATE RANDOM FILE WITH NUMBER INDEX
  DIMENSION INDEX(11), DATA(25)
  CALL OPENMS(3, INDEX, 11, 0)
  DO 50 NRKEY = 1,10
    .
    . (Generate record in array named DATA.)
    .
    CALL WRITMS(3, DATA, 25, NRKEY)
50 CONTINUE
END

PROGRAM MS2(TAPE3)
C
C MODIFY RANDOM FILE CREATED BY PROGRAM MS1
C NOTE LARGER INDEX BUFFER TO ACCOMMODATE TWO NEW RECORDS.
  DIMENSION INDEX(13), DATA(25), MORDAT(40)
  CALL OPENMS(3, INDEX, 13, 0)
C READ 8TH RECORD FROM FILE TAPE3
  CALL READMS(3, DATA, 25, 8)
    .
    . (Modify array named DATA.)
    .
C WRITE MODIFIED ARRAY AS RECORD 8 AT END-OF-INFORMATION
C IN THE FILE
  CALL WRITMS(3, DATA, 25, 8)
C READ 6TH RECORD
  CALL READMS(3, DATA, 25, 6)
    .
    . (Modify array)
    .
C REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6
  CALL WRITMS(3, DATA, 25, 6, 1)
C READ 2ND RECORD INTO LONGER ARRAY AREA
  CALL READMS(3, MORDAT, 25, 2)
    .
    . (Add 15 new words to array named MORDAT.)
    .
C IN-PLACE REWRITE OF RECORD 2. IT WILL DEFAULT TO A NORMAL
C WRITE AT END-OF-INFORMATION SINCE THE NEW RECORD IS LONGER
C THAN THE OLD ONE, AND FILE SPACE IS THEREFORE UNAVAILABLE.
  CALL WRITMS(3, MORDAT, 40, 2, -1)
C READ THE 4TH AND 5TH RECORDS
  CALL READMS(3, DATA, 25, 4)
  CALL READMS(3, MORDAT, 25, 5)
    .
    . (Modify the arrays named DATA and MORDAT.)
    .
C WRITE THE ARRAYS TO THE FILE AS TWO NEW RECORDS
  CALL WRITMS(3, DATA, 25, 11)
  CALL WRITMS(3, MORDAT, 25, 12)
END

```

Figure 7-31. Random File With Number Index

Figure 7-32 shows how program MS3 creates a random file with a name index. The key names are RECORD1 through RECORD4.

Finally, in figure 7-33 program MS4 creates a subindexed file with a number index. The program uses four subindexes with nine records within each subindex, for a total of 36 records.

DEBUGGING ROUTINES

A number of miscellaneous routines for debugging are provided. See section 10 for the description of additional debugging capabilities.

```

PROGRAM MS3(TAPE7)
C
C CREATE A RANDOM FILE WITH NAME INDEX
  DIMENSION INDEX(9), ARRAY(15, 4)
  CHARACTER REC1, REC2
  DATA REC1, REC2/'RECORD1','RECORD2'/
  CALL OPENMS(7, INDEX, 9, 1)

      (Generate data in array area.)

C WRITE FOUR RECORDS TO THE FILE. NOTE THAT
C KEY NAMES ARE RECORD1, RECORD2, RECORD3, AND RECORD4
  CALL WRITMS(7, ARRAY(1, 1), 15, REC1)
  CALL WRITMS(7, ARRAY(1, 2), 15, REC2)
  CALL WRITMS(7, ARRAY(1, 3), 15, 'RECORD3')
  CALL WRITMS(7, ARRAY(1, 4), 15, 'RECORD4')
C CLOSE THE FILE
  CALL CLOSMS(7)
  END

```

Figure 7-32. Random File With Name Index

```

PROGRAM MS4(TAPE2)
C GENERATE SUBINDEXED FILE WITH NUMBER INDEX. FOUR
C SUBINDEXES WILL BE USED, WITH NINE DATA RECORDS IN
C EACH SUBINDEX, FOR A TOTAL OF 36 RECORDS
  DIMENSION MASTER(5), SUBIX(10), RECORD(50)
  CALL OPENMS(2, MASTER, 5, 0)
  DO 10 MAJOR = 1, 4
C CLEAR THE SUBINDEX AREA
  DO 20 I = 1, 10
    SUBIX(I) = 0
  20 CONTINUE
C CHANGE THE INDEX IN CURRENT USE TO SUBIX
  CALL STINDX(2, SUBIX, 10)
C GENERATE AND WRITE NINE RECORDS
  DO 30 MINOR = 1, 9

C WRITE A RECORD
  CALL WRITMS(2, RECORD, 50, MINOR)
  30 CONTINUE
C CHANGE BACK TO THE MASTER INDEX
  CALL STINDX(2, MASTER, 5)
C WRITE THE SUBINDEX TO THE FILE
  CALL WRITMS(2, SUBIX, 10, MAJOR, 0, 1)
  10 CONTINUE
C READ THE 5TH RECORD INDEXED UNDER THE 2ND SUBINDEX
  CALL READMS(2, SUBIX, 10, 2)
  CALL STINDX(2, SUBIX, 10)
  CALL READMS(2, RECORD, 50, 5)

      (Manipulate the selected record as desired.)

  END

```

Figure 7-33. Subindexed File With Number Index

DUMP AND PDUMP SUBROUTINES

DUMP (figure 7-34) and PDUMP (figure 7-35) dump central memory on the OUTPUT file in the indicated format, except when the ARG=FIXED parameter is specified.

CALL DUMP(a,b,f[,a,b,f] . . .)

- a Specifies the beginning of storage to be dumped.
- b Specifies the end of storage to be dumped.
- f Is a format indicator that can be:
 - 0 Produce octal dump.
 - 1 Produce real dump.
 - 2 Produce integer dump.
 - 3 Same as 0.

Figure 7-34. DUMP Call

CALL PDUMP(a,b,f[,a,b,f] . . .)

- a Specifies the beginning of storage to be dumped.
- b Specifies the end of storage to be dumped.
- f Is a format indicator that can be:
 - 0 Produce octal dump.
 - 1 Produce real dump.
 - 2 Produce integer dump.
 - 3 Same as 0.

Figure 7-35. PDUMP Call

PDUMP returns control to the calling program; DUMP terminates program execution. The maximum number of arguments for the triplet a, b, and f is 20.

For f values 0 through 3, a and b are the first and last words dumped. If 4 is added to any f value, the values of a and b are used as the addresses of the first and last words dumped within the job's field length. The LOCF function can be used to get addresses for the a and b parameters.

STRACE SUBROUTINE

STRACE (figure 7-36) provides traceback information from the subroutine calling STRACE back to the main program. Traceback information is written to the file OUTPUT.

CALL STRACE

Figure 7-36. STRACE Call

LEGVAR FUNCTION

LEGVAR (figure 7-37) checks the value of variable a and returns the result -1 if the variable is indefinite, +1 if out of range, and 0 otherwise. Variable a is type real; the result is type integer.

LEGVAR(a)

a Is a real variable.

Figure 7-37. LEGVAR Function

SYSTEM SUBROUTINE

SYSTEM (figure 7-38) enables the user to issue an execution-time error message.

CALL SYSTEM(ernum,mesg)

- ernum Is the integer error number. A decimal integer value from 0 through 9999. Error numbers used by the compiler retain the severity associated with them. Error numbers 51 (nonfatal) and 52 (fatal) are reserved for the user. If an error number greater than the highest number defined in appendix B is specified, 52 is substituted.
- mesg Is the error message. Entered as character constant with the first character used as a carriage control character and not printed.

Figure 7-38. SYSTEM Call

If error number zero is entered, the message is ignored, the output buffers are flushed, and control is returned to the calling program. Each line is printed unless the line limit of the OUTPUT file is exceeded, in which case the job is terminated.

Example:

```
CALL SYSTEM (3,'CHECK DATA')
```

SYSTEMC SUBROUTINE

SYSTEMC (figure 7-39) enables the user to replace the standard fatal/nonfatal error table with an enhanced error table. The error table contains specifications that regulate error processing. The error table is ignored for erroneous data input from a connected (terminal) file.

In the error table, the first error corresponds to error number 1, the second to error number 2, and so forth. Each entry has the format shown in figure 7-40.

CALL SYSTEMC(errnum,speclist)

errnum Is the integer number for which nonstandard recovery is to be implemented. The errnum ranges from 0 through 9999. If the errnum is greater than the highest internally defined error number, error number 52 is used. If the errnum is negative, SYSTEMC returns immediately with no action.

speclist Is an integer array containing error processing specification in consecutive locations:

Word 1 F/NF (1 = fatal, 0 = nonfatal).

Word 2 Print frequency.

Word 3 Frequency increment.

Word 4 Print limit.

Word 5 User-specified error recovery routine address.

Word 6 Maximum traceback limit applicable to all errors; this limit is 20 unless changed by a call to SYSTEMC.

Figure 7-39. SYSTEMC Call

In an overlay program, if SYSTEMC is not called in the (0,0) overlay, the routine might not be available to higher level overlays. When SYSTEMC is called from an overlay or segment, it must reside in the (0,0) overlay or the root segment.

A negative value for any word in the speclist indicates that the current value of that specification is not to be changed. A user-specified error recovery routine activated by a call to SYSTEMC can be canceled by a subsequent call with word 5 of the speclist set to zero.

If SYSTEMC has been called, an error summary is issued at job termination indicating the number of times each error occurred since the first call to SYSTEMC. Figure 7-41 shows a standard error recovery in a math library routine and how to suppress error message 115.

For an error detected by a routine in the math library, a user-supplied error recovery routine should be a function subprogram of the same type as the function detecting the error. For any other error, a user-supplied error recovery should be a subroutine subprogram. The error recovery subprogram must not invoke any code during recovery that was involved with the error being handled. For example, if an I/O related error is being processed, I/O operations of any type must not be done, unless the recovery routine is intended to terminate execution.

When an error previously referenced by a SYSTEMC call is detected, the following sequence of operations is initiated:

1. Diagnostic and traceback information is printed in accordance with the specification in the pertinent error table entry. The traceback information is terminated for any of the following conditions:

Calling routine is a program.

Maximum traceback limit is reached.

No traceback information is supplied.

2. If the SYSTEMC call references a user-specified error recovery routine address, SYSTEMC, FORSYS-, and the routine detecting the error are delinked from the calling chain, and the user-supplied error recovery routine is entered.
3. If the error is nonfatal, control returns to the routine that called the routine detecting the error. An error summary is printed at job termination.
4. If the error is fatal, all output buffers are flushed, an error summary is printed, and the job is terminated.

If a nonstandard recovery address is specified in the SYSTEMC call, the following information in table 7-4 is available to the user recovery routine.

LIMERR SUBROUTINE AND NUMERR FUNCTION

The LIMERR subroutine (figure 7-42) and the NUMERR function (figure 7-43) enable the user to input data without the risk of termination when improper data is encountered. When LIMERR is used, the program does not terminate when data errors are encountered until the number of errors occurring after the call exceeds the value of lim. The NUMERR function returns the number of errors since the last LIMERR call. The result is integer.

LIMERR can be used to inhibit job termination when data is being input with a formatted, NAMELIST, or list directed read or with DECODE statements. It operates only when data is encountered that would ordinarily cause job termination under error number 78 ("ILLEGAL DATA IN FIELD") or error number 79 ("DATA OVERFLOW"). LIMERR has no effect on the processing of errors in data input from a connected (terminal) file.

LIMERR initializes an error count and specifies a maximum limit (lim) on the number of data errors allowed before termination. LIMERR continues in effect for all subsequent READ statements until the limit is reached. LIMERR can be reactivated with another call, which will reinitialize the error count location and reset the limit. A LIMERR call with lim specified as zero nullifies a previous call; improper data will then result in job termination as usual.

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	By default, print frequency value is 0. If the value is changed to n by a call to SYSTEMC, diagnostic and traceback information is listed every nth time until the print limit is reached.					
frequency increment	By default, frequency increment value is 1. This specification can be changed by a call to SYSTEMC if the call specifies print frequency as 0. When frequency increment is 0, diagnostic and traceback information is not listed; when it is 1, such information is listed until the print limit is reached; when the frequency increment is n > 1, such information is listed only the first n times unless the print limit is reached first.					
print limit	By default, print limit value is 10. It can be changed by a call to SYSTEMC.					
detection total	Detection total is a running count of the number of times an error occurs. The final value is reported in the error summary issued at end-of-job if SYSTEMC is called during execution.					
F/NF	This bit specifies the severity of the error: 1 indicates a fatal error; 0, nonfatal. The severities of system defined errors are given in appendix B. All errors defined by the user with these numbers in a call to SYSTEMC retain the specified severity. The severity of any error can be changed by a call to SYSTEMC, however.					
A/NA	The A/NA bit is ignored unless a nonstandard recovery address is specified; it can be set only during assembly of SYSTEMC. When this bit is set, the address in an auxiliary table is passed in the third word of the secondary argument list to the recovery routine. Each word in the auxiliary table must have the error number in its upper 10 bits, so that the address of the first error number match is passed to the recovery routine. An entry in the auxiliary table for an error is not limited to any specific number of words.					
user-specified recovery address	This address is specified in a call to SYSTEMC.					

Figure 7-40. Error Table Entry

```

PROGRAM EXPECT
DIMENSION IRAY(6)
DATA IRAY/6* -0/
C SET PRINT LIMIT TO ZERO
IRAY(4) = 0
X = EXP(800.0)
X = EXP(-800.0)
C CALL SYSTEMC TO INHIBIT PRINTING OF ERROR 115
C AND START ERROR SUMMARY ACCUMULATION
CALL SYSTEMC(115,IRAY)
PRINT *
PRINT *, '*****SYSTEMC IS CALLED TO SUPPRESS '//
+ 'PRINTING OF ERROR 115'
X = EXP(800.0)
X = EXP(-800.0)
PRINT *
PRINT *, '*****ERROR 115 DETECTED BUT NOT PRINTED'
END

```

Figure 7-41. Suppressing an Error Message

TABLE 7-4. INFORMATION AVAILABLE TO ERROR RECOVERY ROUTINE

Register	Contents
A1	Address of argument list passed to routine detecting the errors detected by a math library routine. Address of the FIT for error 103. Undefined for all other errors.
X1	Address of the first argument in the list for errors detected by a math library routine. Undefined for all other errors.
A0	Address of argument list of routine that called the routine detecting the error.
B1	Address of a secondary argument list containing, in successive words: Error number associated with this error. Address of message associated with this error. Address within auxiliary table if A/NA bit set; otherwise 0. In upper 30 bits, instruction consisting of RJ to SYSERR.j; in lower 30 bits, address of trace-back information for routine detecting the error. Information in the secondary argument list is not available to user-supplied error recovery routines coded in FORTRAN.
A2	Address of error table entry for this error.
X2	Contents of error table entry for this error.

```
CALL LIMERR(lim)

lim    Is the limit for the number of
       errors.
```

Figure 7-42. LIMERR Call

```
NUMERR()
```

Figure 7-43. NUMERR Function

When improper data is encountered in a formatted or NAMELIST read (or in a DECODE statement) with LIMERR in effect, the bad data field is bypassed, and processing continues at the next field. When improper data is encountered in a list directed read, control moves to the statement immediately following the READ statement.

NUMERR returns the number of errors since the last LIMERR call. The previous error count is lost when LIMERR is called, and the count is reinitialized to zero.

Figure 7-44 illustrates the use of LIMERR and NUMERR to suppress normal fatal termination when large sets of data are being processed. When LIMERR is called, a limit of 200 errors is established. The number of errors is reset to zero. After ARAY is read, NUMERR() is checked. If errors occur, the following statements are not processed and a branch is made to statement 500. Had LIMERR not been called, fatal errors would have terminated the program before the branch to statement 500. At statement 500, LIMERR once more initializes the error count, and execution continues.

```

125    CALL LIMERR(200)
      READ(1, 125) (ARAY(I), I = 1, 1500)
      FORMAT (3F10.5, E10.1)
      IF (NUMERR( ) .GT. 0) GO TO 500
      .
      .
500    CONTINUE
      CALL LIMERR(200)
      READ(1, 125) (BRAY(I), I = 1, 1500)
      IF (NUMERR( ) .GT. 0) GO TO 600
      .
      .
600    CONTINUE
      CALL LIMERR(100)
      READ(1, 230) (LRAY(I), I = 1, 500)
      WRITE(2, *) NUMERR( )
      READ(4, 127) (MRAY(I), I = 1, 500)
      WRITE(2, *) NUMERR( )
      .
      .
```

Figure 7-44. Suppressing Fatal Termination

COLLATING SEQUENCE CONTROL SUBPROGRAMS

Character relational expressions are evaluated according to a collating sequence, determined by a collating weight table. A weight table is a 1-dimensional array. The upper bound of the array is 63 for installations using either the 63- or 64-character set. The lower bound of the array is zero for installations using the 64-character set; the lower bound of the array is 1 for installations using the 63-character set. The value of element *i* of the weight table is the collating weight for the character with the character code of *i*. The character codes and graphic representations for all characters supported by the processor are given in

table 7-5. If c(i) and c(j) are characters and i and j are their respective collating weights, then c(i).op.c(j) has the value .TRUE. if and only if i.op.j has the value .TRUE., where .op. is any of the relational operators.

The value of a weight table element does not have to be unique within the table; that is, several characters can have the same collating weight.

Collation can be directed by the fixed collating weight table or by the user-specified collating weight table. The fixed table is predefined as the display code weight table and cannot be modified; the user-specified weight table is predefined as the ASCII6 weight table and can be accessed and modified by the program. The CS parameter on the FTN5 control statement and the C\$ COLLATE compiler directive determine whether the fixed or user-specified table controls current collation. The predefined weight tables appear in table 7-5.

The ASCII collating sequence used by the intrinsic functions LGE, LGT, LLE, and LLT is independent of both the fixed and user-specified weight tables and is therefore unaffected by either the compiler-call statement option or the C\$ COLLATE directive. The

intrinsic function INDEX does not use either collating weight table.

A program can access or modify the user-specified weight table by using the procedures COLSEQ, WTSET or CSOWN.

COLSEQ SUBROUTINE

COLSEQ (figure 7-45) selects a processor-defined user-specified weight table.

The collating sequences that can be selected are:

- ASCII6
- COBOL6
- DISPLAY
- STANDARD

If STANDARD is specified, COBOL6 values are used when the operating system is using the CDC graphic set; and ASCII6 values are used when the operating system is using the CDC ASCII subset.

TABLE 7.5 COLLATING WEIGHT TABLES

CDC Graphic	Octal Character Code	Decimal Weights			CDC Graphic	Octal Character Code	Decimal Weights		
		ASCII6	COBOL6	Display			ASCII6	COBOL6	Display
:	00 [†]	26	53	0	5	40	21	59	32
A	01	33	25	1	6	41	22	60	33
B	02	34	26	2	7	42	23	61	34
C	03	35	27	3	8	43	24	62	35
D	04	36	28	4	9	44	25	63	36
E	05	37	29	5	+	45	11	15	37
F	06	38	30	6	-	46	13	18	38
G	07	39	31	7	*	47	10	17	39
H	10	40	32	8	/	50	15	19	40
I	11	41	33	9	(51	8	21	41
J	12	42	35	10)	52	9	13	42
K	13	43	36	11	\$	53	4	16	43
L	14	44	37	12	=	54	29	22	44
M	15	45	38	13	blank	55	0	0	45
N	16	46	39	14	, (comma)	56	12	20	46
O	17	47	40	15	.(period)	57	14	12	47
P	20	48	41	16	≡	60	3	5	48
Q	21	49	42	17	[61	59	3	49
R	22	50	43	18]	62	61	44	50
S	23	51	45	19	%	63 [†]	5	2	51
T	24	52	46	20	≠	64	2	23	52
U	25	53	47	21	↗	65	63	4	53
V	26	54	48	22	√	66	1	34	54
W	27	55	49	23	^	67	6	6	55
X	30	56	50	24	↑	70	7	7	56
Y	31	57	51	25	↓	71	31	8	57
Z	32	58	52	26	<	72	28	24	58
0	33	16	54	27	>	73	30	9	29
1	34	17	55	28	≤	74	32	1	60
2	35	18	56	29	≥	75	60	10	61
3	36	19	57	30	↵	76	62	11	62
4	37	20	58	31	;(semicolon)	77	27	14	63

[†]In installations using the 63-graphic set, the octal character code 00 does not exist, and the weights in the ASCII6 and COBOL6 tables for the octal character code 63 assume the corresponding weights from character code 00.

CALL COLSEQ(a)

- a Is a character expression whose value when any trailing blanks are removed is one of the values ASCII6, COBOL6, DISPLAY, or STANDARD.

Figure 7-45. COLSEQ Call

Example:

To select a collating sequence the same as COBOL6, with the exception that characters \$ and . sort equally ('\$.EQ. '.), the following can be used:

```
CALL COLSEQ ('COBOL6')
CALL WTSET ('$.EQ. '.)
```

The user-specified weight table is initialized to the COBOL standard collating sequence, and the entry for the character code \$ (53 octal) is reset to 12 decimal (the value of the weight table indexed by 57 octal ('.')).

WTSET SUBROUTINE

WTSET (figure 7-46) modifies the user-specified weight table.

CALL WTSET(ind,wt)

- ind Is an integer expression with a value between 0 and 63 for the 64 character set and between 1 and 63 for the 63 character set.
- wt Is an integer expression with a value between 0 and 63.

Figure 7-46. WTSET Call

If ind is a character expression with value c, the element of the weight table indexed by the character code of c is replaced with wt; if ind is an integer expression, the weight table element indexed by ind is replaced with wt.

CSOWN SUBROUTINE

CSOWN (figure 7-47) specifies a partial collating sequence.

CALL CSOWN(str)

- str Is a character expression of length 1 through S inclusive, with a value of the form:

$c(1)c(2)c(3) \dots c(n)$, $1 \leq n \leq S$

No c(i) can equal c(j) unless i equals j.

Figure 7-47. CSOWN Call

CSOWN explicitly defines the weight table elements for the n characters and then sets all other elements to zero. For i from 1 to n, the element p(i) of the user-specified weight table is set to i-1, where p(k) is the character code of c(k).

STATIC LOADING OPTION

NOTE

Refer to appendix G for recommendations on the use of the STATIC option.

Modules required by a program can be loaded statically (when Common Memory Manager is not used) or dynamically (CMM). Dynamic loading is the preferred method, because it gives the more efficient use of memory.

However, dynamic loading is not compatible with all types of programs. For example, a program that overindexes blank common must use static loading to prevent memory conflicts. Also, a program involving time-critical real-time application might find the cost of the Fast Dynamic Loader to be prohibitive. In these instances, the STATIC option should be used.

The STATIC option is available on NOS, NOS/BE, and SCOPE 2. The use of the terms capsule and CYBER Record Manager does not apply to SCOPE 2. However, calls to the STLxxx subroutines are required to inhibit Common Memory Manager operation on SCOPE 2.

To use the STATIC option, the user must:

Compile the main program with the STATIC option specified on the PTNS control statement. (Subroutines can also be compiled by using this option, but it is not essential.)

Specify all files to be used on the PROGRAM statement.

Write the program with calls to all STLxxx routines appropriate for the types of input/output in the program.

The initial load of the program requires the specification of which CYBER Record Manager capsules are to be included. This specification is accomplished through:

A reference to one of the STLxxx subroutines supplied by the FORTRAN 5 library.

The use of the FILE statement USE parameter in conjunction with a LDSET(STAT=logical file name) directive. If INQUIRE or OPEN statements are in the program, this specification cannot be used. This specification also requires at least one call to a STLxxx routine; otherwise, CMM will be loaded.

The use of LDSET(USE=STLxxx/STLyyy...) to include required capsules. This specification is included through batch mode or a procedure file. It can be interactively used under NOS only.

The STLxxx subroutines supply only those capsules required for the default file attributes of a particular input/output operation.

During program execution, each STLxxx subroutine referenced immediately returns control upon invocation. References to STLxxx subroutines are made solely for the load-time effect of the LDSET directives contained within them. The execution of these subroutines is not required.

Necessary FORTRAN error processing capsules are also loaded by any reference to a STLxxx subroutine, which in turn force loads the STLERR subroutine.

In an overlay or segmented environment, all STLxxx calls should be made within the (0,0) overlay or root segment.

Table 7-6 describes the STLxxx subroutine names, the input/output operations requiring them, and the supported file organization (FO), block type (BT), and record type (RT) applicable to each routine. All STLxxx subroutine names are described to provide complete documentation, but only STLERR need be called for SCOPE 2.

TABLE 7-6. STATIC CAPSULE LOADING ROUTINES

Subroutine Name	Required by	Supports		
		FO	BT	RT
CALL STLBK (Backspace)	Sequential BACKSPACE	SQ	I C C	W Z S
CALL STLENF (End file)	Sequential ENDFILE	SQ	I C C	W Z S
CALL STLERR (Error recovery)	All static jobs to omit CMM	Force load FORTRAN error recovery capsule		
CALL STLIBI (Input Binary)	Sequential Unformatted READ	SQ	I	W
CALL STLIBU (Input Buffered)	Sequential BUFFER IN	SQ	C	S
CALL STLICO (Input Coded)	Sequential Formatted, List Directed and Namelist Input	SQ	C	Z
CALL STLIDB (Input Direct Access Binary)	Direct Access Unformatted Input	WA	C	U
CALL STLIDC (Input Direct Access Coded)	Direct Access Formatted Input	WA	C	U
CALL STLINQ (Inquire)	Explicit INQUIRE statements	Force load INQCAP from FTN5LIB		
CALL STLOBI (Output Binary)	Sequential Unformatted Output	SQ	I	W
CALL STLOBU (Output Buffered)	Sequential BUFFER OUT	SQ	C	S
CALL STLOCO (Output Coded)	Sequential Formatted, List Directed and Namelist output	SQ	C	Z
CALL STLODB (Output Direct Access Binary)	Direct Access Unformatted Output	WA	C	U
CALL STLODC (Output Direct Access Coded)	Direct Access Formatted Output	WA	C	U
CALL STLOPE (Open)	Explicit OPEN Statements	Force load OPECAP from FTN5LIB		
CALL STLREW (Rewind)	Sequential REWIND	SQ	I C C	W Z S
CALL STLRMS (Random Access Mass Storage)	Random Access I/O Routines (OPENMS, etc)	WA	C	U

FORTRAN programmer can access several different system and product capabilities through calls to interface routines. Some of these capabilities, such as the data management products, can be used only if the product is available at the installation; others, such as the direct calls to CYBER Record Manager, are always available. The capabilities described below are not necessarily part of the standard FORTRAN product and might not be available at all installations.

Calls include interfaces to:

Permanent File Commands

Under NOS, permanent files can be manipulated within a program as they can from terminal or batch job commands.

CYBER Record Manager (CRM)

CYBER Record Manager includes Basic Access Methods (BAM) and Advanced Access Methods (AAM). BAM can be called directly to create files with sequential or word addressable file organization. AAM can be called directly to create files with indexed sequential, direct access, or actual key file organization. AAM file records are accessed randomly by primary or alternate keys. CRM also includes utility routines.

Sort/Merge

Records and files can be sorted or merged. A program can interface with Sort/Merge 4 under NOS and NOS/BE, with the similar Sort/Merge 1 under SCOPE 2, and with Sort/Merge 5 under NOS 2 and NOS/BE.

Common Memory Manager (CMM)

Common Memory Manager allows a program to manage memory directly during execution. By allocating arrays to memory blocks, a program can better utilize field length under special circumstances.

COMPASS Assembly Language Subprograms

Subprograms written in the COMPASS assembly language can be assembled at the same time as FORTRAN language program units are being compiled. The COMPASS module must use the FORTRAN calling sequence described in this section.

8-Bit Subroutines

Data in ASCII and IBM 8-bit format, rather than in the 6-bit format used for internal CDC files, can be converted and manipulated through these subroutines.

CYBER Database Control System (CDCS)

CDCS is a data management system that controls, monitors, and interprets data base requests from a FORTRAN application program.

Information Management Facility (IMF)

IMF is an information management system that describes, organizes, and controls information. A FORTRAN program accesses the system by means of a implementation language statements that are precompiled.

Queued Terminal Record Manager (QTRM)

QTRM is a simplified means of accessing the AIP portion of the Network Access Method. It allows one program to communicate with other terminals and programs.

Transaction Facility (TAF)

TAF is an application program that allows processing by individual transactions rather than by complete programs. It uses the Network Access Method for its terminal input/output.

PERMANENT FILE INTERFACE

The subroutine named PF allows a program to issue many of the permanent file commands that can be issued from a terminal or from a batch job. It also allows the program to take control of error processing.

The PF subroutine is available under NOS only.

The interface with the NOS permanent file system is established through a single subroutine. The first parameter of the call specifies the particular permanent file operation. The possible operations are:

DEFINE

Create an empty direct access permanent file.

ATTACH

Assign a direct access permanent file to the job.

SAVE

Retain a copy of a local file as an indirect access permanent file.

GET

Retrieve a copy of of an indirect access permanent file for use as a local file.

APPEND

Add information to the end of an existing indirect access permanent file without retrieving the file. APPEND cannot append data to direct access files.

REPLACE

Purge an indirect access permanent file, and replace it with the copy of a local file on mass storage.

PERMIT

Specify that another user can access this private file. PERMIT also changes the mode in which another user can access a semi-private file.

PURGE

Remove the named files from the permanent file device.

CHANGE

Change certain characteristics of a permanent file. The file need not be assigned to the job.

Before issuing a call for the permanent file commands SAVE, REPLACE, or APPEND, the program should execute a REWIND, ENDFILE, or CALL SYSTEM(0,0) statement for the file. These statements ensure that the buffer is flushed and the entire local file is retained as a permanent file.

PF CALL

The PF subroutine call has two types of parameters: the mandatory ones that must appear in the order shown, and the optional ones that can appear in any order. There are three formats for the PF subroutine, depending on the operation requested.

For the APPEND, ATTACH, DEFINE, GET, REPLACE, and SAVE operations, the format of the PF subroutine is:

```
CALL PF ('command', 'lfn', 'pfn'  
[, 'keyword', 'value', ..., 'keyword', 'value'])
```

For example, to use permanent file MYFLE in the program as the file associated with local file SOURCE:

```
CALL PF ('GET', 'SOURCE', 'MYFLE', 'UN',  
      'A23456', 'PW', 'XX')
```

The first three parameters (command, lfn, and pfn) are required:

command

Operation to be performed, specified as a string of characters. It must be one of the permanent file commands listed above.

lfn

The name of the local file to be processed. It is specified as a string of 1 through 7 characters, as a variable, or as an integer representing the FORTRAN unit number. If lfn is an integer, the lfn used is TAPEnnn, where nnn is the integer value converted to characters. If lfn is all blanks or zero, lfn is assumed to be the same as pfn.

pfn

The name of the permanent file to be processed, specified as either a string of 1 through 7 characters or a variable. If pfn is all blanks or 0, pfn is assumed to be the same as lfn. If no optional parameters are specified, pfn can be omitted and pfn is assumed to be the same as lfn.

For the CHANGE operation, the format of the PF subroutine is:

```
CALL PF ('change', 'nfn', 'ofn'  
[, 'keyword', 'value', ..., 'keyword', 'value'])
```

The first three parameters (change, nfn, ofn) are required. The nfn specifies the new permanent file name. The nfn must be specified even if the permanent file name is not changing, in which case nfn is the same as ofn. The ofn specifies the old permanent file name.

For the PERMIT and PURGE operations, the format of the PF subroutine is:

```
CALL PF ('command', 'pfn',  
[, 'keyword', 'value', ..., 'keyword', 'value'])
```

The first two parameters (command and pfn) are required.

The options for a permanent file operation can appear in any order after the required parameters. The options are specified by paired keyword-and-value form, so that unwanted options need not be addressed:

The keyword is the first parameter of each pair. It must be a character constant, character variable, or character expression.

The value is the second parameter of the pair. A comma and the value parameter must immediately follow the keyword parameter to specify the value for the keyword. All values must be specified as a character string or a variable containing a left justified character string; exceptions are the return code variables as noted below.

Table 8-1 lists the options that can be specified for each of the commands. If an optional parameter is not specified in the subroutine call, the default is the same as the corresponding parameter for the NOS permanent file command. (See the NOS reference manuals for details of permanent file operations.) However, NA has a different result in the subroutine call, and several other parameters have no parallel in the control statements of a job.

If the PF subroutine is called twice in the same program unit with a different number of parameters, an informative diagnostic is issued by the compiler.

Meanings of the options of table 8-1 are in table 8-2.

ERROR PROCESSING

The result of an error depends on the source of the error and the processing selected by the NA, RC/RRC, and UP keywords. RRC has the same meaning as RC except the value returned is real.

TABLE 8-1. PF SUBROUTINE PARAMETERS

Parameter	Append	Attach	Change	Define	Get	Permit	Purge	Replace	Save
lfn	x	x	nfn	x	x	-	-	x	x
pfm	x	x	ofn	x	x	x	x	x	x
'UN', 'un'	x	x	-	-	x	x	x	x	-
'PW', 'psivd'	x	x	x	x	x	-	x	x	x
'PN', 'pn'	x	x	x	x	x	x	x	x	x
'R', 'r'	x	x	x	x	x	x	x	x	x
'RT', 'ignr'	-	x	-	-	-	-	-	-	-
'S', 's'	-	-	'CE'	'MR'	-	-	-	-	-
'M', 'm'	-	x	x	x	-	-	-	-	-
'SR', 'sr'	-	-	x	x	-	-	-	-	-
'SS', 'ss'	-	-	x	-	-	-	-	x	x
'BR', 'br'	-	-	x	x	-	-	-	-	x
'PR', 'pr'	-	-	x	x	-	-	-	-	x
'CT', 'ct'	-	-	x	x	-	-	-	-	x
'NA', 'ignr'	x	x	x	x	x	x	x	x	x
'UP', 'ignr'	x	x	x	x	x	x	x	x	x
'UCW', ucw	-	x	-	x	x	-	-	-	x
'RC', r or 'RRC', rc	x	x	x	x	x	x	x	x	x

Legend:

- x Command parameter is valid for this operation
- Not applicable or has no effect
- nfn New permanent file name
- ofn Old permanent file name
- 'CE' This value clears the error file code
- 'MR' This value causes master device residency

These keywords work in combination:

Neither NA nor RC/RRC is specified-

If the operation is successful, the program proceeds as usual. If the operation fails, an error message is written to the dayfile (or terminal) and the program aborts.

Only RC/RRC is specified-

If the operation fails, the appropriate return code is returned to the RC/RRC variable.

If the file is unavailable on an ATTACH operation, (for example, some other job is updating the file), the job is swapped out and swapped back in when the file is available.

Both RC/RRC and NA are specified-

If the operation fails, the appropriate return code is returned to the RC/RRC variable. Control then returns to the program.

TABLE 8-2. MEANINGS OF PF COMMAND OPTIONS

Keyword	Value	Meaning
'BR'	'br'	Backup copy selection: 'Y' Keep a tape backup copy even if a copy of the file exists on the MSF device 'MD' Do not keep a tape backup copy if a copy of the file exists on the MSF device 'N' Do not keep any backup copy
'CT'	'ct'	File access category: 'P' or 'PRIVATE' Private file 'S' or 'SPRIV' Semiprivate file 'PU' or 'PUBLIC' Public file
'M'	'm'	Permission granted to alternate users for semiprivate and public files: 'WRITE' or 'W' Read, execute, lengthen, rewrite, or shorten, the file 'READ' or 'R' Read or execute the file 'EXECUTE' or 'E' Execute the file 'READAP' or 'RA' Read or execute the file 'READMD' or 'RM' Read or execute the file 'APPEND' or 'A' Read, execute, or lengthen the file 'MODIFY' or 'M' Read, execute, lengthen, or rewrite the file 'NULL' or 'N' No access is allowed
'NA'	'ig'	Select no-abort. The value is ignored by PF. NA processing is not the same as for the NOS command. See Error Processing below.
'PW'	'pswd'	A 1 through 7 character password.
'PN'	'pknm'	A 1 through 7 character pack name used in conjunction with the R parameter to identify the auxiliary device to be accessed in the permanent file operation.
'PR'	'pr'	Preferred residence: 'M' Copy this direct access file to the MSF device between accesses 'N' No preference
'R'	'r'	Device type on which the permanent file resides: 'DE'; 'DI'; 'DJ'; 'DK'; 'DL'; 'DM'; 'DQ'; or 'DP'
'RC' and 'RRC'	rc	Variable to which an error code is returned. RC specifies an integer variable; RRC a real variable. See Error Processing below.
'RT'	'ig'	Select real-time processing. The value is ignored by PF.
'S'	's'	A 1 through 7 character decimal number of PRUs requested for the file. Leading blanks can appear before the first digit of the number.

TABLE 8-2. MEANINGS OF PF COMMAND OPTIONS (Contd)

Keyword	Value	Meaning
'SR'	'sr'	Select special permanent file function: 'CE' Clear the file error code on a CHANGE operation 'MD' Cause master device residency on a DEFINE operation
'SS'	'ss'	Identify the time-sharing subsystem to be associated with this indirect file: 'NULL' No subsystem 'BASIC' BASIC subsystem 'FORTRAN' FORTRAN subsystem 'FTNTS' FORTRAN Extended 4 subsystem 'EXECUTE' Execute subsystem 'BATCH' Batch subsystem
'UCW'	ucw	Integer variable containing user control information to be stored in the permanent file catalog on a SAVE or DEFINE operation. On an ATTACH operation, the contents of this catalog word are returned to the variable. User control information can be any data that is to be associated with the file.
'UN'	'un'	A 1 through 7 character user number.
'UP'	'ig'	Select program control when permanent file device is unavailable. The value is ignored by PF. NA or RC/RRC must also be selected. See Error Processing below.

Codes that are returned to RC or RRC are:

- >0 Error code as specified in the appropriate NOS reference manual.
- 0 Successful completion of the operation.
- 1 Error in the parameters of the PF subroutine call.

The UP keyword establishes processing when the permanent file device is unavailable. If neither NA nor RC/RRC is selected, the program aborts when the device is not available. If one of these keywords is specified, no abort occurs: rather, the system operator receives a message and the program is swapped out until the device becomes available. When UP is specified in addition to NA or RC/RRC, control immediately returns to the program when the device is unavailable with the appropriate permanent file manager code in any RC/RRC variable.

If the value returned to RC/RRC is -1, an error message appears on the dayfile. These messages are:

PF NO.-1 INVALID PARAMETER -- UNPAIRED

The value is missing from the keyword-and-value pair.

PF NO.-1 INVALID PARAMETER -- xxxxxxxx

The keyword or value identified is not valid.

PF NO.-1 xx cccccccc

xx is the error code, and ccc...ccc the message returned by the permanent file manager.

CYBER RECORD MANAGER (CRM)

CYBER Record Manager has two uses. It is the means by which input/output is implemented within FORTRAN and other standard products. Also, it is the product that can be called directly from a FORTRAN program to read and write files with sequential (FO=SQ), word addressable (FO=WA), indexed sequential (FO=IS), direct access (FO=DA), or actual-key (FO=AK) file structures.

The direct calls of CRM are available under NOS and NOS/BE only. Direct calls are not compatible with the ARG=FIXED option of the FORTRAN compiler control statement.

CRM has two parts: BAM (Basic Access Methods) consists of the sequential and word addressable file processors; AAM (Advanced Access Methods) consists of the indexed sequential, direct access, and actual-key file processors. AAM includes the MIP (Multiple Index Processor) that allows a record to be retrieved by alternate key values as well as by primary key.

FILE INFORMATION TABLE (FIT)

The file information table is the means of communication between a program and CRM. Each file to be accessed directly must have an associated FIT. The program must dimension an array of 35-words, then call the FILExx routine to construct the FIT in that array. All subsequent direct calls to CRM identify the file through the FIT array name.

Fields in the FIT are identified by keyword mnemonics. The minimum number of fields that must be set depends on the file organization, record type, and processing desired. Fields can be set in several ways:

When the FILExx call is executed, where xx is the mnemonic for the file organization, fields specified in the call are set to the indicated values. For instance, CALL FILESQ (MYFIT, LFN, MYFILE) establishes the correspondence between the table in array MYFIT and the logical file name MYFILE.

The STOREF call sets an individual field to the specified value. STOREF can be called before or after a file is opened.

When OPENM executes to open a file, field values are obtained from several sources. Fields that have not been referenced on FILExx or STOREF receive default values. Some default values are triggered by other fields: for instance, RT=W forces BT=I. If the file has AAM organization, some fields are set from the file statistics table that is a permanent part of every AAM file. (The file statistics table contains information that cannot change during the life of the file, such as the location of the key within a record or the name of a hashing routine.) Then, any fields specified on a FILE control statement (figure 8-1) are set in the FIT. The FILE control statement overrides any other existing value.

```
FILE, lfn, field=value, . . . field=value,
```

lfn Logical file name of file referenced in the program by means of the FIT array name.

field Mnemonic for a FIT field. Most FIT fields can be referenced: the exceptions are the values that refer to a program location.

value Value for the field.

Figure 8-1. FILE Control Statement

As a call to a file manipulation routine executes, fields are set according to parameters in that call. Then, CRM takes action on the basis of the FIT. Better programming practice is to set fields directly.

The last value set in a field governs any operation using that field.

Table 8-3 shows the FIT fields for processing all file organizations.

CALL SYNTAX

Table 8-4 shows the direct calls to CYBER Record Manager. A parameter is explained only when it does not have an equivalent FIT field.

In the FILExx call that establishes the FIT, parameters for the fields and values are paired. (The file organization FO field will be set according to xx of the call.) A particular FIT field is identified by a hollerith constant.

In all other calls, parameters are positional. Most fields required for processing can be set directly in the FIT by FILExx or STOREF. Subsequent file processing calls then can be limited to a single parameter that identifies the array in which the FIT exists.

The preferred processing method is to set the FIT field before execution of a call for file access.

The current contents of a FIT field can be retrieved through the IFETCH function. The format of the value returned is affected by the type of information in the field:

Fields specified as integers are returned as integers.

Length fields return an integer that indicates the number of 6-bit characters in the value. The BFS value for buffer size is returned as a word count, however.

If a symbolic value is stored as a 1-bit field, the value is returned as a positive integer when the bit is 0. When the bit is set to 1, a negative integer is returned. For example, the NDX field which indicates an index file operation for an alternate key file is stored as "YES" but fetched as a positive integer.

Symbolic field values that are stored as more than one bit are returned as positive integers. For example, the LFN field is returned left-justified and zero filled in the integer variable specified in IFETCH.

Some values are returned as octal codes. For example, FP which indicates file position returns 100 octal at end of information.

If a program that accesses an actual key file is compiled with OPT=2 or OPT=3, ka should be in a common block.

FILE PROCESSING

Once the FIT is established, the file can be opened for processing. The program is responsible for record access.

A working storage area is required for record access. The program must dimension an INTEGER array or word-aligned CHARACTER variable and set the WSA field. CRM takes a record from the WSA location for a write operation; it returns a record to that location for a read operation. WSA can change for each file access.

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
ASCII	ASCII character set	0,1,2	0	Yes	F0=SQ only. Used for terminal files on NOS/BE only
BBH	Buffer below highest high address	YES, NO	NO	Yes	BAM only
BCK	Block checksums	YES, NO	NO	No	AAM only. IFETCH return: 0 = NO 1 = YES
BFS	Buffer length in words	0 thru 131071	Provided by CRM	Yes	AAM always ensures minimum needed. Must be greater than MBL and PRU size from BAM
BN	Block number				F0=SQ fetch only
BT	Block type	I,C,K,E	I	No	F0=SQ only
BZF	Input/output status word location				Fetch only
CF	Close flag	R, N, U, RET, DET, DIS	R		
CL	Trailer count field length	1 thru 6	0	No	RT=T only
CM	Conversion mode	YES, NO	NO	No	Must be NO for RT=W
CNF	Connect file flag	YES, NO	NO	No	F0=SQ for terminal files
CP	Trailer count field start	0 thru BAM 1310710 or AAM 8180	0	No	RT=T only
CPA	Compression routine address	Routine number or name	0	Yes	AAM only
CI	Binary length field	YES, NO	NO	No	RT=D/T only
DCA	Decompression routine address	Routine name	0	Yes	Required if CPA is routine name
DCT	Display code to collating sequence table	Table location	Standard CDC	No	F0=IS KT=S only
DFC	Dayfile control	0,1,2,3	0	Yes	

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
DP	Data block padding percentage	0 thru 99	0	No	FO-IS/AK only
DX	End-of-data exit	Routine name	0	Yes	
ECT	Trivial error count				
EFC	Error file control	0,1,2,3	0	Yes	
EMK	Embedded primary key	YES, NO	NO	No	AAM only
EO	Error option for parity errors	T, D, A, TD, DD, AD	T	Yes	BAM only IFETCH return: 0=T 4=TD 1=D 5=DD 2=A 6=AD
EOIWA	End-of-information word address				FO-WA fetch only
ERL	Trivial error limit	0 thru 511	0	Yes	0 allows indefinite number of errors
ES	Error status				Fetch only of 3-digit octal error code
EX	Error exit	Routine name	0	Yes	
FF	File flush	YES, NO	NO	Yes	BAM only. Buffer flush on abnormal termination
FL	Record length	1 thru BAM 1310710 or AAM 81870	0	No	RT-F/Z only
FLM	Maximum number of records in AAM file	0 thru 536870911	0	Yes	0 allows unlimited number
FNF	Fatal/nonfatal flag				Fetch only: 0 = nonfatal error 1 = fatal error
FO	File organization	SQ, WA, IS, DA, AK	Required	No	IFETCH return: 1 0 = SQ 1 = WA 3 = IS 5 = DA 6 = AK

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
FP	File position				Fetch only: octal FO=WA 1 = BOI 20 = EOR 100 = EOI FO-IS/AK/DA 1 = BOI 20 = EOR 100 = EOI MIP 0 = middle of keylist 1 = BOI 10 = EOK 100 = EOI FO-IS/AK/DA with MIP 1 = BOI 10 = REL and next alternate key is different 20 = REL and next alternate key is same 100 = EOI FO=SQ mid-record 1 = end of labels 2 = BOI 4 = EOY 10 = EOS 20 = EOR 40 = EOP 100 = EOI
FWB	First word address of buffer	Program location	Provided by CRM	Yes	
FWI	Flush buffer IMMEDIATELY	YES, NO	NO	Yes	AAM only. IFETCH return: 0 = NO 1 = YES
HL	Fixed header length	1 thru 81870	0	No	RT=T only
HMB	Number of home blocks	1 thru 16777215	Required	No	FO=DA only
HRL	Hashing routine	Routine name	0	No	FO=DA only; 0 uses system hashing routine
IP	Index block padding percentage	0 thru 99	0	No	FO=IS only
KA	Key location	Program location		Yes	AAM only. Required if EMK=NO

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
KL	Key length in characters	FO-IS 10 (KT=1) or 1 thru 255 (KT= S/U) FO-AK 1 thru 8 FO-DA 1 thru PRL	Required	No	
KNE	Primary key not equal to alternate key	0, 1			BAM only
KP	Key starting position within KA	0 thru 9	0	Yes	FO-IS KT=S/U only
KT	Key type	T, S, U	S	No	FO-IS only. IFETCH returns: 1 = S 2 = I 3 = U
LA	User label area	Label area address	0	Yes	FO-SQ only
LBL	Label area length	1 thru 900	0	Yes	FO-SQ only
LCR	Label check/creation	CHK, CRT	CRT	Yes	FO-SQ only
LFN	Logical file name	1 to 7 letters or digits	Required	Yes	Must start with a letter
LL	Length field length	1 thru 6	0	No	RT=D only
LOP	Last operation code				BAM only. IFETCH returns (octal): 01 = OPENM 02 = CLOSEM 03 = GET/GETP 43 = PUT/PUTP 56 = REPLC 05 = SKIP 47 = WEOR 10 = REWNO 63 = WTKM 74 = ENDFILE
LP	Length field start	0 thru MNR	0	No	RT=D only
LT	Label type	UL, S, NS, ANY	UL	No	FO-SQ only
LX	Label exit	Label routine address	0	Yes	FO-SQ only

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
MBL	Maximum block length	1 thru 1310710	5120; BFS in characters minus 20	No	FO=SQ only. Required for BT=K/E
	Data and index block length	1 thru 81870		No	AAM only. AAM can calculate
MFN	Multifile name	1 to 6 letters or digits			FO=SQ only. Corresponds to the SI or M parameter on the NOS LABEL statement, and to the M parameter on the NOS/BE LABEL statement. Must start with a letter.
MKL	Major key length	0 thru KL	0	Yes	FO=IS only. KT=S/U only; set to 0 at end of every operation.
MNB	Minimum block length	1 thru MBL	0	No	FO=SQ BT=K/E only
MNR	Minimum record length	0 thru MRL	0	No	Required for AAM.
MRL	Maximum record length	0 thru BAM 1310710 or AAM 81870	0	Yes	RT=D/R/S/T/U/W only. Required for FO=AK.
MUL	Multiple of characters per block	0 thru 62	2	No	FO=SQ only. BT=K/E only; must be an even number
NDX	Index flag	0, 1	0	Yes	AAM only. 1 = index file operation
NL	Number of index levels	1 thru 15	1	No	FO=IS only
NOFCP	No FILE control statement processing	YES, NO	NO	Yes	
OC	Open/close status				Fetch only: 0 = never opened 1 = opened 2 = closed
OF	Open flag	FO=SQ R, E, N FO=IS R, E FO=WA R, N	R R R	Yes Yes Yes	Set by OPENM for all

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
ON	Old or new file	OLD, NEW	OLD	Yes	AAM only. Can be set by OPENM
ORG	Old/new file organization	OLD, NEW	OLD	No	AAM only. Must be set to NEW
PC	Padding character	Any character	76 octal	Yes	F0=SQ only. Specify in octal display code
PD	Processing direction	INPUT, OUTPUT, I-O	INPUT	Yes	OPENM sets I-O; IFETCH return: 0 = INPUT 1 = INPUT 2 = OUTPUT 3 = IO
PEF	Parity error flag				F0=SQ fetch only: 0 = no error 1 = parity error
PKA	Primary key address	Address or 0			F0=AK/IS only
PNO	Multifile position				F0=SQ only. Corresponds to LABEL control statement
PTL	Partial transfer length				F0=SQ only. Set by GETP or PUTP
RB	Records per block	1 thru F0=SQ 4095	1	No	BT=K only
		AAM 1 thru 4095	2	No	MBL overrides RB
RC	Record count				Fetch only. Set by GET and PUT for F0=SQ
REL	Relation, key value to key position	EQ, GE, GT	EQ	Yes	F0=IS only
RKP	Relative key position in RKW	0 thru 9	0	No	F0=IS/DA only. Required if EMK=YES
RKW	Relative key word in record	0 thru MRL	0	No	F0=IS/DA only. Required if EMK=YES
RL	Record length	BAM 1 thru MRL	0	Yes	Set by GET and PUT for F0=SQ
		AAM MNR thru MRL	0	Yes	Set by PUT
RMK	Record mark character	Any character	62 octal	No	RT=R only; must not be the same as padding character

TABLE 8-3. CYBER RECORD MANAGER FIT FIELDS (Contd)

FIT Field Mnemonic	Meaning	Allowable Values	Release Default If Any	Change After Creation	Notes
RT	Record type	AAM F, R, Z, D, T, U FO=WA F, U, W FO=SQ F, R, Z, W, D, T, U, S	U W W	No	
SB	Sign overpunch length field	YES, NO	NO	No	RT=D/T only
SBF	Suppress buffer flag	YES, NO	NO	Yes	BAM only. YES suppresses allocation of buffers and circular buffering
SES	System parity error severity				FO=SQ fetch only: 1 = read level 1 2 = read level 2 3 = read level 3 4 = read level 4 5 = write level 1 6 = write level 2
SPR	Suppress read ahead	YES, NO	NO	Yes	FO=SQ only. Reset to NO at the end of processing
TL	Trailer length	1 thru 131071	0	No	RT=T only
ULP	User label processing	NO, V, F, VF, U, VU, FU, VFU	NO	Yes	FO=SQ only (binary): 0 = NO 001 = V 010 = F 011 = VF 100 = U 101 = VU 110 = FU 111 = VFU
VF	Volume close flag	U, R, N	U		FO=SQ only
VNO	Volume number				FO=SQ only
WA	Word address	Memory location			FO=WA only
WSA	Working storage area	Program location	Required	Yes	
XBS	Index file block size	0 thru 81870	Data block size	No	AAM only
XN	Index file name	1 to 7 letters or digits	0	No	AAM only. Must start with letter

All lengths are in characters unless otherwise noted.

TABLE 8-4. CYBER RECORD MANAGER CALLS

Purpose	Description	Syntax
<p>File Information Table Creation and Maintenance (applicable to all file organizations)</p>	<p>Create file information table with values for the fields specified.</p> <p>Set a value in specified FIT field.</p> <p>Retrieve value from specified FIT field.</p> <p>Dump contents of the FIT to the error file. The call forces the EFC field to a value of 2 or 3 to write error messages and/or statistics/notes.</p>	<p>CALL FILExx (fit,field,value,field,value,...field,value)</p> <p>Must be the first call executed for the file.</p> <p>CALL STOREF (fit,field,value)</p> <p>Can precede an OPENM call.</p> <p>IFETCH (fit,field)</p> <p>This is a function only, not a subroutine.</p> <p>CALL FITDMP (fit,id)</p> <p>id is any 10-character identifier to be printed along with the FIT from this particular call.</p> <p>The CRMEP utility must be used to print the file.</p>
<p>File Initialization and Termination (applicable to all file organizations)</p>	<p>Open file.</p> <p>Close file, positioning according to cf specification.</p> <p>Perform file close operations such as buffer flush, but do not close file.</p>	<p>CALL OPENM (fit,pd,of)</p> <p>For FO=SQ, tape labels are processed as long as the file is rewound.</p> <p>CALL CLOSEM (fit,cf,type)</p> <p>IFETCH, STOREF, and FITDMP can follow before file is re-opened. For FO=SQ files, type is FILE or VOLUME.</p> <p>CALL FLUSHI (fit)</p>
<p>File positioning</p>	<p>Reposition file to beginning of information.</p> <p>For FO=SQ/IS/AK, skip records forward or backward.</p> <p>For FO=IS and index file operations, position file.</p>	<p>CALL REWND (fit)</p> <p>For FO=SQ, unlabeled and nonstandard labeled tape files are rewound to the beginning of the current volume; mass storage and standard label files are rewound to beginning of information.</p> <p>CALL SKIP (fit,+count)</p> <p>count is the number of records; -count skips back.</p> <p>For FO=SQ, output files can be positioned only backward; not valid for RT=D/R/T/U or BT=K/E. RT=U cannot be skipped forward.</p> <p>CALL STARTM (fit,ka,kp,mkl)</p> <p>Subsequent sequential processing would follow. Does not transfer a record.</p>

TABLE 8-4. CYBER RECORD MANAGER CALLS (Contd)

Purpose	Description	Syntax
AAM File Data Transfer and File Updating	<p>Read record randomly by key.</p> <p>Read next record in sequence.</p> <p>Transfer next record to working storage.</p> <p>Write record by primary key.</p> <p>Delete record by primary key.</p> <p>Replace record.</p> <p>Initiate block transfer to the file buffer.</p>	<p>CALL GET (fit,wsa,ka,kp,mk1,r1,ex)</p> <p>CALL GETN (fit,wsa,ka,ex)</p> <p>Key of record read is returned to location ka.</p> <p>CALL GETNR (fit,wsa,ka,ex)</p> <p>CALL PUT (fit,wsa,r1,ka,kp,0,ex)</p> <p>For RT=U, FL must be set.</p> <p>CALL DLTE (fit,ka,kp,0,ex)</p> <p>CALL REPLC (fit,wsa,r1,ka,kp,0,ex)</p> <p>Primary key of replacing record must be the same as record being replaced.</p> <p>CALL SEEKF (fit,ka,kp,mk1,ex)</p>
BAM File Data Transfer and File Updating	<p>Read record.</p> <p>For FO=SQ, read partial record of pt1 characters.</p> <p>Write record.</p> <p>For FO=SQ, write pt1 characters to record under construction.</p> <p>For FO=SQ, replace record.</p>	<p>CALL GET (fit,wsa,wa,0,0,r1,dx)</p> <p>For FO=WA, the WA field is incremented to the next record.</p> <p>CALL GETP (fit,wsa,pt1,skip,dx)</p> <p>If skip is 0, read from current position; if 'skip', read from beginning of next record.</p> <p>CALL PUT (fit,wsa,r1,wa,0,0,ex)</p> <p>For FO=WA, writing always begins on word boundary.</p> <p>CALL PUTP (fit,wsa,pt1,r1,ex)</p> <p>Not valid for RT=R.</p> <p>CALL REPLC (fit,wsa,r1,0,0,0,ex)</p> <p>Valid only for mass storage files with BT=C and RT=W.</p>
FO=SQ Boundary Conditions	<p>Write end-of-partition.</p> <p>Write end-of-section terminator of level lev.</p> <p>Write tapemark.</p>	<p>CALL ENDFILE (fit)</p> <p>Synonymous with operating system end-of-file boundary.</p> <p>CALL WEOR (fit,lev)</p> <p>Use to terminate RT=S record being constructed through PUTP. Synonymous with operating system end-of-record.</p> <p>CALL WTMK (fit)</p> <p>Recommended for user tape labels only.</p>

The minimum number of parameters required to create a new file or to access an existing file depends on the file organization. See the CYBER Record Manager user's guides for AAM and BAM for details.

If a file is to be processed both by direct calls to CRM and by standard FORTRAN input/output statements, it must be closed by a standard language CLOSE statement that references the unit specifier before the other access method is invoked.

Figure 8-2 shows an example of a program that processes an indexed sequential file that has alternate keys defined.

CRM UTILITIES

CYBER Record Manager includes several utilities that can be called by control statements. They are summarized in table 8-5.

For FO=DA files, a utility exists to analyze proposed hashing routines. The utility is executed through program calls. It requires a routine in

relocatable binary format and KYAN directives in the job structure. See the CYBER Record Manager AAM manuals for more information.

SORT/MERGE

Sort/Merge is a generalized utility that sorts or merges records according to their key values. A FORTRAN program accesses the product by a series of direct calls similar to the Sort/Merge directives that are processed during a sort or merge executed by a control statement.

Several different versions of Sort/Merge are available:

Sort/Merge 1 is available under SCOPE 2 only. It is similar to Sort/Merge 4.

Sort/Merge 4 is available under NOS and NOS/BE.

Sort/Merge 5 is available under NOS 2 and NOS/BE only.

```

PROGRAM MAKXIS
  IMPLICIT INTEGER (A-Z)
C
C *****
C * THIS PROGRAM ILLUSTRATES THE CREATION OF AN IS FILE *
C * (FLRECS) FROM A SEQUENTIAL FILE (MOVIES). ALL RECORDS *
C * ARE THEN LISTED. TWO ALTERNATE KEYS ARE DEFINED (THE *
C * 2-CHARACTER YEAR-MOVIE-MADE AND THE ACTOR'S NAME). *
C * THE INDEX FILE IS NAMED FLRXIP. *
C *****
C
  DIMENSION XISFIT(35), REC(8)
  CALL FILEIS (XISFIT, 'LFN', 'FLRECS', 'XN', 'FLRXIP',
+             'ORG', 'NEW', 'RT', 'F', 'FL', 80,
+             'KL', 28, 'EMK', 'YES', 'RKW', 0, 'RKP', 3,
+             'WSA', REC, 'EFC', 3, 'DFC', 3)
C
C OPEN FILE BEFORE ALTERNATE KEYS DEFINED
C
  CALL OPENM (XISFIT, 'NEW')
  IF (IFETCH (XISFIT, 'ES') .NE. 0) GO TO 850
  CALL RMKDEF (XISFIT, 0, 0, 2, 0, 'S', 'I')
  CALL RMKDEF (XISFIT, 3, 3, 11, 0, 'S', 'I', 11, 4)
  OPEN (2, FILE = 'MOVIES')
50  CONTINUE
  READ (2, '(8A10)', END = 800) REC
  CALL PUT (XISFIT)
  IF (IFETCH (XISFIT, 'ES') .NE. 0) THEN
    PRINT 902, IFETCH (XISFIT, 'ES')
    CALL CLOSEM (XISFIT)
    STOP ' CRM ERROR RETURNED ON PUT '
  END IF
  PRINT 900, REC
  GO TO 50
800  CALL CLOSEM (XISFIT)
  STOP
850  PRINT 902, IFETCH (XISFIT, 'ES')
  CALL CLOSEM (XISFIT)
  STOP 'CRM ERROR RETURNED ON OPEN'
900  FORMAT (1X, 8A10)
902  FORMAT ('ES = ', 03)
  END

```

Figure 8-2. Example of Creating Indexed Sequential File Having Alternate Keys

TABLE 8-5. CYBER RECORD MANAGER UTILITIES

Description	Syntax
<p>Read information written to the error file during program execution. Information is on a local file that disappears at job termination.</p>	<p>CRMEP,LO=what,SF=lfn,OF=lfn,SN=nnn,ON=nnn,L=lfn,RU,PW.</p> <p>LO Select notes, fatal messages, data manager messages, trivial messages by N, F, D, or T. Omit the specified message for -what.</p> <p>SF, OF Select or omit messages for specified files.</p> <p>SN, ON Select or omit message nnn.</p> <p>L Write output to lfn.</p> <p>RU Return/unload error file after processing.</p> <p>PW Specify 40-160 characters for listing line length.</p>
<p>For FO=IS, compute block sizes based on estimated file characteristics.</p>	<p>FLBLK, listlf, NR=numrecs, KL=keylgh, RL=recTgh, IP=indexpad, DP=datapad, NL=indexlevels, MRL=mrl.</p> <p>Printed output on file listlf shows minimum block size MBL for each index level, as well as information about disk usage at that level.</p>
<p>For AAM files, obtain statistics from the life of data file lfn and any associated index file.</p>	<p>FLSTAT, lfn, listlf.</p> <p>Output is printed on file listlf or OUTPUT. It includes information about the physical structure of the file and transaction counts. It also shows the location and type of the primary key.</p>
<p>For AAM files, create or modify an index file (MIP file) so that data records can be accessed by alternate key values as well as by primary key values.</p> <p>A job calling MIPGEN must adequately describe the files through FILE control statements. It must also issue appropriate permanent file commands.</p>	<p>MIPGEN, dataalfn, directivefn, outlf.</p> <p>dataalfn is the data file; outlf is the file for the MIPGEN listing. directivefn specifies the file containing RMKDEF directives that describe alternate keys.</p> <p>RMKDEF(dataalfn, rkw, rkp, kl, Q, kf, ks, kg, kc, nl, ie, ch)</p> <p>Parameters rkw, rkp, and kl correspond to equivalent FIT fields. kf is key format S, I, or U or P (purge); ks is substructure U, I, or F. Repeating group length is kg, while kc is the number of occurrences. nl suppresses a key value of all zeros or spaces. ch indicates sparse control characters, while ie includes or excludes the value containing the sparse key value by I or E.</p>
<p>For MIP files, disassociate or reassociate the data file and the index file. (Useful when an index file is no longer needed, or when the data file is being restructured into different size blocks.)</p>	<p>MIPDIS, dataalfn, da, indexlf.</p> <p>dataalfn is the data file; indexlf is the index file of alternate keys. da is D for disassociation; A for reassociation.</p>

TABLE 8-5. CYBER RECORD MANAGER UTILITIES (Contd)

Description	Syntax
<p>For FO-DA, create a data file.</p> <p>The job must describe the file with the FILE control statement, load library AAMLIB for Sort/Merge use, load any user hashing routine, and preserve permanent files.</p>	<p>For each input record, the program must read an input file and call SDACRTU. After all records are read, the program must call SDAENDC to sort the records by block and to write the file.</p> <p>CALL SDACRTU (wsa,ka,r1).</p> <p>CALL SDAENDC</p> <p>The file INPUT must contain a CREATE directive to identify the data file LFN. If the system hashing routine is not used, CREATE also must specify the name of the user hashing routine and the file on which it exists in relocatable format.</p> <p>CREATE, data1fn,hashname,hash1fn.</p> <p>data1fn is the name of the data file; hash1fn is the file with any user hashing routine hashname.</p>

Only sequential files can be sorted. The job is responsible for attaching any needed files and for preserving them after Sort/Merge execution.

Sort/Merge is not compatible with the STATO memory management option of the FORTRAN 5 compiler call under NOS and NOS/BE. Under SCOPE 2, STATO is required.

The first call in the program begins a series of specifications for a sort or merge. Subsequent calls establish further specifications, but Sort/Merge does not process any file until a call for execution is encountered. For example, SM5SORT or SM5MERG must be the first specification under Sort/Merge 5; no execution occurs until SM5END is called.

SORT/MERGE 5

Sort/Merge 5 interface subroutines correspond to the directives described in the Sort/Merge 5 reference manual. Table 8-6 shows the FORTRAN program calls in the order they might be specified.

Execution of Sort/Merge 5 requires library SRT5LIB. Sort/Merge 5 can process calls in Sort/Merge 4 format when LDSET(LIB=SRT5LIB) is specified. (Sort/Merge internally substitutes Sort/Merge 5 equivalents for Sort/Merge 4 calls.) Only one style of calls can be used in a program, nevertheless.

Calls can be in any order between SM5SORT or SM5MERG and SM5END. However, if a user-supplied collating sequence is specified, calls to SM5SEQx calls have a particular order. A CLOSE or REWIND statement is needed before Sort/Merge 5 calls if the program has written to the files Sort/Merge will use.

Sort/Merge 5 normally uses CYBER Record Manager for input/output. Records must be described in a FILE control statement, unless they are default BT=C, RT=Z, FL=150 records.

Sort/Merge 5 Keys

Sort/Merge 5 can order character keys either according to a user-defined collating sequence specified by SM5SEQx calls or according to one of the predefined collating sequences. The predefined collating sequences are: ASCII6, COBOL6, DISPLAY, INTBCD, and EBCDIC6.

Numeric keys are sorted or merged in numeric order. A key can have any of the numeric data formats shown in table 8-7.

More than one key field can be specified. Records are ordered first by the key established by the first SM5KEY call, then according to key of the subsequent SM5KEY calls in the order the fields are specified. Key fields cannot overlap. The maximum number of characters in all key fields and all sum fields together is 256.

Summing

One of the features of Sort/Merge 5 totals the contents of specified fields. It allows, for example, a set of transactions to be accumulated into one record.

When one record has the same key value as another, Sort/Merge adds the contents of the specified sum field to the contents of the field in the prior record. Then, one of the duplicate records is deleted. A third record with the same key value is processed the same. At the end of the summing operation, only one record remains, and the contents of the sum field shows the sum from all the records with that same key value. SM5SUM can be called as many as 100 times, as long as the keys and the sum fields do not exceed 256 characters together.

Summing cannot be specified if owncode 5 processing is selected.

TABLE 8-6. SORT/MERGE 5 CALLS

Purpose	Description	Syntax
<p>Processing Control</p>	<p>Begin specifications for merge or sort.</p> <p>Specify array for return of statistics after execution is complete. The amount of information in the array can be limited by setting the first word of the array to the number of elements to be returned. 0 suppresses statistics return.</p> <p>End specifications and begin Sort/Merge execution.</p> <p>Specify estimated number of input records.</p> <p>Select Sort/Merge to directly read records, rather than having CYBER Record Manager read the records. (Also, suppresses some diagnostics.) Valid only if BT=C and RT=F or if BT=I and RT=W.</p> <p>Identify input files from which Sort/Merge is to read records. Default input file name is OLD. (Alternative is to supply records through an owncode routine.)</p> <p>Identify output file to which Sort/Merge writes records. Default output file name is NEW.</p> <p>Identify field for sort or merge key.</p> <p>Retain records with equal sort keys in same order as they are read.</p>	<p>CALL SMSMERG (statistics-array) CALL SMSORT (statistics-array)</p> <p>Values in the 16-word array are integers. Values returned in words 2-13 indicate the number of records: read from input files; deleted by owncode 1; inserted by owncode 1; inserted by owncode 2; sorted or merge; deleted by owncode 3; inserted by owncode 3; inserted by owncode 4; reserved; deleted by owncode 5; combined by summing; written to output file. Words 14-16 indicate in characters: minimum record length; average record length; maximum record length.</p> <p>CALL SMSEND</p> <p>CALL SM5ENR (number)</p> <p>CALL SM5FAST ('option')</p> <p>option is YES to select Sort/Merge; NO to select CRM.</p> <p>CALL SM5FROM ('lfn',... 'lfn')</p> <p>Sort/Merge rewinds the files before and after processing, unless the file name is INPUT or the file was written to by the program.</p> <p>If no input file exists, lfn should be '\$NULL'.</p> <p>CALL SM5TO ('lfn')</p> <p>CALL SM5KEY (first-byte,length, key-type, 'order')</p> <p>Key must be identified by the first character or bit, counting from 1 on left. 'key-type' is a collating sequence name for character keys or a numeric data format; see text for predefined names. 'order' is A for ascending; D for descending.</p> <p>CALL SM5RETA ('YES')</p> <p>Records from multiple input files are read in the order that they are named in SM5FROM.</p>

TABLE 8-6. SORT/MERGE 5 CALLS (Contd)

Purpose	Description	Syntax
<p>Defining Collating Sequence for Character Keys</p>	<p>Name user-defined collating sequence.</p> <p>Specify a value step: that is, identify characters that collate equally.</p> <p>On output, change characters in prior value step to the first character listed.</p> <p>Specify that all characters omitted from SM5SEQS calls are to collate equally.</p>	<p>CALL SM5SEQN ('name')</p> <p>Must precede other SM5SEQx calls.</p> <p>CALL SM5SEQS ('char',... 'char')</p> <p>Each call to SM5SEQS establishes one position in the sequence. (For example, 10 calls are needed to define a 10 step sequence for the digits 0-9.) Characters can be specified by either a literal in the call or a hollerith value in a variable in the parameter list of the call.</p> <p>CALL SM5SEQA ('YES')</p> <p>SM5SEQA can be called once for each SM5SEQS call.</p> <p>CALL SM5SEQR ('YES')</p> <p>SM5SEQR can be called only once for the sequence.</p>
<p>Summing</p>	<p>For records with equal key values, specify a sum field.</p>	<p>CALL SM5SUM (first-char,length, 'numeric-type',rep)</p> <p>Field is specified by its first byte or bit position and its length. 'numeric-type' can be any numeric data type except REAL. rep indicates the number of consecutive identical fields to be involved in summing.</p>
<p>Error Control</p>	<p>Specify file to receive any diagnostic messages. Otherwise, messages go to the file OUTPUT.</p> <p>Specify lowest type of error to be reported.</p> <p>Identify integer variable to which Sort/Merge returns the highest level of error that occurred during execution.</p>	<p>CALL SM5E ('lfn')</p> <p>'\$NULL' suppresses diagnostic messages. Sort/Merge does not rewind the file.</p> <p>CALL SM5EL ('level')</p> <p>T, trivial and all higher levels; W, warning; F, fatal; C, catastrophic.</p> <p>CALL SM5ST (integer-variable)</p> <p>Return codes 10, 20, 30, and 40 correspond to levels T, W, F, and C respectively. 0 indicates that no error occurred.</p>
<p>Owncode Routines</p>	<p>Specify name of routine to be executed at predefined point 1-5.</p> <p>Specify number of characters in fixed length records to enter the sort or merge from owncode routines.</p> <p>Specify maximum number of characters in any variable length record to enter the sort or merge from owncode routines.</p>	<p>CALL SM5OWNn ('proc-name')</p> <p>See text.</p> <p>CALL SM5OFL (integer)</p> <p>CALL SM5OMRL (integer)</p>

TABLE 8-7. SORT/MERGE 5 NUMERIC KEY TYPES

Sort/Merge 5 key-type	Meaning	Sort/Merge 4 Equivalent
NUMERIC_LD	Numeric characters. Leading sign overpunch.	DISPLAY,SIGN,LEADING key-type
NUMERIC_LS	Numeric characters. Leading separate sign.	DISPLAY,SEPARATE,LEADING key-type
NUMERIC_TO	Numeric characters. Trailing sign overpunch.	DISPLAY,SIGN,TRAILING key-type
NUMERIC_TS	Numeric characters. Trailing separate sign.	DISPLAY,SEPARATE,TRAILING key-type
NUMERIC_NS	Numeric characters. No sign.	DISPLAY,SIGN
NUMERIC_FS	Leading blanks, numeric characters.	none
BINARY	Binary integer within character boundaries.	LOGICAL
BINARY_BITS	Binary integer not within character boundaries.	LOGICAL
INTEGER	Ones complement binary integer within character boundaries.	INTEGER
INTEGER_BITS	Ones complement binary integer not within character boundaries.	INTEGER
REAL	Floating point number. Signed.	FLOAT key-type

Sort/Merge 5 Owncode Routines

An owncode routine is a procedure subprogram that executes at one of five predefined points during processing. One user-written routine can be associated with each owncode point 1 through 5. Through an owncode 1 routine, a program can supply individual records to be sorted; alternatively, it can examine and act on each record read from an input file before that record is sorted. Other owncode points allow action at the end of files, before output records are written, and when two records have equal keys.

Owncode routines allow a program to add, modify, and delete records during the sort or merge processing. Table 8-8 shows owncode points, when they execute, and the meaning of the return-code by which the routine controls Sort/Merge action.

Each owncode routine must be specified through a call to SM5OWNa, where a is an owncode point 1 through 5. The routines must also be declared in an EXTERNAL statement. An owncode 1 routine has 5 parameters; the others have 3. Figure 8-3 shows the parameters of the subroutine.

SORT/MERGE 4 AND 1

NOTE

Refer to appendix G for recommendations on the use of this feature.

Sort/Merge 4 and 1 interface subroutines are described in the Sort/Merge 4 and 1 reference manual. Table 8-9 shows the EXTERNAL program calls.

SUBROUTINE name(return-code,record-a,reca-rl,record-b,recb-rl)

- return-code Integer variable by which the subroutine passes instructions back to Sort/Merge. It has a 0 value when the routine is entered.
- record-a Array in which record is passed. Sort/Merge passes a record to the routine for owncode 5, 3, and, if an input file is specified, owncode 1; otherwise, the array is empty. The routine passes a record back to Sort/Merge in this array.
- reca-rl Number of characters in current record. Sort/Merge sets on entry; subroutine can reset for subsequent Sort/Merge use.
- record-b For SM5OWN5 only, array and length
- recb-rl for the second record.

Figure 8-3. Sort/Merge 5 Owncode Subroutine

TABLE 8-8. SORT/MERGE 5 OWNCODE ROUTINES

Owncode Routine	Valid	When Executed	Return-Code Set by Routine	
			Value	Meaning
1	Sort	After each record read from file specified by SM5FROM.	0	Sort record passed back in record-a.
		If input file does not exist, as Sort/Merge is ready for another record to sort.	1	Delete record passed back in record-a.
2	Sort		After each file specified by SM5FROM has been read.	2
		After owncode 1 routine terminates input.	3	Terminate file input and go to next file.
3	Sort Merge	Before writing record to file specified by SM5TO.	0	Sort record passed back in record-a.
		If output file was not specified, as Sort/Merge is ready to output record.	2	Enter record in record-a into sort. Owncode 2 re-executes with this record in record-a.
4	Sort Merge		After last record written to file specified by SM5TO.	3
		After owncode 3 routine terminates output.	0	Begin processing of next input file.
5	Sort	When 2 records have equal keys.	1	Insert first reca-r1 characters from record-a after last record. Owncode 2 re-executes.
			0	Terminate input.

TABLE 8-9. SORT/MERGE 4 AND 1 CALLS

Description	Syntax
<p>Begin specifications for merge or sort. Specify maximum number of characters in largest record. Optionally specify number of words in Sort/Merge working storage.</p> <p>Select mass storage merge.</p>	<p>CALL SMERGE (mrl,buffer-size)</p>
<p>Select mass storage sort.</p>	<p>CALL SMSORT (mrl,buffer-size)</p>
<p>Select balanced tape processing.</p>	<p>CALL SMSORTB (mrl,buffer-size)</p>
<p>Select polyphase type processing.</p>	<p>CALL SMSORTP (mrl,buffer-size)</p>
<p>Identify intermediate tape file names: minimum of 4 for balanced sort, 3 for polyphase sort.</p>	<p>CALL SMTAPE ('lfn',... 'lfn')</p>
<p>For each input and output file, identify use, type of input/output to access file, action at end of sorting/merging.</p> <p>The same file should not be used for both input and output, as it will reset the maximum record length for the file to the Sort/Merge value. An OPEN statement with RECL=nnn can restore the original value.</p>	<p>CALL SMFILE ('use','access','lfn','disp')</p> <p>use SORT; MERGE; or OUTPUT. access FORMATTED or CODED, formatted; BINARY, unformatted. lfn Unit specifier or file name. disp REWIND; UNLOAD; or NONE.</p> <p>If the file is being processed by direct calls to CYBER Record Manager (not supported by SCOPE 2), access should be 0, and lfn should be the FIT array.</p>
<p>For each sort key, describe key and order on output file.</p>	<p>CALL SMKEY (first-byte,first-bit,num-byte,num-bits,'key-type','order')</p> <p>CALL SMKEY (first-byte,first-bit,num-byte,num-bits,'DISPLAY',... 'seq-name','order')</p> <p>Key must be identified by its first character and bit within that character counting from 1 on left. Key length is specified in number of full 6-bit characters and number of additional bits. 'key-type' is: DISPLAY, display code or internal BCD; FLOAT, floating point format; INTEGER, signed integer; LOGICAL, unsigned integer. 'order' is A for ascending order; D for descending.</p> <p>If key-type is DISPLAY, numeric data can be described with additional words 'SIGN', 'LEADING', 'TRAILING', and 'SEPARATE' as shown in table 8-6. A collating sequence must also be specified. Predefined collating sequences are: ASCII6, COBOL6, DISPLAY, INTBCD.</p> <p>If key-type is FLOAT or INTEGER, num-byte and num-bits must total to 60 bits.</p>
<p>Name user-defined collating sequence, and identify array containing characters in the order they are to be collated.</p>	<p>CALL SMSEQ ('seq-name',seq-array)</p> <p>Specify each character in lRx or octal format. Terminate the array with a negative number.</p>
<p>Specify that listed characters collate equally. Each must be previous specified on SMSEQ.</p>	<p>CALL SMEQU ('seq-name',equ-array)</p> <p>Specify characters in lRx or octal format. Terminate the array with a negative number.</p>

TABLE 8-9. SORT/MERGE 4 AND 1 CALLS (Contd)

Description	Syntax
Specify list of options that: suppress dayfile messages; check output file for correct sequencing; retain identical key records in order read from sort files; checkpoint dump at end-of-volume, after n records processed, or not at all; specify merge order; select key comparison or key extraction technique.	<p>CALL SMOPT ('option',... 'option')</p> <p>VERIFY Checking output file for correct sequencing.</p> <p>RETAIN Retain identical key records in order read from sort files.</p> <p>NODAY chkpt Suppress dayfile messages. Checkpoint dump: VOLDUMP, end-of-volume; DUMP,n after n records; NODUMP, suppress dump.</p> <p>ORDER,mo Specify intermediate merge order 2-64.</p> <p>technq Specify COMPARE or EXTRACT key processing technique.</p>
Specify list of owncode exit number 1-5 and associated owncode subroutine name.	<p>CALL SMOWN (exit-number,routine-name,... exit-number,routine-name)</p> <p>Subroutines must be declared EXTERNAL. Exit from routine with call to SMRTN(parameters). See Sort/Merge 4 and 1 reference manual for details.</p>
End Sort/Merge specifications without initiating execution.	CALL SMABT
Begin Sort/Merge execution.	CALL SMEND

Sort/Merge 4 and 1 are not compatible with the ARG=FIXED control statement option of the FORTRAN 4 compiler call. Neither are they compatible with programs in static mode (NOS and NOS/BE) or dynamic mode (SCOPE 2). Sort/Merge must not be called from the (0,0) overlay in an overlay structured program that uses blank common.

The first call must be SMSORT, SMSORTB, SMSORTP, or SMERGE; a call to SMEND must be the last. Otherwise, calls can be in any order. The program must open all files before Sort/Merge is called. If a file is processed by CYBER Record Manager routines as well as by Sort/Merge, the program must call OPEN.

The MRL option on the SMSORT, SMSORTB, SMSORTP, and SMERGE calls is used only when one or more files are FORTRAN files. If all files are processed through direct calls to CYBER Record Manager (access=0 in SMFILE calls), the MRL option is required only as a dummy parameter.

COMMON MEMORY MANAGER

Common Memory Manager (CMM) is a set of small CP COMPASS routines that manage memory during job execution. Program field length is usually managed by CMM unless the STATIC loading routines are called from within the program. CMM routines allocate and release space, change the size of allocated blocks, and obtain memory statistics.

CMM is available under NOS, NOS/BE, and SCOPE 2.

A program can call CMM directly to manipulate memory as the program executes. A program can

request a fixed-position fixed-size block for use like any other piece of memory. CMM returns the first word address (fwa) of the block, and the program subsequently uses fwa to identify the block. Once a block is obtained, it remains active until it is freed by a CMM call. The program can access the entire block contents.

Program loading establishes a static area and a dynamic area: a basic initial load static area size will not change during program execution; an overlay load static area size will fluctuate as execution proceeds as each overlay load occurs. Normally, CMM then manages the total size of the dynamic area without program knowledge, by managing the current space between the limits of the highest high address (hha, which sets the boundary for blocks that must be protected from subsequent loading) and maximum field length. With direct calls to CMM, a program can request and release blocks, and can change the size of the static area by changing the starting address of the dynamic area (daba). Otherwise, daba would be changed only by another job step.

CMM interface routines for NOS and NOS/BE are in the library SYMLIB. For SCOPE 2, the library is SYMIO.

Table 8-10 lists the direct CMM calls. The last 3 characters of the routine name identifies the specific operation. All calls except CMMFRF, CMMSFF, CMMSLF, CMMGLF, CMMCSF, CMMFGR, CMMGBI, and CMMGFS can be used as the first direct call to CMM. Arguments for these calls must be integers, except as noted below. For more information, see the CMM reference manual.

TABLE 8-10. COMMON MEMORY MANAGER CALLS

Type of Call	Description	Syntax
<p>Fixed-Position Block Calls</p>	<p>Allocate a fixed-position block as the address specified. Identify the group to which the block might belong; a group is initially activated by a call to CMMAGR. Indicate changes that can occur in block size by size-code.</p> <p>Same as CMMALF but for flexible allocation of block.</p> <p>Free the specified fixed-position block when it is no longer needed. Block contents cannot then be accessed.</p> <p>Delete a specific number of words from the first word address end of an active block. The contents of the deleted words are lost.</p> <p>Delete a specific number of words from the last word address end. The contents of the deleted words are lost.</p> <p>Add a specific number of words at the last word address (lwa) end of a fixed-position block. Block contents are undefined initially.</p> <p>Change the current size-code of an active fixed-position block.</p> <p>Associate an arbitrary user-supplied 30-bit value with a unique identifier so that the value is accessible through CMMFWA.</p> <p>Obtain the location and current number of words in a block.</p>	<p>CALL CMMALF (blksize,size-code, group-id,fwa)</p> <p>CALL CMMFAF (minsize,maxsize,size-code, group-id,fwa,blksize)</p> <p>Block size is specified by a minimum size and a maximum size, which allows CMM to make more efficient use of free space.</p> <p>CALL CMMFRF (fwa)</p> <p>CALL CMMSSFF (fwa,words-shrink)</p> <p>CALL CMMSLF (fwa,words-shrink)</p> <p>The number of words to lose must not be negative and must not be greater than the current block size.</p> <p>CALL CMMGLF (fwa,words-add)</p> <p>The current block size-code must allow such an addition.</p> <p>CALL CMMCSF (fwa,size-code)</p> <p>Specify new size-code the same as for CMMALF. The block is identified by its current fwa.</p> <p>CALL CMMSIV (user-value,id,unique-id)</p> <p>If id is 0, CMM creates the identifier and returns it in unique-id. Any other positive even integer in id becomes the user-specified unique identifier.</p> <p>CALL CMMFWA (unique-id,fwa-return, blksize-return)</p> <p>The block is specified by the unique identifier established by CMMSIV. This allows the fwa of a fixed block to be recovered across overlay loads.</p>
<p>Group Block Calls</p>	<p>Activate a new logical block group. CMM assigns an integer to the group for future identification in CMMALF, CMMFAF, CMMGBI.</p>	<p>CALL CMMAGR (group-type,group-id-return)</p> <p>If group-type is 0, the group has unrestricted lifetime. If 1, lifetime limited to the next increase in daba. All blocks in a type 1 group must then have a lifetime shorter than the next call to CMMSDA, CMMLDV, or CMMLOV, as these calls increase the size of the static area.</p>

TABLE 8-10. COMMON MEMORY MANAGER CALLS (Contd)

Type of Call	Description	Syntax
Group Block Calls (Contd)	Free all blocks belonging to the specified group. The group is then deactivated, and block contents cannot be accessed.	CALL CMMFGR (group-id)
Overlay, Segment, and Dynamic Area Calls	<p>Initiate loading of an overlay.</p> <p>Initiate loading of a higher level overlay created for processing by the Fast Overlay Loading feature. For more information on Fast Overlay Loading, see the CYBER Loader Reference Manual. Fast Overlay Loading is not available under SCOPE 2.</p> <p>Specify an explicit increase or decrease of the dynamic area base address.</p> <p>Respecify the highest high address. It provides for extended overlay loading, and can be used to give CMM more space for fixed-position blocks when a large overlay is no longer needed.</p> <p>Deactivate CMM so that it no longer exercises any control over the dynamic area. Any block currently active can still be referenced.</p>	<p>CALL CMMLDV (name,l1,l2,n,u,v,e,lwa,fwa,ovlname,eptname,e,fe,status,eptaddr)</p> <p>Arguments correspond to the overlay loading parameters described in the appropriate loader reference manual.</p> <p>CALL CMMLOV (ident,fwa,eptaddr)</p> <p>ident is the overlay name and/or level; fwa is the location; eptaddr is the entry point address of the overlay.</p> <p>CALL CMMSDA (new-daba)</p> <p>The new-daba must be less than or equal to the current value of hha. If new-daba is greater than the value of daba at the time of the call, a type 1 block group cannot be active.</p> <p>CALL CMMSHA (new-hha)</p> <p>The new-hha must be greater than or equal to the current value daba. If new-hha is greater than the current value of hha, no active fixed-position blocks should exist before the call.</p> <p>CALL CMMKIL</p> <p>If CYBER Record Manager is being used, various aborts can occur. Use CMMKIL with caution.</p>
Calls for Current Information	<p>Obtain current information about the block specified by fwa.</p> <p>Determine the size of the largest fixed-position block that can be allocated at the moment. No allocation occurs. Any other call to CMM, except CMMFWA or another CMMGFS call, affects the validity of the value obtained.</p>	<p>CALL CCGBI (fwa,size-code-return,group-type-return,group-id-return,blksize-return)</p> <p>Returns current size-code, the type of group and identifier for the group to which the block belongs, and the current number of words required for the block.</p> <p>CALL CMMGFS (f1-flag,group-type,blksize-return)</p> <p>f1-flag and group-type are specified to CMM as restrictions. If f1-flag is 0, CMM calculates with the current field length. If 1, CMM assumes field length could expand to the maximum field length. The group-type should be 1 if the block can extend below hha, or 0 if it cannot so extend.</p>

TABLE 8-10. COMMON MEMORY MANAGER CALLS (Contd)

Type of Call	Description	Syntax
<p>Calls for Current Information (Contd)</p>	<p>Obtain information about the current state of the dynamic area, returning it to the specified array.</p> <p>Obtain information about the overall history of the dynamic area, returning 6 words to the specified array. Information is applicable to job step summaries, but also could be used for other staistical purposes.</p>	<p>CALL CMMGOS (return-array)</p> <p>Values in the 15-word array will be integers, except for the word containing a percentage. Several words returned are reserved for future use by CDC.</p> <p>Information returned, beginning with the first word of the array, is: dynamic area base address (daba); highest high address (hha); field length; maximum field length; percent of memory used (type real value); reserved; number of contiguous free words; reserved; reserved; number of fixed-position blocks; reserved; number of allocated fixed-position words including the 8 CMM overhead words associated with each fixed block; reserved; number of free areas; number of free words.</p> <p>CALL CMMGSS (return-array)</p> <p>Information returned, beginning with the first word of the array, is: maximum number of allocated words; maximum field length; reserved; number of field length increases; number of field length decreases; reserved.</p>
<p>Miscellaneous Options</p>	<p>Suspend execution of subsequent space-reducing calls until CMMOP2 is called.</p> <p>Reverse CMMOP1.</p> <p>Reduce field length by compacting the dynamic area.</p>	<p>CALL CMMOP1</p> <p>CALL CMMOP2</p> <p>CALL CMMOP4</p> <p>Use when CMM would not be called for some period of time.</p>
<p>User Control of Error Conditions</p>	<p>Pass control to a user-supplied subprogram at the time a CMM abort would otherwise occur. After execution of the user subprogram, CMM restores itself to normal error processing. Any subsequent error will cause an abort, unless CMMOWN is reestablished within the error exit subprogram.</p>	<p>CALL CMMOWN (subprogram)</p> <p>The subprogram must be declared EXTERNAL. It must have one formal parameter that allows CMM to pass a 3-word integer array. The array will contain any memory error condition; the number of any user error that caused the abort; and the address from which CMM was called.</p>
<p>Legend:</p> <p>size-code Integer 0-7 whose value is determined by the bits in a 3-bit code. Each bit that is set to 1 indicates whether an end of the block can change. Bits are: 1xx, block can shrink from fwa; xlx, block can shrink from lwa; xxl, block can grow from lwa.</p> <p>The size-code value places a limit on calls to CMMSSFF, CMMSLF, and CMMGLF that change block size. The size-code can be changed by CMMCSF, although CMMCSF cannot subsequently allow growth from lwa.</p>		

Figure 8-4 shows an example of a program that allocates a single fixed-position block to an array. The technique is applicable to programs that manipulate matrices or other large arrays.

```
PROGRAM CMMZ
DIMENSION CMMAR(1)
ILEN=10
CALL CMMALF(ILEN,0,0,IFWA)
IOFF=IFWA-LOC(CMMAR(1))
CALL TWIDDLE(CMMAR(IOFF+1),ILEN)
CALL CMMFRF(IFWA)
END
```

```
SUBROUTINE TWIDDLE(CMMLBK,ILEN)
DIMENSION CMMLBK(ILEN)
```

```
CMMLBK(5)=1.0
```

```
RETURN
END
```

The main program allocates a block of memory by a call to CMMALF. It calculates an offset within that block using the LOC function, then passes the block address to the subroutine. Within the subroutine, words in the block are referenced as elements of the array. The main program frees the array when it is no longer needed.

Figure 8-4. Example of CMM Use

COMPASS ASSEMBLY LANGUAGE INTERMIXED SUBPROGRAMS

Subprograms written in the COMPASS assembly language can be intermixed with FORTRAN language program units, such that they are assembled by the compiler call.

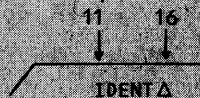


Figure 8-5. COMPASS IDENT Statement

A COMPASS subprogram must begin with a source line (figure 8-5) containing the word IDENT in columns 11 through 15, with columns 2 through 10 blank, and column 16 blank. The subprogram must end with any legal COMPASS END line.

A COMPASS subprogram must not interrupt a FORTRAN program unit. The IDENT line must either immediately follow a previous END line or be the first line passed to the compiler.

COMPASS subroutines and functions can be called from a FORTRAN program. When a FORTRAN-generated subprogram is called, the calling routine must not depend on values being preserved in any register other than A0.

When the COMPASS subprogram is entered by a non-character function reference, the subprogram must return the function result in X6. If the result is double precision or complex, it must be returned in registers X6 and X7 with the less significant or imaginary part of the result in X7.

Within the COMPASS subprogram, register A0 must be restored to its initial condition before returning control to the calling routine.

See the following description of subprogram linkage for additional information about communication between subprograms. See the COMPASS reference manual for details of systems tests.

COMPILER AND SUBPROGRAM LINKAGE

NOTE

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

The FORTRAN 5 compiler uses two methods of passing arguments to subprograms: pass by reference and pass by value. Their use depends on the type of subprogram being called.

Pass By Reference Sequence

The FORTRAN compiler uses the pass by reference sequence for the following references:

References to user-defined functions.

References to intrinsic functions when DB=IB is specified on the FTN5 control statement.

References to any subroutine.

The pass by reference sequence generated is as follows:

```
SAL list-address
+ RJ subprogram-name
- VFD 12/lr
- VFD 13/TRACE.
```

List-address is the address of the argument list; this instruction is not present if the call has no arguments and the ARG-FIXED option is specified on the FTN5 control statement.

Subprogram-name is the name of the subprogram being called. If an intrinsic function is being called, then the name is suffixed with an equal sign.

lr is the source line number of the statement containing the call.

TRACE. is the address of the traceback word in the calling routine.

Arguments in the call must correspond with the argument usage in the called routine, and they must reside in the same memory level. An actual argument of level 1 or 2, however, can correspond to a dummy argument of level 0.

The argument list consists of consecutive words having the format given in table 8-11. If ARG=FIXED is not selected, a zero word terminates the list. Statement label actual arguments appear at the end of the list, preceded by a generated-label (fall through address) parameter, and have a type of label (6 in bits 48 through 50 of argument list word) if ARG=COMMON is specified. The called routine accesses the addresses by offsetting the address of the argument list, which is contained in register A1.

Pass By Value Sequence

The pass by value sequence is used for reference to mathematical intrinsic functions when DB=TB is not specified on the FTNS control statement.

The pass by value code sequence consists of instructions to load the first argument into registers X1 and X2, and the second argument into registers X3 and X4 (the second register of each pair is used only if the argument is type double precision or complex) followed by a return jump instruction. The external linkage name for external functions called by value consists of the function name suffixed with a period.

FUNCTION RESULT

In both calling sequences, the result value of a noncharacter function is returned in register X6 (X6 and X7 for type double precision or complex). For functions whose result is type character, however, the compiler inserts an extra word at the beginning of the argument list to transmit the result. This word has the format shown in table 8-11.

ENTRY POINT

For subprograms written in FORTRAN, the first two words of the subprogram are trace words. The first word contains the subprogram name in left-justified display code (blank-filled) in the upper 42 bits and the subprogram entry address in the lower 18 bits. The second word is used to save the contents of A0 upon entry to the subprogram. The subprogram restores A0 upon exit. The code generated is as follows:

Trace word:

VFD 42/name,18/entry address

A0 word:

DATA 0

Entry point:

DATA 0

TABLE 8-11. ARGUMENT LIST FORMAT

Bit Position	Description
59	Memory where argument resides: 0 Central memory or SCM. 1 Extended memory (ECS or LCM).
58-51	Zero.
50-48	Data type (present only when ARG=COMMON specified): 0 Real, Boolean, Intrinsic 1 Integer. 2 Double precision. 3 Complex. 4 Logical. 5 Character. 6 Label.
47-30	Argument length, in characters. Zero if argument is noncharacter.
29-28	Zero.
27-24	Beginning character position of argument. Value is 0 through 9: 0 means the leftmost character position of a word, 9 means the rightmost character position. 0 if argument is noncharacter.
23-0	First word address of argument.

RESTRICTIONS ON USING INTRINSIC FUNCTION NAMES

Functions written in FORTRAN that have names identical to the mathematical intrinsic function names described in section 7, such as AMAX1 or SQRT, must be declared EXTERNAL in the calling program unit. This declaration causes the compiler to reference the user-defined function, using a pass by reference sequence.

Functions written in COMPASS that have intrinsic function names can use either a pass by reference or a pass by value sequence. An EXTERNAL declaration and a pass by reference sequence must be used in the following cases:

If the function is to be passed as an argument to a subprogram.

If the function has the same name as an intrinsic function that generates in-line code (If the function is not declared EXTERNAL, the compiler generates in-line code).

If a COMPASS routine has the same name as an intrinsic function and pass by value is to be used, a period must be appended to the function name (for example, SIN.).

Figure 8-6 illustrates a sample program containing a call to the intrinsic function SQRT, a call to an external function ZEUS, and a reference to an intrinsic function, AMAX1, that generates in-line code. The object code generated by these calls is shown in figure 8-7. The code generated for the external function ZEUS is illustrated in figure 8-8.

```

PROGRAM SUBLNK
  X=SQRT(7.0)
  Y=ZEUS(X,1.0)
  END
FUNCTION ZEUS(ARG1,ARG2)
  ZEUS=AMAX1(ARG1,ARG2,0.)
  RETURN
  END

```

Figure 8-6. Program SUBLNK and Function ZEUS

8-BIT SUBROUTINES

The 8-Bit Subroutines is a utility package for working with ASCII and IBM sequential files in 8-bit format. It allows data to be converted, moved, compared, and packed. It is the means by which IBM file formats are changed to CDC internal file format.

8-Bit Subroutines is available under NOS and NOS/BE only.

Table 8-12 summarizes the calls to the utility. See the 8-Bit Subroutines reference manual for details of the IBM and CDC data formats and the conversion specifications.

A utility routine is also available to convert IBM print files to CDC print files. The control statement that calls the utility is:

```
COPYSP,ibm-lfn,cdc-lfn,option,...option.
```

ibm-lfn and cdc-lfn

Names of the IBM format and CDC format files, respectively.

option

Description of IBM file or FOLD:
 RECFM=rec-format (F, V, U, FB, VB);
 BLKSIZE=blk-length; LRECL=rec-length;
 CODE=char-code; FOLD to fold output in to the 64-character character set.

CYBER DATABASE CONTROL SYSTEM

CYBER Database Control System (CDCS) is a data management system. CDCS works with a data base defined and controlled by a person known as a data base administrator. The data base itself is described by a schema. Various subschemas define portions of the data for use by application programs. A FORTRAN subschema must be available before a FORTRAN program can access the data in the data base.

CDCS is available on NOS and NOS/BE.

The interface between a FORTRAN application program and CDCS is not the same as direct calls to a product from within an otherwise normal program. There are two types of communication with CDCS. The first is by way of calls to subroutines that provide utility services. These calls to DMLRPT and DMLDBST are written as CALL statements in the program.

Communication with CDCS that allows access to the data itself is by way of Data Manipulation Language (DML) statements in the program. The procedure is:

1. Insert DML statements in the program. SUBSCHEMA, INVOKE, PRIVACY, and OPEN must be the first 4 DML statements. CLOSE and TERMINATE must be the last DML statements. INVOKE is required in any subprogram that uses CDCS.
2. Attach the subschema library. Preprocess the source program using the DML control statement.

DML modifies the program by inserting the subschema description of the data into the program. The modified program is written to local file DMLOUT.
3. Compile the program that DML output.
4. Make library DMSLIB available, and execute the preprocessed object code.

Note that DML statements processed by the preprocessor are not CALL statements. Table 8-13 lists the DML statements. Each statement must begin in column 7. Parameters that appear on the statements are:

END=slab

For sequential read, indicates control transfer when end-of-file is encountered. slab must be a statement label for an executable statement in the program.

ERR=slab

Specifies control transfer after an error occurs. slab is as in END=.

KEY...

Specify the name of the primary or alternate key item. The symbol in the parameter can be =, .EQ., .GT., or .GE. so that an example is: KEY .GE. BIGKEY or KEY=LOWONE.

MODE=open

Specify access mode: I, read only; O, create realm only; IO, updating operations allowed. The mode must match that on any PRIVACY statement for the realm.

PRIVACY=cde

Specify access code for realm. Can be 1-30 characters.

See the FORTRAN data base facility manuals for a further description of DML.

BLOCK	ADDRESS	LENGTH		IDENT	SUBLNK
START.	0B	6B		SUBLNK	TRACE.
CODE.	6B	10B		FILVEC.	BSS 0
LITERL.	16B	2B			ADDR 0,1
FORMAT.	20B	0B			FVEC 11610B
TEMPS.	20B	0B			USE LITERL.
APLST.	20B	4B		CON.	CON 1722700000000000000000B } constant table
IOAPL.	24B	0B			CON 1720400000000000000000B }
NAMLST.	24B	0B			USE FORMAT.
VAR.	24B	4B		AP.1	USE APLST.
SUB.	30B	0B			BSS 0
SUBO.	30B	0B			APL 0003,X+0
BUFER.	30B	0B			APL 0003,CON.+1
					0
				AP.2	BSS 0
					APL 0000,TRACE.+0
					USE IOAPL.
					USE NAMLST.
					USE CODE.
				*	LINE 2 ← source line number
6B	6102000002				SBO B2+0+2
					SA1 CON. ← get actual parameter into X1
7B	0100000000				RJ =XSQRT.
10B	5160000026				SA6 X
				*	LINE 3
11B	6102000003				SBO B2+0+3
					SA1 AP.1 ← get address of parameter list into A1
12B	01000000000003000000				RJT =XZEUS,3
13B	5160000027				SA6 Y
				*	LINE 4
14B	6102000004				SBO B2+0+4
					SA1 AP.2
15B	01000000000004000000				RJT =XEND5.,4
16B					BSS 0
16B					USE START.
4B	6102777747				SBO B2+0-30B
					SBO B2+TRACE.
5B				SUBLNK	BSS 0
5B	5110000002				SA1 FILVEC.
5B					RJ =XQ5RPV.
6B					BSS 0
6B					USE TEMPS.
20B				ST.	BSS 0
20B				CT.	BSS 0
20B				IT.	BSS 0
20B				OT.	BSS 0
20B				VD.	BSS 0
20B				LC.	BSS 0
					USE BUFER.
30B				LENP.	EQUIN
					END
					SUBLNK

Figure 8-7. Object Listing for Program SUBLNK

BLOCK	ADDRESS	LENGTH			
START.	0B	10B			
CODE.	10B	10B			
LITERL.	20B	0B			
FORMAT.	20B	0B			
TEMPS.	20B	0B			
APLST.	20B	0B			
IOAPL.	20B	0B			
NAMLST.	20B	0B			
VARS.	20B	3B			
SUB.	23B	0B			
SUBO.	23B	0B			
BUFER.	23B	0B			
					name of program unit and entry point address
0B	3205252355555000006		ZEUS	IDENT	ZEUS
2B			SAVEA1.	TRACE.	
2B	0000000000000000000			BSS	0
3B				0	
				USE	LITERL.
				USE	FORMAT.
				USE	APLST.
				USE	IOAPL.
				USE	NAMLST.
				USE	CODE.
			*		LINE 2
10B	6102000002			SBO	B2+0+2
	13111			BX1	X1-X1
11B	5020000001			SA2	A0+0+1
	53220			SA2	X2
	31021			FX0	X2-X1
12B	21073			AX0	73B
	13121			BX1	X2-X1
	11310			BX3	X1*X0
	13032			BX0	X3-X2
13B	54300			SA3	A0
	53330			SA3	X3
	31430			FX4	X3-X0
	21473			AX4	73B
14B	13530			BX5	X3-X0
	11054			BX0	X5*X4
	13703			BX7	X0-X3
15B	5170000020			SA7	VALUR.
		+	*		LINE 3
16B	6102000003			SBO	B2+0+3
16B	0400000004+			EQ	EXIT.
			*		LINE 4
17B	6102000004			SBO	B2+0+4
20B				USE	TEMPS.
20B			SUBO1.	BSS	0
20B				BSS	0
20B				USE	START.
3B	6102777754			SBO	B2+0-23B
	6102000000+			SBO	B2+TRACE.
4B			EXIT.	BSS	0
4B	5120000001			SA2	TEMPAO. } restores A0 on exit
	53020	+		SA0	X2
5B	5140000020			SA4	VALUE. }
	10644	+		BX6	X4
6B			ZEUS	BSS	0 ← entry point
6B	0400000006			EQ	ZEUS
7B	74600			SX6	A0
	54010			SA0	A1
	5160000001+			SA6	TEMPAO. }
10B				BSS	0
10B				USE	TEMPS.
20B			ST.	BSS	0
20B			CT.	BSS	0
20B			IT.	BSS	0
20B			OT.	BSS	0
20B			VD.	BSS	0
20B			LC.	BSS	0
				USE	BUFER.
23B			LENP.	EQU	END

Figure 8-8. Object Listing for Function ZEUS

TABLE 8-12. 8-BIT SUBROUTINE CALLS

Description	Syntax
Define a file to be read or written.	<p>CALL XFILE (lfn,workspace,file-string,size)</p> <p>workspace is the working storage area for file lfn; it is size words long. file-string is file information in keyword form, such as FT=C for a card file or USE=W for an output file.</p>
Read a record from an input file, converting data as specified. XREREAD rereads the record with a same or different conversion specification.	<p>CALL XREAD (workspace,destination,conversion-string,status)</p> <p>The record is read into array destination with the resulting format described by a string of one or more conversion specifications. Status is a real variable that receives a status code after the operation.</p>
Write record to output file, converting data as specified.	<p>CALL XWRITE (workspace,source,conversion-string,status)</p> <p>A record is written from array source according to the conversion-string specification. Status is a real variable that receives a status code after the operation.</p>
Compare two character strings of same or different character codes.	<p>CALL XCOMP (xy,source-1,source-2,status,length,position-1,position-2)</p> <p>length characters are compared, starting at position-1 of array source-1 and at position-2 of array source-2. x is the collating sequence for source-1; y is the collating sequence for source-2. Status is a real variable that receives a status code after the operation.</p>
Move character string between arrays, converting as specified.	<p>CALL XMOVE (xy,source,destination,length,position-s,position-d)</p> <p>length characters are moved from position-s of array source to position-d of array destination. x specifies the character set of the source string; y specifies the character set of the destination string.</p>
Compress 12-bits of 8-bit data into words with only 8-bits.	<p>CALL XPACK (string-u,string-p,length,position)</p> <p>length characters of array string-u are packed into array string-p, starting at string-u.</p>
Reverse XPACK process.	<p>CALL XPAND (string-u,string-p,length,position)</p> <p>length characters of array string-u are packed into array string-p, starting at position of string-u.</p>

TABLE 8-13. DML STATEMENTS AND UTILITY CALLS

Type	Description	Syntax
Statements	<p>Identify single subschema to be used in program.</p> <p>Establish link with CDCS. Required in any subprogram that accesses CDCS.</p> <p>Establish right of program to access realm. One statement is required for each realm with controlled access.</p> <p>Initiate realm (or realms joined in a relation) processing.</p> <p>Set lock on realm to inhibit updating by any other program. (CDCS automatically locks each record read when realm is open in IO mode.)</p> <p>Release lock on realm.</p> <p>Logically position realm (or realms joined in a relation) for subsequent sequential read.</p> <p>Transfer data from realm (or realms joined in a relation) to variables defined in the subschema description.</p> <p>Replace last record read.</p> <p>Remove record from realm.</p> <p>Write record using current values of variables defined in the subschema record description. Must be preceded by statements that set the primary key value for the record.</p> <p>Terminate processing of a single realm (or realms joined in a relation).</p> <p>Break link with CDCS.</p>	<p>SUBSCHEMA (subschem-name)</p> <p>Must appear after specification statements and before any data, namelist, or any executable statement.</p> <p>INVOKE</p> <p>Causes all necessary realms (files) to be attached.</p> <p>PRIVACY (realm-name,MODE=open,PRIVACY=cde)</p> <p>open is access mode I, IO, or O. cde is the privacy key of 1-30 characters.</p> <p>OPEN (realm-name,MODE=open,ERR=slab) OPEN (relation-name,MODE=open,ERR=slab)</p> <p>LOCK (realm-name,ERR=slab)</p> <p>UNLOCK (rea,ERR=slab)</p> <p>START (realm-name,KEY.rel.item,ERR=slab) START (relation-name,KEY.rel.item,ERR=slab)</p> <p>If KEY is omitted, operation is performed according to the primary key. The parameter is written as KEY.rel.name, where rel is GE, GT, or EQ, and item is the location of an alternate key value.</p> <p>READ (realm-or-relation-name,KEY.rel.item,ERR=slab) READ (realm-or-relation-name,ERR=slab,END=slab)</p> <p>REWRITE (realm-name,ERR=slab)</p> <p>DELETE (realm-name,ERR=slab)</p> <p>WRITE (realm-name,ERR=slab)</p> <p>CLOSE (realm-name,ERR=slab) CLOSE (relation-name,ERR=slab)</p> <p>TERMINATE</p>

TABLE 8-13. DML STATEMENTS AND UTILITY CALLS (Contd)

Type	Description	Syntax
Subroutines	<p>Establish array to receive status information. CDCS updates values after each operation on the realm or relation.</p> <p>Establish point at which data base can be recovered without loss of data. Use only with consent of data base administrator.</p>	<p>CALL DMLDBST (array,length)</p> <p>The status block is 11 words long, but the array can be set to length to limit the information returned to only the desired information. Array type is integer.</p> <p>Contents of the words are: CDCS or CRM octal error (not status) code; subschema item ordinal for CDCS errors at item level; FP field of CRM FIT: reserved; name of operation being performed when error occurred (left-justified and blank-filled); rank of realm on which CDCS or CRM error occurred during relation; lowest rank on which control break occurred during relation; lowest rank having null record occurrence during relation; name of realm on which error occurred (up to 30 characters).</p> <p>CALL DMLRPT (point-return,comment)</p> <p>point-return receives the integer recovery point number. Comment is a 3-word array that the program can use for any comment. The program is responsible for preserving the comment and number.</p>
<p>Legend:</p> <p>slab Label of an executable statement to which control transfers when an error occurs (ERR=) or the end of file is encountered (END=).</p>		

Figure 8-9 shows the DML control statement that preprocesses the source program.

<p>DML,SB=sublfn,LV=F5,I=inlfn,O=outlfn, E=errlfn,ET=errcde,DS.</p>	
sublfn	Name of file with subschema library.
inlfn	Name of file with program to be preprocessed.
outlfn	Name of file to which preprocessed program is to be written.
errlfn	Name of file to which error information is to be written.
errcde	Type of error that terminates preprocessing: T, W, F, or C.
DS	Suppress listing control (C\$) directives in output listing.

Figure 8-9. DML Preprocessor Control Statement

INFORMATION MANAGEMENT FACILITY

Information Management Facility (IMF) is a software package for describing, organizing, and controlling information in an information base. IMF works from a metabase, which is the stored definition of three different types of schemas that control information. (The conceptual schema defines the rules governing the data base. The internal schema controls the physical organization of the data. Multiple external schemas define the data from the standpoint of different application programs.) Schemas must be available before a FORTRAN program can access the data in the information base.

IMF is available under NOS only.

The interface between a FORTRAN application program and IMF is not the same as direct calls to a product from within an otherwise normal program. Rather, the procedure for accessing data controlled by IMF is:

1. Insert Information Manipulation Language (IML) statements in the program. (These statements are not calls to subroutines.)

The INVOKE statement is required to identify the external and conceptual schemas. It must appear in every main program or subprogram that either contains IML statements or references data in the external schema. It must appear before any executable FORTRAN statement. Other IML statements appear throughout the source program as needed to update the data.

2. Preprocess the source program using the IMF control statement.

IMF modifies the program by expanding the IML statements into FORTRAN statements. The IML statements themselves remain in the original source program in the form of comments. The modified source program is written to the local file SOURCE.

3. Rewind and compile the modified source program that resides on file SOURCE. The FORTRAN 5 compiler must be used.
4. Make the IMF object-time routines on IMFLIB part of the local library set.
5. Execute the object code. Execution-time parameters can be specified on the call to LGO or other execution call.

An IML statement is identified in a FORTRAN program by the character \$ in column 1. Only one IML statement can appear on each line; contiguous continuation lines are allowed, but \$ must appear in column 1 of each line. A line with \$ in column 1 must not otherwise be blank.

Table 8-14 lists the IML statements and uses. Record-names used in the statements must be names of record types defined in the external schema.

The preprocessor control statement and the execution-time statement contain a period within the statement itself. They are shown in figure 8-10 and 8-11, respectively.

IMF,FSIML,I=inlfn,L=listlfn,IBN=metaown, S=metapf,PW=pswr	
inlfn	File containing original source program.
listlfn	File to which diagnostics and listing are to be written.
metaown	User name of metabase owner, assigned when metabase is created.
metapf	Metabase permanent file name.
pswr	Permanent file passwords for metabase.

Figure 8-10. IML Preprocessor Control Statement

LGO,IBN=metaown,S=metapf,PW=pswr,I0=n

metaown Same as for IML control statement.
metapf
pswr

n Number of buffers to be shared by physical files. (Normally the number of information base files plus 1.)

Figure 8-11. Execution-Time Parameters for IMF

During preprocessing, IMF inserts two types of information into the program. First, a common block is established as a communication zone between IMF and the program. During execution, the program can access fields in the zone to examine and test function status, error status, and data constraints. The second type of information inserted by IMF is the data declarations for the record types in the external schema. All fields in the record can then be accessed by the program.

See the IMF application programming reference manual for a description of the communication zone and an example of a program using IMF.

QUEUED TERMINAL RECORD MANAGER (QTRM)

Queued Terminal Record Manager (QTRM) is a utility package that interfaces with the Network Access Method (NAM). With it, a FORTRAN application program can receive and send messages to other terminals and programs without directly using the Application Interface Program (AIP) protocols and supervisory message formats.

QTRM is an indirect means of using the AIP routines. It allows a program to send and receive messages as if it were writing or reading a mass storage file.

QTRM is available under NOS only. Libraries NETIO and NETIOD are required.

The FORTRAN application program communicates with QTRM through a network information table constructed in a program array. The first 10 words of the array are the global portion of the table. The program sets fields to specify the desired connection and the details of message format. Consequently, the calls to QTRM have only a single parameter.

The network information table continues with one 10-word entry for each terminal or application program that the program accesses at one time. During execution, the program can read these entries for status information.

Calls to QTGET and QTPUT are analogous to calls that read or write data. Output transmitted through QTPUT must begin with a format effector character (similar to a carriage control character) that controls display of the message. When the message is in display code characters, QTRM deter-

TABLE 8-14. IML STATEMENTS

Description	Syntax
Specify name of previously compiled external schema and identify conceptual schema from external schema is derived. Match any privacy lock defined in the conceptual schema for data or IMF procedures.	<p>\$ INVOKE EXTERNAL SCHEMA external-schema \$ OF CONCEPTUAL SCHEMA concept-schema \$ KEY IS key \$ USE record-name record-name...</p> <p>If USE is specified, the subprogram can access only the records named.</p>
Prepare information base for processing by this external schema.	\$ OPEN external-schema
Establish a particular record occurrence as the current record occurrence. Does not transfer a record to the program.	\$ FIND record-selection-expression
Return selected record occurrence to the program according to description in the external schema. Establish record as current record occurrence.	\$ OBTAIN record-selection-expression
Add new record to the information base. Specify new record as current occurrence.	\$ STORE record-name RECORD
Modify fields in current record occurrence.	\$ MODIFY record-name RECORD
Remove current record occurrence from information base.	\$ DELETE record-name RECORD
Transfer control to an information base procedure.	\$ CALL information-base-procedure-name
Terminate processing according to specified external schema.	\$ CLOSE external-schema
<p>Legend:</p> <p>record-selection-expression must be one of 4 formats:</p> <p>record-name RECORD VIA access-path-name</p> <p>record-name RECORD VIA access-path-name OR {NEXT HIGHER} {PRIOR LOWER}</p> <p>{FIRST} {LAST} record-name RECORD VIA access-path-name</p> <p>{NEXT } {DIFFERENT} {PRIOR} {DUPLICATE} record-name RECORD VIA access-path-name</p>	

mines the number of characters in the message block and inserts the zero-byte terminator at the end of the block. The program controls any interim line terminators within the block.

The format of the network information table, and the names of the reserved words of fields in the table, are described in the QTRM section of the NAM reference manual. A FORTRAN application program using QTRM must not call AIP routines directly.

Table 8-15 lists the QTRM subroutine calls.

TRANSACTION FACILITY (TAF)

The Transaction Facility is a network host product that controls on-line transaction processing. (A

transaction is a single access to a data base obtained by a user who is not aware of programs.)

TAF requires the facilities of the Network Access Method (NAM). TAF is available under NOS only, where it runs as a subsystem.

Before a terminal user can access a data base, a task (application program) must be written and installed as part of the libraries and tables of the network and TAF. The program can use data management abilities of CYBER Database Control System, CYBER Record Manager, Total, or the TAF data manager. See the Transaction Facility reference manual for a description of the transaction executive, the TAF interface with a data manager, system tasks, and the FORTRAN calls that request TAF capabilities for communicating with terminals.

TABLE 8-15. QUEUED TERMINAL RECORD MANAGER CALLS

Description	Syntax
Establish communication between application program and network.	CALL QTOPEN (net-info-table)
Initiate application-to-application connection.	CALL QTLINK
Obtain input from a connection. Or, obtain status information about a connection.	CALL QTGET (receive-text-area)
Send text data to a connection.	CALL QTPUT (send-text-area)
Disconnect single terminal from program.	CALL QTENDT
Break communication with the network.	CALL QTCLOSE

NOTE

Refer to appendix G for recommendations on the use of Overlays.

Overlays and OVCAPS are used to reduce the amount of storage required, and to make efficient use of the available field length.

The creation and use of overlays and overlay capsules are discussed below.

OVERLAYS

Each overlay is an executable program, and the overlays are a collection of programs combined into an overlay structure. Before program execution, the object modules of an overlay program are linked by the loader and placed on a storage device in their absolute form. Overlays are loaded at execution time without relocation; no linking is required because the linking has already been done. (For more information, see the appropriate loader reference manual.) As a result, the size of the resident loader for overlays can be substan-

tially reduced. Overlays can be used when the organization of the program in memory can be defined prior to execution.

When each overlay is generated, the loader includes library and user subprograms and links them together. The generated overlay is in fixed format, with internal references fixed in their relationship to one another. The generated overlay has a fixed origin address within the field length and is not relocatable. At execution time, the loader simply reads the required overlay from the overlay file and loads the overlay at the preestablished origin within the field length for the user program.

The following discusses main, primary, and secondary overlays; overlay communication; creating overlays; and calling overlays.

MAIN, PRIMARY, AND SECONDARY OVERLAYS

Overlays are loaded into memory at three levels. The general positioning of main, primary, and secondary level overlays is shown in figure 9-1.

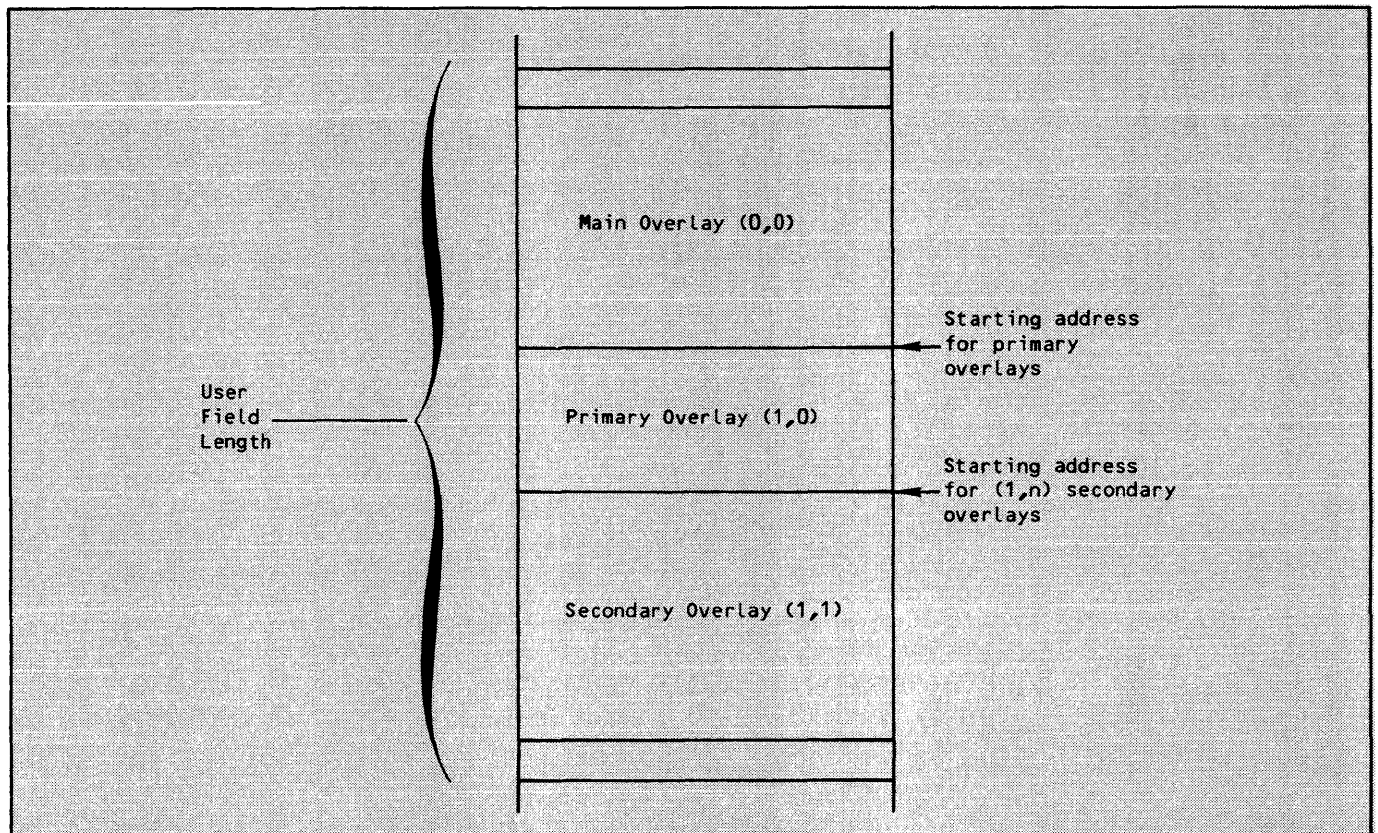


Figure 9-1. Overlay Positioning

Overlays are identified by a pair of integers, in the following way:

- (0,0) Main overlay
- (n,0) Primary overlay
- (n,k) Secondary overlay

The values *n* and *k* are positive octal integers in the range 1 through 77 octal. For any given program execution, all overlay identifiers must be unique. For example, (1,0), (2,0), (3,0), and (4,0) would be primary overlays; and (3,1), (3,2), (3,5), and (3,7) would be secondary overlays associated with primary overlay (3,0). Secondary overlays are identified by the primary overlay number and a nonzero secondary number. Overlay numbers of the form (0,*n*) with *n* greater than zero are not valid.

The main or zero overlay is loaded first and remains in memory at all times. A primary overlay can be loaded immediately following the zero overlay, and a secondary overlay can be loaded immediately following the primary overlay. Overlays can be overlaid by other overlays. For example, if a different secondary overlay is required, the loader simply moves it from the overlay file into memory, placing it at the same starting address as the previously loaded secondary overlay.

When a primary overlay is loaded, the previously loaded primary overlay and any associated secondary overlays are destroyed. For this reason, no primary overlay can load other primary overlays. In the same way, loading a secondary overlay destroys a previously loaded secondary overlay.

A secondary overlay can be called into memory only by its primary overlay. For example, overlay (1,0) can call overlay (1,2), but overlay (2,0) cannot call overlay (1,2). Execution is faster if the more commonly used subprograms are placed in the main overlay, which remains in memory at all times. The less commonly used subprograms can be placed in primary or secondary overlays that are called into memory as required.

An overlay can consist of one or more FORTRAN or COMPASS program units. Each overlay must contain one FORTRAN main program, but the FORTRAN main program need not be the first program unit in the overlay. When the overlay is called, the program name in the PROGRAM statement becomes the primary entry point for the overlay.

OVERLAY COMMUNICATION

Data is passed between overlays through labeled common or blank common. An element of a blank or labeled common block in the main overlay (0,0) can be referenced by any higher (primary or secondary) level overlay. Any blank or labeled common block declared in a primary overlay can be referenced by the primary overlay and the associated secondary overlays, but not by the main overlay. An example is shown in figure 9-2.

Labeled common blocks are generated in the overlay in which they are first encountered. Data in a labeled common block can only be preset in the overlay in which the labeled common block is generated.

Blank common is located at the end, that is, the highest address of the first overlay in which blank common is declared. Blank common located at the end of the (0,0) overlay is accessible to all other overlays. If blank common is declared in the (1,0) overlay, blank common is at the end of the (1,0) overlay and is accessible only to the associated (1,*k*) secondary overlays.

The loader cannot satisfy external references from lower level to higher level overlays. Thus, the user should include a call to OPEN, INQUIRE, BACKSPACE, or any of the intrinsic functions in the primary overlay if the external references are to be used in only the associated secondary overlays. If the external references are to be used in different primary or secondary overlays, the user should include the call in the (0,0) overlay. These calls cause the routines to reside in either the primary overlay or the (0,0) overlay, depending on where the calls are included.

CREATING OVERLAYS

An overlay is identified by an OVERLAY statement that precedes the PROGRAM statement in each overlay. An overlay consists of all program units between the OVERLAY statement and the next OVERLAY statement, or the end of the source input.

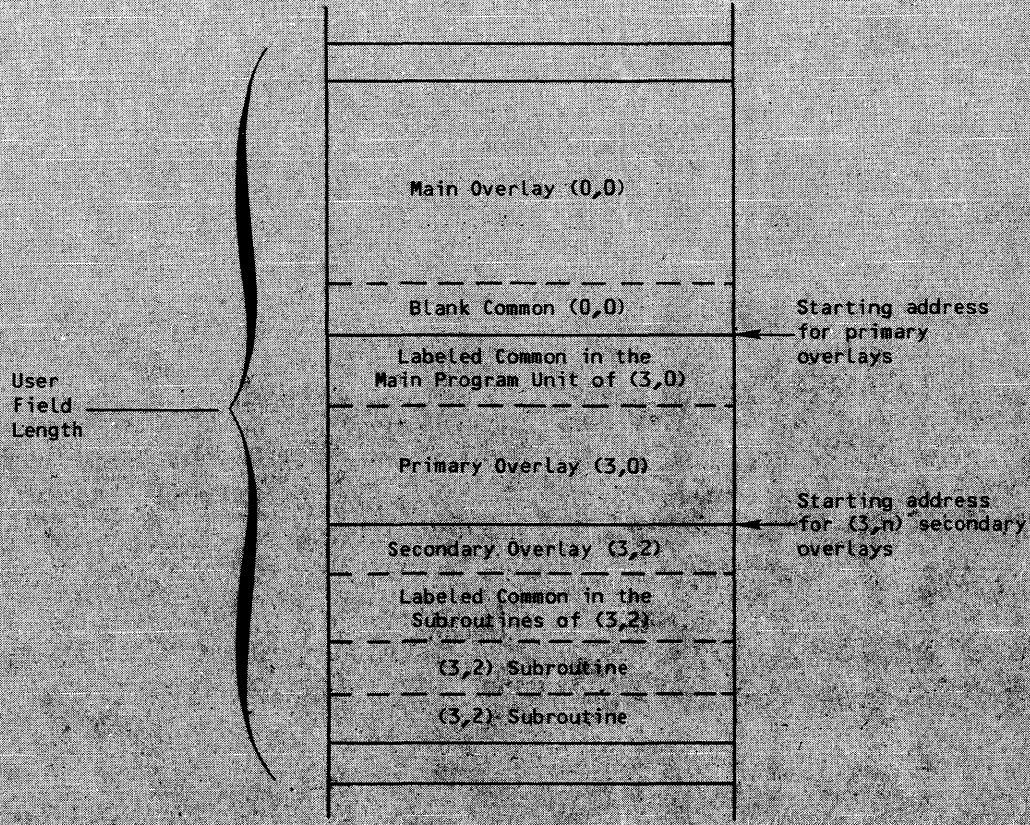
The OVERLAY statement is shown in figure 9-3.

The OVERLAY statement cannot be continued. The required order of the OVERLAY statements, and the requirements for *fname*, *i*, *j*, *orig*, and *OV=m* are described in greater detail in the appropriate loader reference manual.

An efficient technique is to specify *OV=n*. Loading overlays from a file requires an end-around search of the file for the specified overlay, which can be time-consuming in large files. If speed is essential, the overlays should be called in the same order in which they were generated, or fast overlay loading should be used.

The group of relocatable decks to be processed by the loader to create an overlay-structured program must be presented to the loader in a specific order. The main overlay must be loaded first. Any primary group is followed by its associated secondary group, then any other primary group is followed by its associated secondary group, and so forth.

The first OVERLAY statement must have a file name and *i,j* must be 0,0 for the main overlay. Subsequent statements can omit file name, indicating that the overlays are to be written on the same file. The second OVERLAY statement must be of a primary overlay such as (3,0). If the *orig* parameter is omitted, the overlay is loaded in the normal way directly after the main overlay. The *orig* parameter cannot be included on the main OVERLAY statement, but is used on primary and secondary overlay statements to change the size of blank common at overlay generation time.



Labeled common declared in the main program unit of primary overlay (3,0) is at the bottom of the primary overlay. For the secondary overlay (3,2), the blank common in the main program unit is at the end of the main program part of the overlay. The labeled common declared for the two subroutines precedes the subroutines.

Blank common is declared in the main (0,0) overlay. Blank common is located at the end of the (0,0) overlay and is accessible to all other overlays.

Figure 9-2. Overlay Positioning Showing Common

OVERLAY ([fname,i,j[,orig][,OV=m])									
fname	Is the name of the file on which the generated overlay is to be written.								
i,j	Are the overlay level numbers in octal (0 through 77). The numbers specified are not checked or converted by FORTRAN.								
orig	Is an optional parameter specifying the origin of the overlay; not allowed for (0,0) overlay. The loader accepts any of the following forms: <table border="0" style="margin-left: 20px;"> <tr> <td>Cnnnnnn</td> <td>The overlay is loaded nnnnnn words from the start of blank common; nnnnnn must be an octal number (0 through 77777).</td> </tr> <tr> <td>0=nnnnnn</td> <td>The overlay is loaded at the address specified; nnnnnn must be an octal number $\geq 110_8$.</td> </tr> <tr> <td>0=ept</td> <td>The overlay is loaded at the address of the entry point specified, which must have been declared in a lower level overlay.</td> </tr> <tr> <td>0=ept±nnnnnn</td> <td>The overlay is loaded at the address of the entry point specified, but the address is biased by the amount of the offset nnnnnn.</td> </tr> </table>	Cnnnnnn	The overlay is loaded nnnnnn words from the start of blank common; nnnnnn must be an octal number (0 through 77777).	0=nnnnnn	The overlay is loaded at the address specified; nnnnnn must be an octal number $\geq 110_8$.	0=ept	The overlay is loaded at the address of the entry point specified, which must have been declared in a lower level overlay.	0=ept±nnnnnn	The overlay is loaded at the address of the entry point specified, but the address is biased by the amount of the offset nnnnnn.
Cnnnnnn	The overlay is loaded nnnnnn words from the start of blank common; nnnnnn must be an octal number (0 through 77777).								
0=nnnnnn	The overlay is loaded at the address specified; nnnnnn must be an octal number $\geq 110_8$.								
0=ept	The overlay is loaded at the address of the entry point specified, which must have been declared in a lower level overlay.								
0=ept±nnnnnn	The overlay is loaded at the address of the entry point specified, but the address is biased by the amount of the offset nnnnnn.								
OV=m	Is an optional parameter specifying a decimal number for the total number of higher level overlays in the overlay structure. The OV parameter is valid only for the (0,0) overlay and causes the overlay generator and loader to use Fast Overlay Loading (FOL) Default is OV=0. See the CYBER Loader reference manual.								

†Not valid on SCOPE 2.

Figure 9-3. OVERLAY Statement

CALLING OVERLAYS

Only the main overlay (0,0) is loaded when the control statement that calls the main overlay is encountered. Primary and secondary overlays are called with the OVERLAY call shown in figure 9-4.

CALL OVERLAY (fname,i,j[,recall[,k]])				
fname	Is either the file name of the file containing the (i,j) overlay to be executed (if k is absent or zero), or the overlay name (if k is non-zero). The name can be either: <table border="0" style="margin-left: 20px;"> <tr> <td>A Boolean expression with the value nHf, where f is the name of the file.</td> </tr> <tr> <td>An arithmetic expression with the value nHf, after conversion $BOOL(fname)$.</td> </tr> <tr> <td>A character expression containing the name of the file, after trailing blanks are removed.</td> </tr> </table>	A Boolean expression with the value nHf, where f is the name of the file.	An arithmetic expression with the value nHf, after conversion $BOOL(fname)$.	A character expression containing the name of the file, after trailing blanks are removed.
A Boolean expression with the value nHf, where f is the name of the file.				
An arithmetic expression with the value nHf, after conversion $BOOL(fname)$.				
A character expression containing the name of the file, after trailing blanks are removed.				
i,j	Are the overlay level numbers and can be integer or Boolean expressions. The values are converted as necessary to INT(i) and INT(j).			
recall	Is the recall parameter and can be: <table border="0" style="margin-left: 20px;"> <tr> <td>A Boolean expression with the value $\delta HRECALL$.</td> </tr> <tr> <td>An arithmetic expression with the value $\delta HRECALL$, after the conversion $BOOL(recall)$.</td> </tr> <tr> <td>A character expression with the value 'RECALL', after trailing blanks are removed.</td> </tr> </table>	A Boolean expression with the value $\delta HRECALL$.	An arithmetic expression with the value $\delta HRECALL$, after the conversion $BOOL(recall)$.	A character expression with the value 'RECALL', after trailing blanks are removed.
A Boolean expression with the value $\delta HRECALL$.				
An arithmetic expression with the value $\delta HRECALL$, after the conversion $BOOL(recall)$.				
A character expression with the value 'RECALL', after trailing blanks are removed.				
k	Is an indicator affecting the interpretation of fname. If k is non-zero, fname is an overlay in a library. See the appropriate loader reference manual.			

Figure 9-4. OVERLAY Call

If the recall parameter is specified, the overlay is not reloaded if it is already in memory. If the overlay is already in memory and the recall parameter is not used, the overlay is actually reloaded, thereby changing the values of variables in the overlay.

When a RETURN or END statement is encountered in the main program of the main overlay, execution of the program terminates and control returns to the operating system. When either of the statements is encountered in the main program of a primary or secondary overlay, control returns to the next executable statement after the CALL OVERLAY statement that invoked the current overlay. A RETURN in a subroutine returns only to the calling routine.

An example is shown in figure 9-5.

```

OVERLAY(XFILE,0,0,OV=2)
PROGRAM ONE
.
.
CALL OVERLAY(SHXFILE,1,0,0)
.
.
STOP
END
OVERLAY(XFILE,1,0)
PROGRAM ONEZERO
CALL OVERLAY(SHXFILE,1,1,0)
.
.
RETURN
END
OVERLAY(XFILE,1,1)
PROGRAM ONEONE
.
.
RETURN
END

```

Execution of RETURN in the (1,1) overlay returns control to the (1,0) overlay at the statement following the (1,1) call. Execution of RETURN in the (1,0) overlay returns control to the statement in the main overlay that follows the (1,0) call.

Figure 9-5. Sample Overlay Structure

OVCAPS

The use of overlay capsules is another method for reducing the amount of storage required for running large programs. Overlay capsules, commonly called OVCAPs, provide a capability similar to overlays. The difference is that OVCAPs are not specified by levels and there exists no limitation on the number of OVCAPs that can be present in memory at any one time. OVCAPs are not supported on SCOPE 2.

An overlay has a fixed first word address (FWA); an OVCAP has no such limitation. An OVCAP is loaded into memory within a block that is made available by Common Memory Manager (CMM) at execution time. This allows user code to co-exist, in the CMM-managed portion of memory, with other capsules and buffers that can belong to the user or to system routines. By making use of the user-callable CMM routines, the user can manage memory very efficiently by utilizing OVCAPs instead of overlays.

OVCAP COMMUNICATION

Data communication between OVCAPs is carried out similar to data communication between overlays. An OVCAP, as defined by the loader, is a logical extension of an (0,0) overlay. OVCAPs require a (0,0) overlay to be present; communication is via

common blocks specified in the main overlay and in the capsule. Any common blocks that exist in an OVCAP, but not in the (0,0) overlay, are processed as local to the OVCAP.

An OVCAP cannot contain references to data resident in ECS or LCM. If a user places an intermixed COMPASS routine in an OVCAP, no non-standard relocation is permitted (standard relocation is the relocation of 18 bit fields whose rightmost bit is bit 0,15 or bit 30 of a central memory word).

CREATING OVCAPS

An OVCAP consists of one or more program units. The first program unit specifies the OVCAP name. Unlike overlays, the first program unit must be a subroutine and not a main program; no main programs are permitted in OVCAPS.

An OVCAP is identified by an OVCAP directive placed before the first subroutine of each OVCAP. The OVCAP consists of all program units between the OVCAP statement and the next OVCAP statement, or at the end of source input. The name of the OVCAP is the name of the first subroutine in that OVCAP, which must immediately follow the OVCAP statement. The first subroutine must be a subroutine subprogram without alternate entry points. The format of the OVCAP directive is shown in figure 9-6.

```

OVCAP.

```

Figure 9-6. OVCAP Directive

OVCAP directives are placed in the source input stream in the same manner as overlay directives. The only exception is that at least one overlay directive, and all overlay directives for the current overlay structure, must precede OVCAP directives. The first directive of the source input stream must be a (0,0) overlay directive whether or not any other overlays are to be created.

The loader requires that the OV=n parameter of the OVERLAY statement be specified in the (0,0) overlay. The n parameter must be the sum of the number of primary overlays, secondary overlays, and OVCAPs.

An OVCAP generation load sequence must be terminated by a NOGO directive, or a fatal loader error is generated.

All OVCAPs must be generated in the same load sequence.

CALLING OVCAPS

The FORTRAN 5 compiler includes three calls for manipulation of OVCAPs: CALL LOVCAP, CALL XOVCAP, and CALL UOVCAP.

CALL LOVCAP (figure 9-7) causes the specified OVCAP to be loaded into memory. Execution is not initiated. OVCAPs are loaded from the same place as the main overlay; this can be a file or a library.

CALL LOVCAP(name)

name Is a variable or constant of type character.

Figure 9-7. LOVCAP Call

CALL XOVCAP (figure 9-8) initiates execution of the OVCAP specified. If the OVCAP has not been loaded prior to this call, it is first loaded and then executed. The user parameters are passed to the subroutine entered in the OVCAP.

CALL XOVCAP(name, user plist)

name Is a variable or constant of type character.

user plist Is a list of parameters passed to the subroutine entered in the OVCAP.

Figure 9-8. XOVCAP Call

CALL UOVCAP (figure 9-9) unloads the specified OVCAP.

CALL UOVCAP(name)

name Is a variable or constant of type character.

Figure 9-9. UOVCAP Call

Up to 4095 OVCAPs can be in memory at any time. Any OVCAP can load and execute any other OVCAP. An OVCAP must exit through the entry point of the first subroutine to return to the statement after the statement that initiated execution of the OVCAP. Any OVCAP can unload any other OVCAP. Be sure that no OVCAP unloads an OVCAP that was called in the chain prior to the execution of the present OVCAP. Undefined results occur if this is attempted.

If execution of an OVCAP is initiated by an overlay other than the (0,0) overlay, that overlay must not be replaced by an overlay of the same level until the called OVCAP has returned to the statement that initiated its execution.

CYBER Interactive Debug cannot be used to debug OVCAPs. If OVCAPs are present during a Post Mortem Dump run, they are ignored.

A batch job set up to compile and execute a user job containing OVCAPs is shown in figure 9-10.

```
JOB.  
USER(TOM,DICK)  
CHARGE(1234,56789)  
FTN5.  
LOAD(LG0)  
NOGO.  
FRUIT.  
7/8/9(multi-punch)  
OVERLAY (FRUIT,0,0,OV=3)  
PROGRAM APPLE  
.  
.  
CALL XOVCAP ('PEACH',A,3)  
.  
.  
CALL UOVCAP ('PEACH')  
CALL UOVCAP ('PEAR')  
CALL LOVCAP ('BERRY')  
.  
.  
END  
OVCAP.  
SUBROUTINE PEACH (X,1)  
.  
.  
CALL LOVCAP ('PEAR')  
CALL LOVCAP ('BERRY')  
.  
.  
CALL XOVCAP ('PEAR')  
END  
OVCAP.  
SUBROUTINE BERRY  
.  
.  
END  
OVCAP.  
SUBROUTINE PEAR  
.  
.  
CALL XOVCAP ('BERRY')  
.  
.  
END  
6/7/8/9 (multi-punch)
```

Figure 9-10. Batch Job Set Up for OVCAPs

Two debugging aids can help find execution-time errors in a FORTRAN program. These are the CYBER Interactive Debug facility (CID) and the Post Mortem Dump facility (PMD). Sample programs illustrating the use of CYBER Interactive Debug and Post Mortem Dump are included in section 12.

CID is intended primarily for interactive use. It is available under NOS and NOS/BE, but not under SCOPE 2. Only a brief summary of CID is presented in this section. Refer to the CYBER Interactive Debug reference manual for more detailed information.

PMD is intended primarily for use with batch jobs. It is available under all operating systems.

CYBER INTERACTIVE DEBUG

CYBER Interactive Debug is a supervisory program that allows program execution to be monitored and controlled from a terminal. CID provides the ability to:

- Suspend execution at specified program locations. These locations are called breakpoints.

- Suspend execution when specified events (such as the loading of an overlay) occur. These events are called traps.

- Display or alter the contents of program variables after execution is suspended.

- Resume program execution at the point of suspension.

No special statements are required in the source program to use CID. However, a special compilation is required if the symbolic address capabilities and special FORTRAN commands are to be used.

CID does not support OVCAPS.

Program compilation in debug mode, session initiation, and some CID commands are discussed below.

PROGRAM COMPILATION

To use all the CID capabilities, the program must be compiled in debug mode. This can be done either by specifying the DEBUG control statement prior to compilation or by specifying ID as an option of the DB parameter on the FTN5 control statement.

When a source program is compiled in debug mode, the compiler produces a symbol table and a line number table along with the binary object code.

CID accesses these tables during program execution to allow the user to reference program addresses symbolically.

To execute under control of CID, debug mode must be turned on prior to the program load. The CID module is loaded along with the compiled code and becomes part of the user's field length.

If a source program has not been compiled in debug mode, it can still be executed under CID control. In this case, however, program addresses cannot be referenced symbolically and the special FORTRAN commands for displaying and assigning values to variables cannot be used.

CID requires OPT=0 compilation.

DEBUG Control Statement

The DEBUG control statement activates debug mode. The format of this statement is:

```
DEBUG.  
or  
DEBUG(ON)
```

Debug mode can be subsequently turned off by the statement:

```
DEBUG(OFF)
```

A program that has been compiled with DEBUG(ON) can be executed in a normal manner with DEBUG(OFF).

DB Parameter

The DB=ID parameter can be specified on the FTN5 control statement to compile a program in debug mode. After compilation, the DEBUG(ON) control statement must be specified to execute the program under CID control.

When any DB option is selected, the remaining options assume the initial values discussed in section 11. To select only the ID option, DB=0/ID must be specified. A parameter of DB=0 overrides a previous DEBUG(ON) control statement for the duration of compilation.

An example of an FTN5 statement to compile a program in debug mode is:

```
FTN5,I=PROGA,DB=ID.
```

The source file PROGA is compiled and written to file LGO along with debug tables. The Post Mortem Dump facility is also available.

DEBUG SESSION

To execute a program under CID control, the `DEBUG(ON)` control statement must be in effect.

The debug session is then initiated by entering the name of the binary object file. Normally, this causes execution of the user program. In debug mode, however, control transfers to an entry point in CID rather than to the program itself. CID responds with the message:

```
CYBER INTERACTIVE DEBUG
```

```
?
```

The ? is a prompt that indicates CID is waiting for input. Typically, CID commands to set traps and breakpoints are entered to define the conditions under which execution is to be suspended. Then, program execution is initiated by the command:

```
GO
```

The GO command is also used to resume execution after suspension.

The program will execute until a breakpoint or trap is reached. Then, control passes back through CID to the user. Any CID command can then be entered. Typically, commands would be used to display or change the current values of program variables, as shown below.

To terminate the debug session, the `QUIT` command is used. The format of the command is:

```
QUIT
```

This command terminates the present execution under CID control. Debug mode remains in effect until `DEBUG(OFF)` is entered.

The following paragraphs introduce some CID commands that can be used to conduct a simple debug session.

SET,BREAKPOINT Command

A breakpoint is a location within a program where execution is to be suspended. The format of the command to establish a breakpoint is:

```
SET,BREAKPOINT,L.n
```

where n is the FORTRAN source line number where the breakpoint is to be established. When CID encounters a breakpoint during program execution, control immediately transfers to CID which issues the message and prompt:

```
*B #n at L.n
```

```
?
```

At this point, the user can enter any CID command. In a typical session, commands are entered to display or change the current values of program variables.

SET,TRAP Command

A trap is a CID device that detects the occurrence of a specified condition during program execution, suspends execution at that point, and transfers control to CID. The format of the command to establish a trap is:

```
SET,TRAP,type,location
```

where type is the trap type and location is the trap location.

CID provides several trap types. One of the traps most useful to the FORTRAN programmer is the `LINE` trap. This trap gives control to CID immediately prior to the execution of each FORTRAN line, allowing the user to step through the execution of a program one line at a time. An example of a command to set a `LINE` trap in program `PROGA` is:

```
SET,TRAP,LINE,P:PROGA
```

PRINT Command

The `PRINT` command displays the values of specified variables. This command is identical in form and function to the FORTRAN list directed `PRINT` statement. The format of the `PRINT` command is:

```
PRINT*,list
```

where list is a list of variable names separated by commas. The values of the variables are formatted according to the type declared in the source program. An example of a `PRINT` command that displays the values of A, B, and the fifth element of array C is:

```
PRINT*, A,B,C(5)
```

The `PRINT` command can be issued any time in a debug session after CID has issued a ? prompt.

Assignment Command

The assignment command is used to assign new values to program variables. This command is identical in form and function to the FORTRAN assignment statement. The format of the assignment command is:

```
variable=expression
```

where expression is any valid FORTRAN expression not involving exponentiation or function references. The expression is evaluated and the value replaces the current value of the variable. Some examples of assignment commands are as follows:

```
A=1.0
```

```
DX=DY*BZ+1.0
```

```
F=(A+T)/(B+T)
```


OTHER CID FEATURES

Other CID features include:

A trap to suspend program execution when data is stored into a specified variable

A trap to suspend program execution when an overlay is loaded

The capability of defining sequences of CID commands to be executed automatically on the occurrence of a trap or breakpoint

The capability of saving trap and breakpoint definitions on a separate file

The capability of interrupting a debug session

Refer to the CYBER Interactive Debug reference manual for descriptions of these and other CID features.

POST MORTEM DUMP

Post Mortem Dump (PMD) is designed to analyze the cause of execution time errors in FORTRAN programs. PMD provides interpreted output in a form which is more easily understood than the octal dump normally output following a fatal error. PMD prints a readable summary of the error condition and the state of the program at the time of failure in terms of the names used in the original program. Thus, the names and values of the variables in the routine in which the error was detected are printed; this is repeated tracing back through the calling sequence of routines until the main program is reached.

PMD can be used with overlay programs. In this case, only variables defined in the overlay currently in memory are dumped. The overlay numbers of the current overlay appear in the PMD output.

PMD does not support OVCAPS.

When PMD is selected, the interpreted dump occurs whenever a fatal execution error occurs. In addition, PMDLOAD or PMDSTOP calls can be placed in a program to cause a dump when the statement executes.

COMPILATION

To use PMD, the DB=PMD parameter must be specified on the FTN5 control statement.

Use of PMD does not significantly increase the execution time for jobs that terminate normally; nor does it significantly increase the run-time field length.

PMD does not require modification of the source program or the control statements, and it does not significantly affect the compiler-generated code. Thus, PMD can be combined with other compiler options such as OPT (optimization) and other DB (debug) selections.

Different compilation optimization levels generate different sequences of object code. At the actual time of an abort, the machine instruction being executed for a specified optimization level might be different from the instruction being executed for a different optimization level. Thus, different PMD output can occur.

Variable values printed by PMD might differ for successive executions of the same program on certain computer systems. This can occur on systems with multiple functional units such as the 6600, 7600, CYBER 70 Models 74 and 76, CYBER 170 Models 175 and 176, and the CYBER 700 Models 750 and 760.

LOADING

PMD suppresses any user-specified load map directive or MAP(ON) control statement. For example, the following statements do not produce a load map if DB=PMD was specified:

```
LDSET(MAP=SBEX)
LOAD(LGO)
REXECUTE.
```

However, the loader always writes a block-type map to file ZZZZMP if DB=PMD was specified. The contents of this file can be printed by copying it to file OUTPUT. If nonfatal loader errors occur, a summary of the errors is included in the PMD output.

When DB=PMD is specified on the FTN5 control statement, the compiler generates a loader request to preset all memory to a special value for initialization testing. This preset is similar to that produced by the following load sequence:

```
LDSET(PRESETA=60000000000433400000)
LOAD(LGO)
EXECUTE.
```

Any user LDSET(PRESET=) loader specification is overridden.

DUMP INFORMATION

Information provided by the dump includes the following (where applicable):

A summary of all nonfatal loader errors

A list of all COMMON block length clashes

The nature of the error that activated PMD

The array-dumping parameters selected and the field length required to load and run the user program

The activity of each file used by the user program at the time of the error

The overlays in memory at the time of the error

The location of the error in terms of statement labels and line numbers, if possible

An annotated register dump for some system errors; an attempt is made to associate each address register with a variable or array referenced within the routine in which the error occurred

An alphabetic list of all variables and their values, accessible from the current routines

A printout of arrays according to specified parameters

A message-tracing call beginning at the previous routine and ending when the main program is reached

A completion message upon reaching the main program

The elements of an array to be dumped are controlled through the `PMDARRY` statement. Variables for a routine that is not in the traceback chain are dumped only if the `PMDUMP` statement is in effect. These statements are explained below.

For batch jobs, the dump is written to the file `OUTPUT`. For jobs executed from an interactive terminal, the disposition of the dump is determined by options specified on the execution control statement (typically `LC0`) as follows:

`LC0,*OP=option[option][option].`

where `option` is one of the following:

- T** A condensed form of the dump is displayed at the terminal. This option is valid for interactive jobs only.
- F** A full dump is written on the file `PMDUMP` when the job is executed with the file `OUTPUT` connected. This option is valid for interactive jobs only.
- A** The variables in all active routines are included in the dump. An active routine is a routine that has been executed but is not necessarily in the traceback chain. This option is valid for batch, as well as interactive, jobs.

Dump Variables

Dump variables are printed in an alphabetically sorted list. The column labeled `RELOCATION` is blank for local variables. For `COMMON` variables, it contains the block name. For formal parameters, it reads `F.P. nn` where `nn` indicates the parameter number. For multiple entry points, `nn` indicates the parameter number corresponding to the entry point entered, as listed in the compilation reference map.

In addition to being printed as numbers, integer variables are interpreted as masks or characters in `H`, `L`, or `R` format. In character representation, binary zeros are converted to blanks within a word, but a word with binary zeros at each end has the first binary zero printed as a colon.

The column headed `COMMENTS` flags undefined local variables as `UNDEF`, which indicates a potential source of error.

Variables passed as parameters to the previous routine in the traceback tree are labeled `PARAM nn` in the `COMMENTS` column. The `COMMENTS` column contains `F.P. nn`; where the same variable occurs more than once in an argument string, `nn` points to the last occurrence. Constants passed to the previous routine are also printed at the end of the list and given the symbolic name `CONSTANT`. Untraceable functions and subroutines passed as arguments are printed.

Full checking is carried out on subroutine or function arguments and a warning message is issued if:

A routine is called with the wrong number of arguments.

A type conflict exists between actual and formal arguments.

The argument was a constant and the called routine either treated it as an array or corrupted it.

A conflict in the use of `EXTERNAL` arguments is detected; note that the results given for `EXTERNAL` arguments can be imprecise because several utilities can reside within the same routine and `PMD` cannot differentiate between them. For example, both `SIN` and `COS` reside within the routine `SINCOS`.

A warning message is also issued if a real variable contains an unnormalized value, for example, an integer.

Variables that have never had a value assigned to them are printed as having the value `NOT INITIALIZED`. Variables that have not had storage assigned to them by the compiler because of optimization are printed as having the value `NOT ASSIGNED STORAGE`.

PMD Traceback

`PMD` can, in most cases, trace back from the point at which an error occurs to the main program. Traceback occurs repeatedly through the calling sequence of routines until the main program is reached. In `PMD` traceback, `PMD` prints the names and values of the variables in the routine in which an error is detected.

Traceback is abandoned under the following conditions:

`PMD` discovers a routine name that occurred earlier in the traceback. For example, the descendent of an ancestor routine has called the ancestor again.

Traceback flows through a routine that was not invoked by standard `FORTRAN` calling conventions, such as those used in a function or subroutine reference. For example, if an abort should occur while performing an input/output operation in a `CRM` routine, the traceback is abandoned because `CRM` uses the sequence:

```
SE6 address  
JP B6
```

to perform the equivalent of the standard calling sequence:

RJ address

that a function or subroutine reference generates.

Multiple entry points exist in a system resident routine. If multiple entry points are in a system resident routine, traceback might be abandoned because PMD continues traceback from the first entry point (which is assumed to be the last active entry point for the routine). If traceback continues at an entry point other than the last active entry point for the routine, traceback continues up a calling chain other than the one that produced the error. CPU.SYS is an example of a routine with multiple entry points that is subject to this condition.

When traceback is abandoned the following error message is issued:

```
***TRACEBACK ABANDONED
```

PMD SUBROUTINES

The following paragraphs describe the individual Post Mortem Dump subroutine calls in alphabetical order.

PMDARRY

The PMDARRY call (figure 10-1) controls the number of elements printed in each dimension of an array dump. The number of arguments in the call statement selects the type of arrays to be dumped. If 1 argument exists, only 1-dimensional arrays are dumped; if 2 arguments exist, both 1- and 2-dimensional arrays are dumped; and so forth. If 7 arguments are specified, all arrays are dumped.

```
CALL PMDARRY (i,j,k,l,m,n,o)
```

i through o are integers indicating the limits on the first through seventh array subscripts, respectively.

Figure 10-1. PMDARRY Call

If PMDARRY is omitted, the dump is equivalent to the statement PMDARRY(20,2,1,1,1,1,1). This means that the first 20 elements of the first dimension are printed, the first 2 elements of the second dimension are printed, and only 1 element of any remaining dimensions is printed.

PMDARRY produces no immediate output until a dump is forced by some other PMD call or by a fatal execution error.

The following special form of the execution control statement performs the same function as a call to PMDARRY:

```
LGO,*DA=i+j+k+l+m+n+o.
```

where i through o represent the first through seventh dimensions respectively.

If the call to PMDARRY and LGO,*DA is omitted, the effect is the same as CALL PMDARRY(20,2,1,1,1,1,1); for example, all arrays (1 through 7 dimensions) are dumped, but only elements whose subscripts do not exceed (20,2,1,1,1,1,1) are included in the dump.

Once PMDARRY has been called, the established conditions apply to all program units in the user program. Any number of PMDARRY calls can be included; the most recent call determines the effective conditions.

Array elements are printed with the first subscript varying fastest and with a maximum of six values per line for REAL, INTEGER, and LOGICAL arrays, and a maximum of three values per line for DOUBLE PRECISION and COMPLEX arrays.

Example:

```
DIMENSION RAY(10,10,10)
```

```
.
```

```
.
```

```
.
```

```
CALL PMDARRY(3,4,1)
```

Twelve elements of array RAY will be printed:

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

If all the requested elements of an array have the same value, PMD will print the message:

```
ALL REQUESTED ELEMENTS OF THIS ARRAY  
WERE...
```

If several consecutive elements of an array sub-block have the same value, PMD prints the message:

```
ALL THREE ELEMENTS WERE...
```

PMDDUMP

The PMDDUMP call (figure 10-2) causes a dump of variables in the calling routine when an abort occurs or when PMDLOAD or PMDSTOP is called.

```
CALL PMDDUMP
```

Figure 10-2. PMDDUMP Call

PMDDUMP and PMDLOAD or PMDSTOP need not be called from the same routine. The dump includes an analysis of all active routines that have called PMDDUMP. (An active routine is a routine that has been executed but is not necessarily in the traceback chain.) In addition, following an abort or call to PMDSTOP, all routines in the traceback chain are dumped. Up to ten subprograms can be dumped. If more than ten subprograms call PMDDUMP, the extra calls are ignored.

PMDLOAD

The **PMDLOAD** call (figure 10-3) causes an immediate dump of variables in the calling routine and in any routines that have called **PMDUMP**.

Program execution continues normally after the dump unless **PMDLOAD** is called 10 times, in which case the last call is treated as a call to **PMDSTOP**.

CALL PMDLOAD

Figure 10-3. **PMDLOAD** Call

PMDSTOP

The **PMDSTOP** call (figure 10-4) causes an immediate dump of variables in the calling routine, all routines in the traceback chain, and any routines that have called **PMDUMP**. The job is then aborted.

Programs cannot recover from a call to **PMDSTOP**.

CALL PMDSTOP

Figure 10-4. **PMDSTOP** Call

This section discusses the FTN5 control statement options, the compiler listings, the reference map, the object listing, and the execution control statement.

FTN5 CONTROL STATEMENT

The FORTRAN compiler is called from the system library and executed by an FTN5 control statement. The FTN5 control statement calls the compiler, specifies the files to be used for input and output, and indicates the type of output to be produced. This control statement can have any of the formats shown in figure 11-1. Some examples of FTN5 control statement are as follows:

```
FTN5(ET=W,LO=R,S=0)
FTN5,I=INF,L=LIST.
```

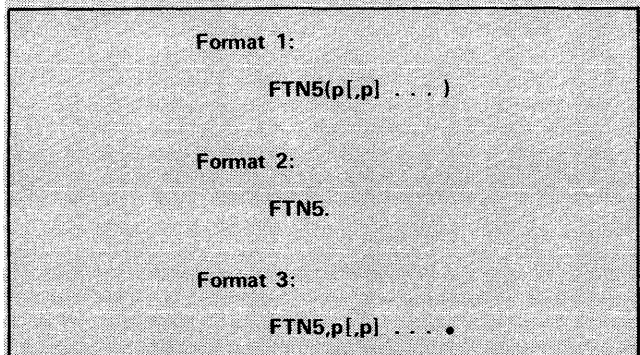


Figure 11-1. FTN5 Control Statement

PARAMETERS

The optional parameters p_1 through p_n must be separated by commas and can be in any order. If no parameters are specified, FTN5 is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the syntax for job control statements as defined in the operating system reference manual, with the added restriction that a comma is the only valid parameter delimiter. Columns following the right parenthesis or period can be used for comments; they are ignored by the compiler, but are printed on the dayfile.

Most parameters need not be specified because a default value will be used if the parameter is omitted. This default value (called the first default) has been chosen to be the most commonly desired value. First defaults are set when the system is installed; since installations can change first default values, the user should determine what values are in effect at the user's particular installation.

There is a second parameter value that is almost as commonly desired as the one chosen for the first default. Writing only the parameter name will select the second most commonly used value for the parameter (called the second default). Second defaults are not installation changeable.

Unrecognizable parameters cause compilation to terminate. Conflicting options either are resolved or cause compilation to terminate, depending on the severity of the conflict; this resolution is indicated in a dayfile entry.

The values of the B, E, G, I, L, ML, PD, S, and X parameters are passed to COMPASS when intermixed COMPASS subprograms are present.

Parameters fall into two general classes: those with two possible settings, on or off; and those that have a specific value, such as a file name, optimization level, and so forth.

Binary Value Parameters

A binary value parameter has two possible settings, on or off. On is selected by writing the parameter name. Off is selected by writing parameter-name=0.

Specified Value Parameters

Specified value parameters provide the compiler with specific values for such things as file names, page size, and so forth. For parameters where multiple values are allowed, the values are separated by slashes. The specified value parameters are indicated in table 11-1.

Multiple Binary Value Parameters

Some binary value parameters are grouped together under a single parameter because they control related operations; for example, listing options. Such groupings of binary value parameters are called multiple binary value parameters. Like all parameters, first and second default values are defined to cover the most frequent usages. The form of a multiple binary value parameter is:

```
parameter=op [/op]...
```

where op is a binary value indicating on or off.

If options are selected for a multiple binary value parameter, they are processed as follows:

1. An initial value is selected for each option corresponding to the most commonly desired options.
2. The option list is scanned from left to right.
3. -op deselects the specified option.

TABLE 11-1. DEFAULTS FOR FTN5 CONTROL STATEMENT

Parameter	First Default (parameter omitted)	Second Default (keyword only)	Initial Values
AL	AL	AL	AL=0
ANSI	ANSI=0	ANSI=T	
ARG	ARG=0	ARG=-COMMON/FIXED	ARG=0
B	B=LGO	B=BIN	
BL	BL=0	BL	
CS	CS=USER	CS=FIXED	
DB	DB=0 if opt=1, 2, or 3; DB=0/ER if opt=0	DB=TB/SB/SL/ER/PMD	DB=TB/SB/SL/ER/PMD
DO	DO=0	DO=OT	DO=0
DS	DS=0	DS	
E	E=OUTPUT	E=ERRS	
EC	EC	EC	
EL	EL=T	EL=F	
ET	ET=0	ET=F	
G	G=0	G=SYSTEXT	
GO	GO=0	GO	
I	I=INPUT	I=COMPILE	
L	L=OUTPUT	L=LIST	
LCM	LCM=D	LCM=1	
LO	LO=S/A	LO=S/A/R	LO=S/A
MD	MD=0	MD=T	
ML	ML=0	ML=0	
OPT	OPT=0	OPT=2	
PD	PD=6	PD=8	
PL	PL=5000	PL=50000	
PN	PN=0	PN	
PS	PS=60 if PD=6; PS=80 if PD=8	None	
PW	PW=136 (PW=72 for connected file)	PW=72	
QC	QC=0	QC	
REW	REW=0	REW=I/B	REW=I/B/E
ROUND	ROUND=A/S/M	ROUND=A/S/M/D	ROUND=0
S	S=SYSTEXT if G omitted S=0 if G specified	S=SYSTEXT if G omitted S=0 if G specified	
SEQ	SEQ=0	SEQ	
STATIC	STATIC=0	STATIC	
TM	TM=characteristics of compiling machine	TM=0	TM=0
X	X=OLDPL	X=OPL	

4. op selects the specified option.

5. 0 deselects all previously selected options.

If the initial value set is close to what the user desires, the easiest way to select the values is by addition or deletion of specific options. For example, if DB=TB/SB/SL/ER/ID is wanted, the user would write FTN5,DB=-PMD/ID to remove PMD from the initial values and to add ID.

If the initial value set is not what is desired, the user should delete all of the initial values and add the desired values. For example, if only DB=PMD is wanted, the user would write FTN5,DB=0/PMD to deselect the initial value set and to add PMD.

Multiple binary value parameters are indicated in table 11-1 by a value in the initial value column.

Multiple Appearances of Parameters

The G and S parameters are the only parameters which can have multiple appearances in a FTN5 control statement. If any other parameter appears more than once, compilation terminates.

PARAMETER OPTIONS

Following are descriptions of the options for each of the FTN5 control statement parameters.

AL Automatic Level

The AL parameter controls the placement of common blocks on machines with LCM or UEM (800 Series). Valid options are:

omitted

Same as AL=0.

AL

Selects placement of all common blocks in LCM or UEM unless overridden by an explicit LEVEL statement. All Elements of such a common block are treated as LEVEL 2 items. All dummy arguments of subroutine and functions will be treated as if LEVEL 0 had been specified for them unless an explicit LEVEL declaration overrides them.

AL=0

All common blocks will be in central memory (SCM) unless overridden by an explicit LEVEL statement.

If automatic level is selected (AL), all items in common blocks will be assigned to LEVEL 2 storage and all dummy arguments will be treated as LEVEL 0. This automatic treatment can be overridden by explicit LEVEL statements specifying LEVEL 1, if central memory residence is desired.

The following precautions should be noted when using the AL option:

Existing COMPASS programs that share common blocks with FTN5 programs may need to have their declaration of the common block changed to specify UEM (LCM).

BUFFER I/O of data in LCM requires a FILE command with SBF=NO specified for the file.

Variable formats in LCM are limited to a maximum of 150 characters.

ANSI Diagnostics

The ANSI parameter specifies whether use of non-ANSI extensions to FORTRAN are to be diagnosed and if so, how severely. Valid options are:

omitted

Same as ANSI=0.

ANSI

Same as ANSI=T.

ANSI=0

No ANSI diagnostics are generated.

ANSI=T

ANSI errors are treated as trivial errors.

ANSI=F

Non-ANSI usages result in a fatal error.
All ANSI warning diagnostics become fatal.

Refer to the EL parameter for an explanation of trivial and fatal diagnostics.

ARG Argument List Attributes

The ARG parameter is a multiple binary value parameter that specifies attributes of external procedure argument lists generated by the compiler. (For more information about selecting and deselecting initial values on this compiler option, see the heading Multiple Binary Value in the beginning of this section.) Valid options are:

omitted

Same as ARG=0.

ARG

Same as ARG=COMMON/FIXED.

ARG=0

Same as ARG=FIXED/COMMON; both options are deselected.

ARG=op [/op]

where op is one of the following:

COMMON - Argument lists generated for external procedures will be of the form `COMMON language communication`. Specification of COMMON implies `FIXED`. The selection of this option must be the same for both the calling and the called program units.

FIXED - All references in the FORTRAN program to a given external procedure have the same number of arguments (the compiler-generated argument lists will not contain a zero terminator).

Initial value is ARG=0. Specification of ARG=COMMON/FIXED is not permitted.

B Binary Output File

The B parameter specifies the name of the compiler output file. Valid options are:

omitted

Same as B=LGO.

B

Same as B=BIN.

B=0

No binary output file is produced. Cannot be specified with GO.

B=lfm

Compiler-generated binary code is output on the file lfm.

The B parameter conflicts with the QC parameter.

BL Burstable Listing

The BL parameter controls page ejects in the listing produced by the compiler. Valid options are:

omitted

Same as BL=0.

BL

Generates output listing that is easily separable into components by issuing page ejects between source listing, cross-reference-attributes map, and object code listing. Also ensures that each program unit listing contains an even number of pages, issuing a blank page at the end if necessary.

BL=0

Generates listings in compact format by minimizing page ejects.

CS Collating Sequence

The CS parameter specifies the weight table to be used for the evaluation of character relational expressions. Valid options are:

omitted

Same as CS=USER.

CS

Same as CS=FIXED.

CS=USER

User-specified weight table.

CS=FIXED

Fixed weight table. (See collating sequence control in section 7.)

DB Debugging Option

The DB parameter is a multiple binary value parameter that selects debugging options. Valid options are:

omitted

Same as DB=0 if Opt=1, 2, or 3; DB=0/ER if Opt=0.

DB

Same as DB=TB/SB/SL/ER/PMD.

DB=0

All options are deselected.

DB=op [/op]...

where op is one of the following:

TB - A full error traceback occurs upon detection of an execution time error. This option causes arguments to intrinsic functions to be passed by reference and argument range errors to be diagnosed.

SB - Subscript bounds checking is performed.

SL - Character substring expressions are checked to ensure that the substring references are within the string.

ER - Selects object time relieve of execution errors. A message identifying the program unit and line number containing the error is output.

ID - A line number table, symbol table, and special stylized object code (required by CYBER Interactive Debug) are generated along with the binary code. CYBER Interactive Debug uses the tables while processing the user program to determine variable locations, source line locations, and other useful debugging information. This option must be specified if the special FORTRAN features of CYBER Interactive Debug are to be used and debug mode has not been turned on by the DEBUG control statement. DB=ID overrides a previous DEBUG control statement specification. DB=ID requires OPT=0.

ST - Same as ID except that the stylized object code is not generated.

PMD - Must be specified if the Post Mortem Dump facility is to be used. Symbol tables are written to separate files that are accessed by Post Mortem Dump so that a symbolic analysis of error conditions, variable names and values, and traceback information can be written to an output file. Use of PMD may increase the run-time field length and increase execution time for jobs that terminate normally.

Initial value is DB=TB/SB/SL/ER/PMD. If PMD is specified, ARG=FIXED must not be selected.

DO Loop Control

The DO parameter is a multiple binary value parameter that specifies the manner in which DO loops are to be interpreted by the compiler. (For more information about selecting and deselecting initial values on this compiler option, see the heading Multiple Binary Value in the beginning of this section.) Valid options are:

omitted

Same as DO=0.

DO

Same as DO=OT.

DO=0

Trip count must be less than 131071 and minimum trip defaults to zero.

DO=op [/op]

where op is one of the following:

LONG - Permits the trip count to exceed 131071.

OT - Sets the minimum trip count for DO loops to one. This option can result in faster program execution.

Most DO loops have a trip count of at least one and will execute correctly under either option. However, if a DO loop has a trip count of zero and DO=OT is specified, the program might not execute correctly. The effects of this parameter can be overridden by the C\$ DO directive. Initial value is DO=0.

DS Directive Suppression

The DS parameter suppresses the recognition of C\$ directives. Valid options are:

omitted

Same as DS=0.

DS

All C\$ directives are treated as comments.

DS=0

All C\$ directives are recognized and processed.

E Error File

The E parameter specifies the name of the file to receive error information. Valid options are:

omitted

Same as E=OUTPUT.

E

Same as E=ERRS.

E=lfn

In the event of an error of EL-specified severity or higher, the error line and diagnostic are written to lfn. If the L (full listing) parameter is specified, this information is also written to the file specified by the L parameters. E=0 is an error.

EC Extended Memory Usage

The EC parameter specifies that OPT=2 tables will use LCM or ECS, when available.

omitted

Same as EC.

EC

Use LCM or ECS for OPT=2 tables, if available.

EC=0

Do not use LCM or ECS for OPT=2 tables.

EL Error Level

The EL parameter indicates the severity level of errors to be printed on the output listing. The levels are ordered by increasing severity. Specification of a particular level selects that level and all higher levels. Valid options are:

omitted

Same as EL=T.

EL

Same as EL=F.

EL=T

Lists trivial errors. The syntax of these errors is correct but the usage is questionable.

EL=W

Lists warning errors. These are errors where the syntax is incorrect but the compiler has made an assumption (such as inserting a comma) and continued.

EL=F

Lists fatal errors. A fatal error prevents the compiler from processing the statement where the error occurred.

EL=C

Lists catastrophic errors. These errors are fatal to compilation; the compiler is unable to continue processing the current program unit. Compilation continues with a subsequent program unit.

ET Error Terminate

The ET parameter specifies the action to be taken by the compiler when compilation has completed. If an error of the specified level or higher occurs, the job skips to an EXIT statement under NOS or to an EXIT(S) under NOS/BE and SCOPE 2. Valid options are:

Omitted

Same as ET=0.

ET

Same as ET=F.

ET=0

The job continues even if errors are encountered.

ET=T

Skips if errors of severity T or higher are detected.

ET=W

Skips if errors of severity W or higher are detected.

ET=F

Skips if errors of severity F or higher are detected.

ET=C

Skips if errors of severity C are detected.

Refer to the EL parameter for a description of error severity levels.

Level T and W errors result in an executable binary file. Level F and C errors result in a binary that will abort the loader.

G Get System Text File

The G parameter specifies the name of a file to be read to obtain a system text for intermixed COMPASS subprograms. Valid options are:

omitted

Same as G=0.

G

Same as G=SYSTEXT

G=0

No system text is loaded.

G=lfm

Loads the system text from file lfm.

G=lfm-recname

Loads the system text from record recname on file lfm. (The hyphen is required separator notation.)

Up to seven system texts can be specified, separated by slashes. Multiple occurrences of this parameter are permitted. A G=0 specification is ignored if any other G option is specified.

GO Automatic Execution

The GO parameter specifies automatic loading and executing. Valid options are:

omitted

Same as GO=0.

GO

The binary output file is loaded and executed after compilation.

GO=0

The binary output file is not loaded and executed after compilation.

The GO option conflicts with the QC and B=0 options.

I Input File

The I parameter specifies the name of the file containing the input source code. Valid options are:

omitted

Same as I=INPUT.

I

Same as I=COMPILE.

I=lfm

Source code to be compiled is contained in file lfm. Compilation ends when an end-of-section, end-of-partition, or end-of-information is encountered. I=0 is an error.

L List File

The L parameter specifies the name of the file where the compiler writes the source listing and any other requested listing information except diagnostics (see LO parameter). L=0 suppresses all listings except that directed to the E file. Valid options are:

omitted

Same as L=OUTPUT.

L

Same as L=LIST.

L=0

Listing suppressed.

L=lfm

Listing is on file lfm.

LCM Extended Memory (LCM, ECS, or UEM Storage) Access

The LCM parameter specifies the requirements on address size for data in extended memory. See the LEVEL statement for further information. Valid options are:

omitted

Same as LCM=D.

LCM

Same as LCM=I.

LCM=D

Specifies direct mode addressing. Provides more efficient code for accessing data assigned to extended memory. Extended memory field length must not exceed 131071 words.

LCM=I

Specifies indirect mode (21-bit) addressing. This is the most efficient mode when extended memory field length exceeds 131071 words. If a single common block exceeds 131071 words, an error results.

LCM=G

Specifies giant mode addressing. Required if any single common block is larger than 131071 words. LCM=G automatically selects DO=LONG.

LO Listing Options

The LO parameter is a multiple binary value parameter that specifies the information that is to appear on the output listing file (L parameter). (For more information about selecting and deselecting initial values on this compiler option, see the heading Multiple Binary Value in the beginning of this section.) Multiple options can be specified. Valid options are:

omitted

Same as LO=S/A.

LO

Same as LO=S/A/R.

LO=0

No O, R, A, M, or S information appears on the output listing.

LO=op [/op]...

where op is one of the following:

O - Output object code (COMPASS mnemonics) is listed on the output file.

R - A cross-reference map (described later in this section) is written to the output file.

A - A list of program entities (variables) and their attributes (data type, class, and so forth) is written to the output file.

M - A map, showing the correlation of program entities and physical storage, is written to the output file.

S - A source listing of the program is written to the output file. To produce only a source listing, specify LO=S/-A.

Initial value is LO=S/A. The effects of the LO parameter can be controlled by the C\$ LIST directive, described in appendix E.

MD Machine Dependent Diagnostics

The MD parameter specifies whether or not the use of machine-dependent language features are to be diagnosed, and how severely. Valid options are:

omitted

Same as MD=0.

MD=0

Machine-dependent diagnostics are not generated.

MD=T

Machine dependencies are treated as trivial errors.

MD=F

Machine-dependent language feature results in a fatal error.

Refer to the EL parameter for explanation of trivial and fatal errors.

ML MODLEVEL Micro

The ML parameter specifies the value of the MODLEVEL micro used by COMPASS. Valid options are:

omitted

Same as ML=0.

ML

Same as ML=0.

ML=0

The current date, in the form yyddd (where yy is the year and ddd is the number of the day within the year), is used for the MODLEVEL micro.

ML=str

The string str is used for the MODLEVEL micro; str consists of 1 through 7 letters or digits.

OPT Optimization Level

The OPT parameter specifies the level of optimization performed by the compiler. Valid options are:

omitted

Same as OPT=0.

OPT

Same as OPT=2.

OPT=0

Minimum optimization is performed, resulting in fastest compilation. OPT=0 is required for DB=ID.

OPT=1

Intermediate optimization is performed.

OPT=2

High optimization is performed, resulting in slower compilation.

OPT=3

Potentially unsafe optimizations in addition to all OPT=2 optimizations are performed.

In optimizing mode, optimizations can be performed in two ways: by the compiler and by the user. User optimization includes not only the standard methods that represent good programming practice, but also certain specific methods that enable the compiler to optimize more effectively. Source code optimization and object code optimization are discussed in the following paragraphs:

OPT=0 Compilation

In the OPT=0 compilation mode, compile time evaluations are made of constant subexpressions; redundant instructions and expressions within a statement are eliminated.

OPT=1 Compilation

In the OPT=1 compilation mode, the following optimizations take place in addition to those in OPT=0:

1. Redundant instructions and expressions within a sequence of statements are eliminated.
2. PERT critical path scheduling is done to utilize the multiple functional units efficiently.

3. Subscript calculations are simplified, and values of simple integer variables are stored in machine registers throughout loop execution, for innermost loops satisfying all of the following conditions:

Having no entries other than by normal entry at the beginning of the loop

Having no exits other than by normal termination at the end of the loop

Having no external references (user function references or subroutine calls; input/output, STOP, or PAUSE statement; or intrinsic function references) in the loop

Having no IF or GOTO statement in the loop branching backward to a statement appearing previously in the loop

OPT=2 Compilation

In the OPT=2 compilation mode, the compiler collects information about the program unit as a whole and the following optimizations are attempted in addition to those in both OPT=0 and OPT=1:

1. Values of simple variables are not retained when they are not referenced by succeeding statements.
2. Invariant (loop-independent) subexpressions are evaluated prior to entering the loops containing them.
3. For all loops, the evaluation of subscript expressions containing a recursively defined integer variable (such as I when I=I+1 appears within the loop) is reduced from multiplication to addition.
4. Array addresses, values of simple variables in central memory, and subscript expressions are stored in machine registers throughout loop execution for all loops.
5. In all loops and in complicated sections of straight-line code, array references and subscript values are stored in machine registers.
6. In small loops, indexed array references are prefetched after safety checks are made to ensure that the base address of the array and its increment are reasonable and should not cause an out-of-bounds reference (mode 1 error).

OPT=3 Compilation

In OPT=3 compilation mode, the compiler performs certain optimizations which are potentially unsafe. The following optimizations are performed in addition to those provided by OPT=2:

1. In small loops, indexed array references are prefetched unconditionally without any safety checks. See figure 11-2 for an example.

```
REAL B(100,100)
.
.
.
DO 20 I = 1,100,10
20 S = S + B(J,I)
```

When the compiler prefetches the reference to B, the last reference to B in the loop is B(J,101) which might cause an out-of-bounds error at execution time if the array B is stored near the end of the field length.

Figure 11-2. Possible OPT=3 Error Example 1

2. When an intrinsic function is referenced, the compiler assumes that the contents of certain B registers are preserved for use following the function processing. See figure 11-3 for an example.

```
REAL A(10),C(10)
.
.
.
DO 10 I=1,N
10 C(I) = EXP(A(I))
```

The compiler might assign I and N to B registers during the loop.

Figure 11-3. Possible OPT=3 Error Example 2

In a loop, the registers available for assignment are determined by the presence or absence of external references. External references are user function references and subroutine calls, input/output statements, and intrinsic functions (SIN, COS, SQRT, EXP, and so forth).

When OPT=3 is not selected, the compiler assumes that any external reference modifies all the registers; therefore, it does not expect any register contents to be preserved across function calls.

If a math library other than the FORTRAN Common Library is used at an installation to supply intrinsic functions, the B register portion of the OPT=3 option must be deactivated by an installation option in order to ensure correct object code.

Source Code Optimization

To achieve maximum object code optimization regardless of optimization level, the user should observe the following practices for programming source code:

1. Since arrays are stored in column major order, DO loops (including implied DO loops in input/output lists) which manipulate multidimensional arrays should be nested so that the range of the DO loop indexing over the first subscript is executed first. Implied DO loop increments should be one whenever possible. See figure 11-4 for an 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)

```

Figure 11-4. Optimization Example 1

2. The number of different variable names in subscript expressions should be minimized. For example:

```
X = A(I+1,I-1) + A(I-1,I+1)
```

is more efficient than:

```

IPI = I+1
IMI = I-1
X = A(IPI,IMI) + A(IMI,IPI)

```

- The use of EQUIVALENCE statements should be avoided, especially those including simple variables and arrays in the same equivalence class.
- Common blocks should not be used as a scratch storage area for simple variables.
- Program logic should be kept simple and straightforward; program unit length should be less than about 600 executable statements.
- The use of dummy arguments (formal parameters) and variable dimensions should be avoided if possible; common or local variables should be used instead.

PD Print Density

The PD parameter specifies print density for all printable output (L and E files). The destination printer must be capable of supporting the specified density. For interactive connected files, PD options are suppressed. Valid options are:

omitted
Same as job default.

PD
Same as PD=8.

PD=6
Compiler output is printed at six lines per inch, single spaced.

PD=8
Compiler output is printed at eight lines per inch, single spaced.

PL Print Limit

The PL parameter specifies the maximum number of records (print lines) that the executing program can write to file OUTPUT. This parameter is opera-

tive only when appearing on an FTN5 statement used to compile a main program. Valid options are:

omitted
Same as PL=5000.

PL
Same as PL=50000.

PL=n
Output must not exceed n lines; n is a decimal integer consisting of one through nine digits for NOS/BE and SCOPE 2 or one through seven digits for NOS.

PN Pagnation

The PN parameter specifies page numbering options for the compiler output listing. Valid options are:

omitted
Same as PN=0.

PN
Page numbering is continuous from program unit to program unit, including intermixed COMPASS output.

PN=0
Page numbers begin at 1 for each program unit.

PS Page Size

The PS parameter specifies the number of lines to be included on a printed page of the output listing. Valid options are:

omitted
Same as job default.

PS=n
Specifies the maximum number of printed lines; n must not be less than 4.

PW Page Width

The PW parameter specifies the width of an output line. Valid options are:

omitted
For a connected listing (L) or error (E) file, same as PW=72. For all other output files, same as job default.

PW
Same as PW=72.

PW=n
Printed lines are to contain n characters; n is a decimal integer and must not be less

than 50 or greater than 136. Lines shorter than 136 characters are reformatted rather than truncated, as described later in this section.

QC Quick Syntax Check

The QC parameter specifies that the compiler is to perform a quick syntax check of the source program. Valid options are:

omitted

Same as QC=0.

QC

The compiler performs a full syntactic scan of the program, but no binary code is produced. No code addresses are provided if a reference map is requested. QC compilation is substantially faster than normal compilation; but it must not be selected if the program is to be executed.

QC=0

Quick syntax check is not performed.

The QC option conflicts with the B, GO, and LO=O/M options.

REW Rewind Files

The REW parameter specifies the files to be rewound prior to compilation. Valid options are:

omitted

Same as REW=0.

REW

Same as REW=I/B.

REW=0

No files are rewound.

REW=op [/op]...

where op is one of the following:

I - Rewinds the input file (specified by the I parameter).

E - Rewinds the error file (specified by the E parameter).

B - Rewinds the binary output file (specified by the B parameter).

L - Rewinds the output file (specified by the L parameter).

Initial value is REW=I/B/E.

ROUND Rounded Arithmetic Computations

The ROUND parameter specifies which arithmetic operations are to be performed using rounded arithmetic. This parameter controls only the in-line

object code compiled for arithmetic expressions; it does not affect computations performed by library subroutines, library intrinsic functions, or library input/output routines.

The intrinsic functions that are not contained in the library are affected by compile statement options. Nothing in the library will be affected by compile statement options because the library and compiler function separately. Valid options are:

omitted

Same as ROUND=A/S/M.

ROUND

Same as ROUND=A/S/M/D.

ROUND=0

No rounding is performed.

ROUND=op [/op]...

where op is one of the following:

A All addition operations are rounded.

S All subtraction operations are rounded.

M All multiplication operations are rounded.

D All division operations are rounded.

Initial value is ROUND=0.

S System Text File

The S parameter specifies the name of the system text to be read by the compiler. Valid options are:

omitted

Same as S=SYSTEXT if G parameter is not specified.

Same as S=0 if G parameter is specified.

S

Same as S=SYSTEXT if G parameter is not specified.

Same as S=0 if G parameter is specified.

S=0

System text file is not loaded when COMPASS is called to assemble any intermixed COMPASS subprograms.

S=sname

Specifies the system text name to be sname and searches the global library set.

S=lib-sname

Searches the library named lib for the system text named sname. (The hyphen separating lib and sname is required.)

Multiple names can be specified by separating them with slashes, up to a maximum of seven names. Multiple occurrences of this parameter are permitted. An S=0 specification is ignored if any other S option is specified.

SEQ Sequenced Input

The SEQ parameter specifies source file sequencing format. Valid options are:

omitted

Same as SEQ=0.

SEQ

The source input file is in sequenced line format.

SEQ=0

The source input file is in standard FORTRAN format.

STATIC Static Load

The STATIC parameter specifies static inclusion of file buffers. Valid options are:

omitted

Same as STATIC=0.

STATIC

Inhibits dynamic file allocation at execution time by run-time library. Required library programs must be selected by calls to the STLxxx routines described in section 7.

STATIC=0

Use of dynamic memory management at execution time by run-time library.

TM Target Machine

The TM option specifies attributes of the object time machine. This parameter is an installation option. (Not available on SCOPE 2.) Valid options are:

omitted

Attributes of the object time machine are assumed to be identical to those of the compile time machine.

TM

Same as TM=0.

TM=0

Object time machine is assumed to have none of the possible attributes; for example, no LCM.

TM=LCM

Object time machine is assumed to have large central memory available.

Object code generated according to the TM option will execute on another machine, but the code will not be optimized for that machine.

X External Text Name

The X parameter specifies the name of the file from which the COMPASS assembler reads the external text when it encounters an XTEXT directive in the intermixed COMPASS program. Valid options are:

omitted

Same as X=OLDPL.

X

Same as X=OPL.

X=lfm

COMPASS assembler reads external text from file lfm.

The X parameter is intended for use with intermixed COMPASS subprograms only.

FTNS CONTROL STATEMENT EXAMPLES

Some examples of FTNS control statements are as follows:

Example 1:

FTNS (ET=F,EL=F,GO,L=SEE,LO=M/R,S=0)

selects the following options:

ET=F

On fatal compilation errors, skips to an EXIT (NOS) or EXIT,S (NOS/BE and SCOPE 2) control statement.

EL=F

Fatal diagnostics only are listed.

GO

Generated binary object file is loaded and executed at end of successful compilation.

L=SEE

Listed output appears on file SEE.

LO=M/R

Reference map and storage map are listed in addition to source listing and attributes list.

S=0

When COMPASS is called to assemble an intermixed COMPASS subprogram, it does not read in a system text file.

Example 2:

FTN5 (GO,DB=ID)

selects the following options:

GO

The program is loaded and executed after successful compilation.

DB=ID

Tables are generated for use by CYBER Interactive Debug. Second defaults are selected for other DB options.

Example 3:

FTN5.

selects default options. Refer to table 11-1 for a summary of the default options.

COMPILER LISTINGS

The listings produced by FORTRAN during compilation are determined by control statement parameters. The types of listings produced and the control statement parameters that influence them are as follows:

Source listing

Includes all source lines submitted for compilation as part of the source input file. The C\$ LIST(S) directive can be used to suppress the listing of selected source lines. Listed lines are preceded by a line number. Information contained in the source listing is determined by the LO parameter.

Diagnostics

Includes informative, note, warning, ANSI, fatal diagnostics, and catastrophic as determined by the EL and ANSI parameters (appendix B). Catastrophic diagnostics cannot be suppressed. Diagnostics appear immediately after the source line where they were detected. (Some declarative processor diagnostics appear at the end of the declarative statements.)

Object code

Includes generated object code, listed as COMPASS assembly language instructions. Selected by the LO=0 option.

Reference map

Includes compiler assigned locations, as well as other attributes, of all symbolic names, statement labels, and other program entities in each program unit. Contents are determined by the LO parameter.

Optimizer statistics

Includes a summary of optimizations performed by the compiler. Optimizations are determined by the OPT parameter.

Statistics

Includes program field length and CPU seconds used for compilation.

A header line at the top of each page of compiler output contains the program unit type and name, the computer used for compilation, and the target computer for which the code is being compiled, some of the control statement options, compiler version and mod-level, date, time, and page number. The source program is listed at 60 lines per page (including headers) unless a different value is specified by the PS parameter.

The output listing contains certain information about control statement parameters. The information is contained on the first three lines of the output listing.

Title line one contains the ROUND and the directive suppression parameter values for the compilation.

Title line two contains the DO loop control option, the argument list attributes, the collating sequence, the debugging options, and the print limit selection.

Title line three contains the FTN5 control statement up to a maximum size of 120 characters.

All three lines appear on the first page of each program unit. Subsequent pages show only the first line. A control statement binary value parameter, within the title lines, is indicated as ON if preceded by a blank and as OFF if preceded by a minus sign.

The files to which listings are written is determined by the L and E control statement parameters.

SHORT LINE LISTING

When the page width specified by the PW parameter on the FTN5 control statement is less than 132, the output listing is reformatted so that source statements and error messages fit in a line of the specified width. Source statements are broken at the maximum line length and are resumed in the tenth printed column of the following line with >>>> appearing in columns three through six. Error messages are broken at the nearest blank and are resumed in the same manner as source lines.

When the compiler output listing file is connected to a terminal, the default for the PW parameter is the terminal line length. When the terminal line length is not determinable, a length of 72 characters is assumed.

If the PW value is equal to or greater than 126, the page header occupies one line. If the PW value is less than 126, the header is reformatted into two lines. In this case, the subtitle line is suppressed. Note that PW cannot be less than 50.

LISTING CONTROL DIRECTIVE

The C\$ LIST comment directive provides control over the listings produced by the LO (listing options) parameter selected on the FTN5 control statement. The C\$ LIST directive is described in appendix E.

REFERENCE MAP

The reference map is a dictionary of all programmer-created symbols appearing in a program unit. The symbol names are grouped by class and listed alphabetically within the groups. The reference map follows the source listing of the program and the diagnostics (if present), and precedes the object listing (if present).

The kind of reference map produced is determined by the LO parameter on the FTN5 control statement. The applicable reference map options are as follows:

- A A list of program entities (variables) and their attributes (data type, class, and so forth) is written to the output file.
- M A map, showing the correlation of program entities and physical storage, is written to the output file.
- R A cross-reference map is written to the output file.

The initial values set for the LO parameter are as follows:

- omitted
Selects S/A.
- LO
Selects S/A/R.
- LO=opt/...
Selects S/A as initial values. All options then selected are added.
- LO=0
Deselects all listing options.
- LO=0/opt/...
Deselects all listing options, then selects the specified options.

Examples:

- LO=A Selects S/A.
- LO=R Selects S/A/R.
- LO=0/R Selects R.
- LO=M Selects S/A/M.
- LO=0/S Selects S.
- LO=S Selects S/A.

L=0 forces LO=0, but LO=0 has no effect on L.

Fatal errors in the source program cause certain parts of the map to be suppressed; parts of the map might also be incomplete, or inaccurate. Fatal to execution (F) and fatal to compilation (C) errors cause the DO loop map to be suppressed, and assigned addresses will be different; symbol references might not be accumulated for statements containing syntax errors.

GENERAL FORMAT OF MAPS

Each class of symbol is preceded by a subtitle line that specifies the class and the properties listed. Formats for each symbol class are different, but printouts contain the following information:

The octal address associated with each symbol relative to the origin of the program unit or common block. All addresses will print as blank if QC is selected.

Properties associated with the symbol.

List of references to the symbol (for LO=R only).

All line numbers in the reference list refer to the line of the statement in which the reference occurs.

All numbers to the right of the name are decimal integers unless they are printed in one of two forms; 0"... " or ...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.

The following subsections describe the various sections of the reference map as they would appear for the full map, selected by LO=M/A/R. The sections that appear for a given option are indicated.

Variable Map

Variable names include local and COMMON variables and arrays, dummy arguments and, for FUNCTION subprograms, the defined function name when used as a variable. Figure 11-5 shows the variable map format.

Symbolic Constant Map

A symbolic constant is declared in a PARAMETER statement. The format of the symbolic constant map is shown in figure 11-6.

Procedure Map

Procedures include names of functions or subroutines called explicitly from a program or subprogram, names declared in an EXTERNAL statement, and names of intrinsic and statement functions appearing in the subprogram. Implicit external references, such as calls by certain FORTRAN source statements (READ, ENCODE, etc.) are not listed. The format of the procedure map is shown in figure 11-7.

-VARIABLE MAP-(LO=A/M/R)

NAME-ADDRESS-BLOCK-PROPERTIES-TYPE-SIZE-REFERENCES

name	addr	block	prop1/prop2	type	size	refs
name	Variable name as it appears in FORTRAN source listing. Variables are listed in alphabetical order.					
addr	Relative address assigned to variable name. If name is a member of a COMMON block, addr is relative to the start of block. If name is a non-SAVED local variable, then addr is program relative. If name is a SAVED local variable, then addr is relative to the /\$\$A\$V\$E/ block for this program unit. If name is a dummy argument, then addr is the offset into the program unit composite formal parameter list. (Not necessarily the same as its position in a source program dummy argument list.) If addr is given as NONE, then the compiler (especially OPT=2) has determined that name does not need to be stored in memory.					
block	Name of COMMON block in which variable name appears. If blank, name is a local variable. // Indicates name is in blank COMMON. DUMMY-ARG Indicates variable name is a dummy argument (formal parameter) to this subprogram. STF-DARG Indicates variable name is used only as a statement function dummy argument in current program unit.					

The following are obtained only with the LO=A or LO=M option:

prop	Properties associated with variable name; indicated by the following keywords (listed in the format prop1/prop2. . .):					
UND	Variable name has not been defined. A variable is defined if any of the following conditions hold: Appears in a COMMON or DATA statement. Is equivalenced to a variable that is defined. Appears on the left side of an assignment statement at the outermost parenthesis level. Is the index variable in a DO loop. Appears as a stand-alone actual parameter in a subroutine or function call. Appears in an input list (READ, BUFFERIN, etc.). Otherwise, the variable is considered undefined; however, variables which are used (in arithmetic expressions, etc.) before they are defined (by an assignment statement or subprogram call) are not flagged.					
EQV	Variable name is equivalenced.					
LEVn	Variable name is given a LEVEL due to the source program.					
SAV	Variable name has the SAVE property.					
UNUSED	Name appears only in dummy argument list(s) and/or in a nondimensioning type statement.					
S	Name appears only once in the entire program unit. The user should check carefully for other names with similar spellings. A name will not be flagged as *S*(STRAY) nor UNUSED if it is in COMMON, is a DO loop control index, or is used as a subroutine, function, or entry.					
type	Gives the mode associated with the variable name. LOGICAL, INTEGER, REAL, COMPLEX, DOUBLE, CHARACTER or BOOLEAN. In the case of CHARACTER, the form is: CHAR*n For specified length CHAR*(*) For adjustable length					
size	Number of elements of name, when name is dimensioned. For nonarray names, this field is blank. The size is given by UB-LB+1, where UB = upper bound, LB = lower bound. For adjustable dimensions ADJARY. is given.					

Figure 11-5. Variable Map (Sheet 1 of 2)

refs	(appear only with R option) References and definitions associated with variable name; listed by line number. Certain references are followed by a usage suffix, chosen from:
/A	Argument: pass by reference actual argument in a subroutine or user-function call.
/C	Control: DO statement where name is the loop control variable.
/I	Initialized: in a DATA statement.
/R	Read: name appears as an input list item of a READ or ENCODE; as an internal file identifier of a WRITE, or DECODE statement; or as a BUFFER IN limit.
/S	Store: name appears as a store target in an assignment or ASSIGN statement; or receives a value as an IOSTAT= specifier or in an INQUIRE statement.
/U	Unit: used as an I/O unit designator, except an internal file designator.
/W	Write: value of name is written, by appearing as an output list item of a PRINT, PUNCH, or WRITE; or internal file identifier of READ or ENCODE; or as a BUFFER OUT limit.

Figure 11-5. Variable Map (Sheet 2 of 2)

SYMBOLIC CONSTANTS (LO-A/M/R)			
NAME	TYPE	VALUE	REFERENCES
name	type	value	refs
name		Symbol name as it appears in the source. Listed in alphabetic order.	
type		Same as type field in VARIABLES section.	
value		Value assigned to name. Format depends on type:	
	LOGICAL		Either .TRUE. or .FALSE.
	BOOLEAN		O'nnn'
	REAL, DOUBLE		O'nnn'
	COMPLEX		Real half listed as O'nnn'.
	INTEGER		Integer value if magnitude less than 10000000000, otherwise O'nnn'. Leading minus if < 0.
	CHARACTER		Enclosed in quotes. If value does not fit in the columns provided, then the trailing quote is replaced by an ellipsis (. . .).
refs		Source line number where referenced. Suffix /S indicates definition line. Appears only when R option is specified.	

Figure 11-6. Symbolic Constants Map

PROCEDURES (LO-A/M/R)

NAME TYPE ARGS CLASS REFERENCES

name type arg class refs

The following are obtainable only with the LO-A or M option:

name	Symbol name as it appears in the source listing.			
type	Gives the result mode for a function. One of the following: Blank if class is SUBROUTINE or EXTERNAL. GENERIC if appropriate. Otherwise, one of the type designators listed in VARIABLES section.			
args	Number of arguments. If the number is variable (MAX, MIN, etc.), VAR is given as number of arguments. UNKNOWN if external.			
class	One of the following: SUBROUTINE DUMMY SUBR Dummy argument subroutine INTRINSIC STAT FUNC Statement function FUNCTION Nonintrinsic external function DUMMY FUNC Dummy argument function EXTERNAL None of the above			
refs	(with LO-R option only). Line number on which name is referenced. Reference might be suffixed with: /D Declarative statement, or definition line of a statement function. /A Argument pass by reference actual argument in a subroutine or user-function call.			

Figure 11-2. Procedures Map

Statement Label Map

The statement label map includes all statement labels defined in the program or subprogram. The format of the statement label map is shown in figure 11-8.

Entry Point Map

Entry point names include program and subprogram names and names appearing on ENTRY statements. The format of the entry point map is shown in figure 11-9.

Input/Output Unit Map

The input/output unit map includes constant UNIT designators. Standard or extended internal files are not included. The format of the input/output unit map is shown in figure 11-10.

NAMELIST Map

The namelist map contains the names of the namelist groups defined in the program unit. The format of the namelist map is shown in figure 11-11.

DO Loop Map

The DO loop map includes all DO loops that appear in the program unit, including implied DO loops not in DATA statements, and lists their properties. This map is suppressed if fatal errors have been detected in the program unit or if QC was specified on the PTN5 control statement. Loops are listed in order of appearance in the program. This map appears only when LO-M is selected. The format of the DO loop map is shown in figure 11-12.

Common and Equivalence Map

The common and equivalence map shows the storage layout for common blocks, and the equivalence-induced storage overlap for all variables. This map appears only when LO-M is selected. It is suppressed if fatal errors are detected or if QC was specified on the PTN5 control statement. The format of the common and equivalence map is shown in figure 11-13.

Equivalence-induced storage overlap classes are indicated by enclosing parentheses: the first item in an equivalence class is preceded by a left parenthesis, and the last item is followed by a right parenthesis. Entries under LOCAL EQUIVALENCE include items that are not declared in a COMMON statement.

-STATEMENT LABELS-(LO=A/M/R)

LABEL---ADDRESS---PROPERTY---DEF---REFERENCES

label	addr	prop	def	refs
label				Statement label from FORTRAN source program. Statement labels are listed in numerical order.
addr				Program-relative address assigned to statement label. When no meaningful address can be assigned, one of the following flags will appear: *UNDEF* Statement label is not defined; refs lists all occurrences of the undefined label. Undefined labels also generate a diagnostic. *NO REFS* label is not referenced by any statements. This label can be safely removed from the FORTRAN source program. INACTIVE label has been deleted by optimization. blank no address is available usually due to source program fatal error, or OC option.
prop				One of the following: FORMAT Statement label is a FORMAT. DO-TERM Statement label appeared in a DO statement. NON-EX Label appeared on a nonexecutable statement. If addr is not *NO REFS*, then the program is incorrect, and a diagnostic will have been issued. blank Label is a normal control label.
def				Source line number where label was defined. *UNDEF* if not defined.
refs				Line numbers on which label was referenced. (Appears only with LO=R option.) Usage suffixes are as follows: /A Assign statement /D DO statement /R Input or DECODE format /W Output or ENCODE format

Figure 11-8. Statement Label Map

-ENTRY POINTS-(LO=A/M/R)

NAME---ADDRESS---ARGS---REFERENCES

name	addr	args	refs	
name				Entry point name as defined in FORTRAN source.
addr				Relative address assigned to the entry point.
args				Number of dummy arguments for entry name.
refs				In subprograms only, line number of RETURN statements and ENTRY definition. If line number is followed by /D, this implies the line number on which the name is defined. All RETURN statement refs are to the main entry point. Appears only on R maps.

Figure 11-9. Entry Point Map

-I/O UNITS-(LO-A/M/R)

NAME—PROPERTIES—REFERENCES

name prop1/prop2 refs

name	The value of the constant UNIT designator. If the value is an integer, $0 \leq \text{val} \leq 999$, then name is TAPE n .	
prop	Properties listed include only the ones detectable in the current program-unit:	
	FMT	A formatted operation appeared.
	BIN	A nonformatted operation appeared.
	DIR	A direct access operation.
	SEQ	A sequential operation.
	BUF	A BUFFER IN or BUFFER OUT statement.
	AUX	An auxiliary I/O statement.
refs	Appears only with R option. Source line number of statements which explicitly refer to unit name. Usage suffixes are:	
	/R	Read operation.
	/W	Write operation, including ENDFILE.
	blank	OPEN, CLOSE, BACKSPACE, REWIND, INQUIRE.

Figure 11-10. Input/output Unit Map

NAMELISTS-(LO-A/M/R)

NAME—ADDRESS—REFERENCES

name addr refs

name	Namelist group name as defined in FORTRAN source.	
addr	Relative address assigned to name.	
refs	Line numbers of references to name (with LO-R option only). Line number will be followed by /D, /R, /W. /D = line number on which namelist is defined. /R = line number on which namelist appears in an input operation. /W = line number on which namelist appears in an output operation.	

Figure 11-11. Namelist Map

-DO-LOOPS-

LABEL—ADDRESS—INDEX—PROPERTIES—FROM—TO

label addr index prop first-last

label	Statement label defined as end of loop, or I/O for implied DO loops in I/O statements.			
addr	Relative address assigned to the start of the loop body.			
index	Variable name used as control index for loop, as defined by DO statement.			
prop	Various keywords can appear, describing optimization properties of the loop:			
	XREF	Loop not optimized because it contains references to an external subprogram (including compiler-generated references to library routines).		
	OPEN	Loop not optimized because it can be reentered from outside its range.		
	OUTER	Loop not optimized because other loops are contained inside it.		
	EXIT	Loop not optimized because it contains references to statement labels outside its range.		
first-last	Line numbers of the first and last statements of the loop.			

Figure 11-12. DO Loop Map

--COMMON+EQUIVALENCE--(LO=M/A/R)

/block/ LEVEL lev, SIZE = size units sav.

item [item] . . .

item [item] . . .

·
·
·

LOCAL EQUIVALENCE

item [item] . . .

item [item] . . .

·
·
·

block Name of the common block being described.

lev Storage level of the block (1, 2, or 3).

size Total number of storage units occupied by the block.

units CHARS for a block containing character variables.
WORDS for a block with noncharacter variables.

sav SAVE if the block is saved, otherwise blank.

item Describes the storage position of a variable or array. Each item consists of three fields:

name first : last

name Symbolic name of the item.

first Number of the storage unit occupied by the first element of name.

last Number of the storage unit occupied by the last element of the name.

First and last are given in decimal. The first position in a block is numbered one. For a 2-word scalar, last = first + 1. The item descriptors are printed left-to-right in order by ascending first then descending last. When name is of type character, first and last are given in character storage units, and they are separated by a colon. Otherwise, they are in words, separated by a dash.

Figure 11-13. Common Equivalence Map

Spray Names

If a program contains items with questionable or illegal attributes, the reference map will specify the following attributes:

STRAY

Indicates variable names that appear only once in the entire program unit.

NO-REFS

Indicates statement labels that are not referenced by any statement in the program unit.

UNDEF

Indicates statement labels that are referenced but are not defined in the program unit.

Program Statistics

At the end of each program unit, statistics are printed in octal and decimal. The format of the statistics map is shown in figure 11-14.

If LO=A is not specified, only the diagnostic counts are printed.

DEBUGGING USING THE REFERENCE MAP

When debugging a new program, the reference map can be used to find names that have been punched incorrectly as well as other items that will not show up as compilation errors. The basic technique consists of using the compiler as a verifier and correcting the fatal errors until the program compiles.

STATISTICS

PROGRAM-UNIT LENGTH	Length of program including code, storage for local variables, arrays, constants, temporaries, etc., but excluding buffers and common blocks.
CM STORAGE USED	Maximum memory used during the compilation.
COMPILE TIME	Compilation time of subprogram unit.
nmm { ANSI TRIVIAL WARNING FATAL }	ERROR(S) IN program

Figure 11-14. Program Statistics Map

Using the listing, the LO-R/M reference map, and the original flowcharts, the following information should be checked by the programmer:

Names Incorrectly punched

Stray name flag in the variable map

Functions that should be arrays

Functions that should be in-line instead of external

Variables or functions with incorrect type

Unreferenced format statements

Unused formal parameters

Ordering of members in common blocks

Equivalence classes

When debugging a program, the reference map can be used to understand the structure of the program. Questions concerning the loop structure, external references, common blocks, arrays, equivalence classes, input/output operations, and so forth, can be answered by checking the reference map.

The sample program shown in figure 11-15 is compiled with LO=A/M/R to produce a full reference map. The reference map is shown in figure 11-16.

OBJECT LISTING

The structure of the object code produced by FORTRAN differs depending on which optimization mode is selected (OPT=0, 1, 2, 3 control statement option).

The FORTRAN compiler produces object code in units called USE blocks. (See the COMPASS reference manual.) These blocks include not only the code produced by compilation of the executable statements in the user's program, but also storage for variables, constants, and compiler-generated temporary entities, as well as other special purpose areas.

Also discussed in this section is the arrangement in memory of user code, library routines, and common blocks after the program is loaded.

NOTE

The information in this subsection is intended only as a guide to interpret object code listings and core dumps. The information does not represent a guaranteed interface specification. Users should avoid any programming techniques which depend on a detailed knowledge of compiler-generated code.

The following description of the arrangement of code and data within main programs, subroutines, and functions does not include the arrangement of data within common blocks because this arrangement is specified by the programmer. However, the diagram of a typical memory layout (in section 9) illustrates the position of blank common and labeled common blocks.

```

PROGRAM MAPS
CHARACTER*4 FILE
INTEGER SIZE1,S1,SIZE2,S2,STRAY
EQUIVALENCE (SIZE1,S1), (SIZE2,S2)
NAMelist /PARAMS/ SIZE1,SIZE2
DATA S1,S2 /12,12/

C
OPEN(3,FILE='INPUT')
OPEN(2,FILE='OUTPUT')
100 READ(3,PARAMS)
WRITE(2,PARAMS)
WRITE(2, ' (" SAMPLE PROGRAM TO ILLUSTRATE COMPILER MAPS")')
CALL PASCAL(S1)
WRITE(2, ' (" THE FOLLOWING WILL HAVE NO HEADING")')
CALL NOHEAD(S2)
STOP
END

C
BLOCK DATA
COMMON /ANARRAY/ X(22)
INTEGER X
DATA X(22) /1/
END

C
SUBROUTINE PASCAL(SIZE)
INTEGER L(22), SIZE
COMMON /ANARRAY/ L

C
WRITE(2, ' (" PASCALS TRIANGLE")')
ENTRY NOHEAD
M = MIN0(21,MAX0(2,SIZE-1))
DO 2 K=21,22-M,-1
L(K) = 1
DO 1 J=K,21
1 L(J) = L(J) + L(J+1)
2 WRITE(2, '(1X,22I6)') (L(J),J=K,22)
RETURN
END

```

Figure 11-15. Program MAPS

```

PROGRAM MAPS (INPUT, OUTPUT, TAPE3) INPUT, TAPE2=OUTPUT
CHARACTER*4 FILE
INTEGER SIZE1, S1, SIZE2, S2, STRAY
EQUIVALENCE (SIZE1, S1), (SIZE2, S2)
NAMELIST /PARAMS/ SIZE1, SIZE2
DATA S1, S2 /12, 12/
C
100 READ(3,PARAMS)
WRITE(2,PARAMS)
WRITE(2,*) SAMPLE PROGRAM TO ILLUSTRATE COMPILER MAPS(*)
CALL PASCAL(S1)
WRITE(2,*) THE FOLLOWING WILL HAVE NO HEADING(*)
CALL MDHEAD(S2)
STOP
END
    
```

--VARIABLE MAP--(LD=M/A/R)

NAME	ADDRESS	BLOCK	PROPERTIES	TYPE	SIZE	REFERENCES
FILE	NCNE		UNUSED/**	CHAR**		2
SIZE1	106B		EQV	INTEGER	4	3
SIZE2	107B		EQV	INTEGER	4	3
STRAY	NDNE		UNUSED/**	INTEGER	3	3
S1	106B		EQV	INTEGER	4	6/I
S2	107B		EQV	INTEGER	4	6/I

A=ARGLIST, C=CTRL OF DD, I=DATA INTR,
R=READ, S=STORE, U=I/O UNIT, W=WRITE

--PROCEDURES--(LD=M/A/R)

NAME	TYPE	ARGS	CLASS	REFERENCES
NCHEAD	1	SUBROUTINE	13	
PASCAL	1	SUBROUTINE	11	

D=DEF LINE OF STMT FUNC
A=ACTUAL ARGUMENT

--STATEMENT LABELS--(LD=M/A/R)

NAME	ADDRESS	PROPERTIES	DEF	REFERENCES
100	ND REFS*		0	P/L

--ENTRY POINTS--(LD=M/A/R)

NAME	ADDRESS	ARGS	REFERENCES
MAPS	ZOR	C	1/E

--COMMON/EQUIVALENCE--(LD=M/A/R)

NAME	ADDRESS	REFERENCES
(S17F241)	S2<1>	(S17E141)

--LOCAL EQUIVALENCE--

NAME	ADDRESS	REFERENCES
(S17F241)	S2<1>	(S17E141)

Figure 11-16. Reference Map Example (Sheet 1 of 4)

R=READ, W=WRITE

--I/O UNITS--(LC=M/A/R)
--NAME-- PROPERTIES-----REFERENCES--

TAPE2 FMT/SEO 9/W 10/W 12/W
TAPE3 FMT/SEQ 8/R

--STATISTICS--

NR OF ALLOCS 132
TABLE CRASHES 4
LARGEST PARSED FILE LENGTH 338 * 27
PROGRAM-UNIT LENGTH 1308 = 72
SCM STORAGE USED 742008 = 30848
COMPILE TIME 0.147 SECONDS

BLOCKDATA BKDAT.
DD=LDNR/-OT,ARG=COMMON/FIXED,DR=IM7-39/-SL7 EX/-10/-PMO7-ST,PL=5000
F,LD=PK.

1 C
2 BLOCK DATA
3 COMMON /ANARRAY/ X(22)
4 INTEGER X
5 DATA X(22) /1/
6 END

--VARIABLE MAP--(LD=M/A/R)
--NAME-- ADDRESS--BLOCK--PROPERTIES--TYPE--SIZE--REFERENCES--
X 08 /ANARRAY/ INTEGER 22 3 4 5/1

--COMMON+EQUIVALENCE--(LD=M/A/R)

/ANARRAY/ LEVEL = 1, SIZE = 22 WORDS SCM
X(1)-22

--STATISTICS--

NR OF ALLOCS 33
TABLE CRASHES 1
LARGEST PARSED FILE LENGTH 68 = 6
PROGRAM-UNIT LENGTH 08 = 0
SCM LABELLED COMMON LENGTH 742008 = 30848
COMPILE TIME 0.040 SECONDS

Figure 11-16. Reference Map Example (Sheet 2 of 4)

73/176 DPT=0,ROUND= A7 S/ M7-0,=DS. FTN 5-1+592+TP 12/18/91 14-42-30
 SUBROUTINE PASCAL /-FIXED,CS= USER7-FIXED,DD=-TR7-SB7-ER7-ID7-PMD7-ST,PL=3000
 DC=-LENG7-DT,ARG=-COMMON /-FIXED,CS= USER7-FIXED,DD=-TR7-SB7-ER7-ID7-PMD7-ST,PL=3000
 EVALD=PTF.

```

1 C
2 SUBROUTINE PASCAL(SIZE)
3   INTEGER I(22), SIZE
4   COMMON /ANARRAY/ L
5
6   WRITE(2,*) 'PASCALS TRIANGLE(1)'
7   ENTRY NOHEAD
8   M = MIN(21,MAX(2,SIZE-1))
9   DO 2 K=21,2,-1
10    L(K) = 1
11    DO 1 J=K,21
12     L(J) = L(J) + L(J-1)
13  WRITE(2,*) 'L(1),L(2),L(3),L(4),L(5)'
14  RETURN
15  END
  
```

--VARIABLE MAP--(LD=M/A/R)
 --NAME--ADDRESS--BLOCK--PROPERTIES--TYPE--SIZE--REFERENCES--
 J 127B INTEGER 12 12 13/E 13
 K 125B INTEGER 9/C 10 13/C 13
 L 0B /ANARRAY/ 22 10/S 12 12 12/S 13/M
 M 125B INTEGER 8/S 9/C 12 12 12/S 13/M
 SIZE 1 DUMMY-ARG 1 1 N/A

--PROCEDURES--(LD=M/A/R)
 --NAME--TYPE--ARGS--CLASS--REFERENCES--
 MAXO INTEGER VAR INTRINSIC R/A
 MINO INTEGER VAR INTRINSIC B

--STATEMENT LABELS--(LD=M/A/R)
 --LABEL--ADDRESS--PROPERTIES--DEF--REFERENCES--
 1 INACTIVE DD-TERM 12 13/0 12/L
 2 INACTIVE DD-TERM 13 9/0 13/L

--ENTRY POINTS--(LD=M/A/R)
 --NAME--ADDRESS--ARGS--REFERENCES--
 NOHEAD 109B 0 7/0
 PASCAL 9B 1 2/0 14/R

ARGUMENT, CONTROL OF DD, I=DATA UNIT,
 R=READ, W=WRITE, U=VOID UNIT, W=WRITE

D=DEF LINE OF START RUNC
 A=ACTUAL ARGUMENT

A=ASSIGN STATE, D=DD STATE,
 R=READ, W=WRITE, L=LABEL

D=DEFINITION, R=RETURN

Figure 11-16. Reference Map Example (Sheet 3 of 4)

```

SUBROUTINE PASCAL
--NO LOCKS--(LO=M/A/R)
--LABEL--ADDRESS--PROPERTIES-----INDEX-----FROM-----TO
2 30B XREF/ENTER K 9 13
1 37B J 11 12
I/O 55B XREF J 13 13

```

```

--COMMON+EQUIVALENCE--(LO=M/A/R)

```

```

/ANARRAY/ LEVEL = 1, SIZE = 22 WORDS SCM
(1)-22

```

```

--I/O UNITS--(LO=M/A/R)
--NAME-- PROPERTIES-----REFERENCES-

```

```

TAPE2 FMT/SEQ 6/M 13/M

```

```

--STATISTICS--

```

```

NF OF ALLCS 198
TABLE CRASHES 4
LARGEST PARSED FILE LENGTH 1138 = 75
PROGRAM-UNIT LENGTH 1348 = 92
SCM LABELLED COMMON LENGTH 268 = 22
SCM STORAGE USED 742008 = 308*8
COMPILE TIME 0.212 SECONDS

```

```

***** MRR002 / / / / END OF LIST / / / /
***** HRR002 / / / / END OF LIST / / / /

```

P=READ, W=WRITE

Figure 11-16. Reference Map Example (Sheet 4 of 4)

PROGRAM UNIT STRUCTURE

The code within program units is arranged in the following blocks in the order given:

START.

A table of file names specified in the PROGRAM statement (main program). Code for primary entry point initialization and for saving AO (subprograms).

CODE.

Code generated by compiling executable statements and code for alternate entry points (ENTRY statement).

LITERL.

Contains the following subblocks:

CON. Storage for read-only constants.

FORMAT.

Storage for static FORMAT statements.

TEMPS.

Contains the following subblocks:

- ST. Statement temporary values.
- CT. Character statement, function argument temporary values.
- IT. Global optimization temporary values.
- OT. Scheduler temporary values.
- VD. Nonconstant formal arraybound expressions and formal character passed-length values.
- LC. Local copies of selected scalar formal parameter values (OPT=2 and 3 only).

APLIST.

Actual parameter lists for subprograms called.

IOAPL.

Actual parameter lists for I/O subroutines called (this format differs from APLIST). Array of character descriptors referred to in I/O actual parameter lists (CL sub-block).

NAMLST.

Execution time dimension descriptors (RD block) and descriptors for referenced NAMELIST groups.

VARS.

Storage for local variables and arrays not mentioned in a SAVE statement. Each local equivalence class of type CHARACTER begins on a word boundary. Also some compiler-generated temporary cells including:

- DI. DO loop variable increment values.
- DC. DO loop trip count values.

SUB.

Address substitution lists for formal parameters.

SUBO.

Instructions to be modified for level 0 formal parameter references.

NAMING CONVENTIONS

The names of some system-supplied entities are changed by the compiler to prevent ambiguity for the assembler.

Register Name Conflicts

Variable names which are identical to COMPASS register names can be a source of confusion to the user when reading COMPASS listings, although the assembler can differentiate between the two.

System-Supplied Procedure Names

The name of an intrinsic function called by value is suffixed with a decimal point. The entry point is the symbolic name of the intrinsic function and a decimal point suffix. Examples are: EXP., COS., and CSORT. The names of all intrinsic functions called by value appear in section 7. The functions in section 7 are not called by value if the name appears either in an EXTERNAL statement, or if the DB=TB option is specified on the FIN5 control statement, or if they are other than mathematical functions.

If the function name appears in an INTRINSIC statement, the entry point is the function name suffixed with an equal sign. Otherwise, the pass by name entry point is the function name with no suffix.

The subroutines listed in section 7 are called by reference.

Listing Format

The object code produced for each program unit is listed following the reference map (if any) for that program unit. The 0 parameter of the C\$ LIST directive (described in appendix E) controls the listing of lines of object code. Object code generated by source lines falling between C\$ LIST,0=0 and C\$ LIST,0=1 is not listed.

Optimization can cause code to move so that object code for other statements is listed, or code from the desired statements is not listed.

Certain information which can be obtained from the source listing or reference map is not reproduced in the object listing. This information includes:

- Storage allocation for variables and arrays
- Namelist group definitions
- Data initialization translations
- Loader directives

EXECUTION CONTROL STATEMENT

Optional parameters can be included on the control statement that calls into execution a program compiled by FORTRAN. This control statement is normally either the name of the file to which the binary object code was written (LGO is the default) or an EXECUTE statement specifying the name of the main entry point of the program (the name used on the PROGRAM statement or START, if the PROGRAM statement was omitted). The parameters that can be included on this control statement are of four kinds: file names, print limit specification (PL), user parameters, and Post Mortem Dump options.

FILE NAME SUBSTITUTION

FORTRAN 5 provides a method of substituting file names at execution time. File names declared on the PROGRAM statement are associated with files of the same name unless the user substitutes a different name. For example, with the PROGRAM statement:

```
PROGRAM TEST1(INPUT,OUTPUT,TAPE1,TAPE2)
```

the execution time file names would be:

INPUT

OUTPUT

TAPE1

TAPE2

Note that specification of file names on the PROGRAM statement is optional; the same file names would occur if the statement PROGRAM TEST1 were used.

However, file names on the PROGRAM statement can be changed for the execution of a program by substitution in the execution control statement. A one-to-one correspondence exists between parameters on this statement and parameters in the PROGRAM statement. For example, using the preceding PROGRAM statement, an execution control statement of the form:

```
LGO(, ,DATA,ANSW)
```

would cause the file DATA to be used for file name TAPE1, and file ANSW to be used for file name TAPE2. Files INPUT and OUTPUT are used for file names Input and Output.

If a file name in the PROGRAM statement is equivalent, the logical file name is the file to the right of the equals sign. A corresponding file name in the execution control statement is ignored. For example, using the PROGRAM statement:

```
PROGRAM TEST3(INPUT,OUTPUT,  
+TAPE1=OUTPUT,TAPE2,TAPE3)
```

and the execution control statement:

```
LGO(, ,DATA,ANSW)
```

would cause the file OUTPUT to be used for filename TAPE1, and file ANSW to be used for filename TAPE2. Files INPUT, OUTPUT, and TAPE3 are used for file names INPUT, OUTPUT, and TAPE3.

The user should not substitute the same file name for two different file names on the PROGRAM statement.

PRINT LIMIT SPECIFICATION

A parameter can be specified on the execution control statement to regulate the maximum number of records that can be written at execution time on the file OUTPUT. The *PL parameter has the same form as the PL parameter specified at compilation time on the FTN5 control statement. If specified on the execution control statement, the PL parameter overrides the value specified either explicitly or by default at compilation time.

The print limit parameter (specified either at compilation time or at execution time) is operative only on files with the name OUTPUT in the first word of their corresponding file information table. Thus, if a file name declared in the PROGRAM statement is superseded at execution time by the file name OUTPUT as described previously, the print limit parameter will be operative on the original file name. Conversely, if the file name OUTPUT is superseded at execution time by another file name, the effect of the print limit parameter is nullified. Some examples of *PL parameter usage are as follows:

```
LGO(*PL=2000)
```

```
EXECUTE(,FILE1,OUTPUT,FILE2,*PL=1000)
```

USER PARAMETERS

User parameters that specify values to be accessible from the user program can be included on the execution control statement. These parameters must appear after any file names, *PL specification, and PMD parameters. The format of a user parameter is as follows:

name =value

name Parameter name consisting of 1 through 7 numbers or letters.

value Optional parameter value; string consisting of numbers, letters, or the character *. Can also be a literal, containing any character, delimited by dollar signs (\$). A dollar sign is denoted by two such signs in a row.

An example of an execution control statement containing user parameters is as follows:

```
LGO (INFILE,*PL=1000,V1=25,V2=ABCD)
```

```
LGO (X=$@@$)
```

```
LGO (Y=$AB$$CD$)
```

Parameters on the execution control statement must correspond with the parameters on the PROGRAM statement. For example, if the PROGRAM statement appears

```
PROGRAM SUN (INPUT,OUTPUT,TAPE1)
```

the execution call statement must be

```
LGO,,,MYPRAM=ABCE.
```

User parameters can be accessed from the FORTRAN program using the GETPARM subroutine call described in section 7. User parameters can also be specified by using continuation statements. (User parameters are the only kind of parameters allowed on continuation statements.) Filename, print limit specification, and post mortem dump parameters are not supported on continuation statements.

A continuation is specified by two periods. The continuation indicator is treated as a comma. The example above could also be specified as:

```
LGO..  
MYPRAM=ABCE.
```

This is equivalent since the file name parameters must appear on the first control statement. If they do not appear on the first control statement, they are defaulted as in the previous example.

If the job is interactive on NOS or NOS/BE and there are no more control statements in the job stream, the program will prompt for more parameters by:

```
...?
```

Note: The final terminator must be specified on the last statement.

There is no limit to the number of control statements specified.

(Under NOS, user parameters on the execution control statement must be in product set format. Refer to the NOS reference manual for information on product set format.)

POST MORTEM DUMP PARAMETERS

The Post Mortem Dump facility, described in section 10, provides a symbolic dump of program variables, an error analysis, and traceback information on the occurrence of a program abort. Parameters can be specified on the execution control statement to control Post Mortem Dump output and to specify limits on array subscripts.

Post Mortem Dump Output Parameter

The Post Mortem Dump output parameter specifies the destination and format of the dump. This parameter has the following format:

```
*OP=list
```

The option list consists of one or more of the following, not separated by separators:

A All active routines are included in the dump. An active routine is one that has been executed but is not necessarily in the traceback chain.

F Valid for interactive jobs only; Post Mortem Dump output is sent to file PMDUMP when the job is executed with file OUTPUT connected. The following message appears at the terminal at the time of the dump:

```
*POST PROCESSOR OUTPUT WILL BE  
FOUND ON THE FILE PMDUMP*
```

T Valid for interactive jobs only; a condensed form of the dump is sent to the terminal. File OUTPUT must be connected.

If the *OP parameter is omitted, dumps are sent to file PMDUMP when executing from a terminal with file OUTPUT connected. When the dump occurs, the following message appears at the terminal:

```
*POST PROCESSOR OUTPUT WILL BE FOUND ON  
THE FILE PMDUMP.*
```

```
*FOR A SUMMARY AT THE TERMINAL,
```

```
*RERUN JOB, REPLACING LGO CARD BY  
LGO,*OP=T.
```

An example of a Post Mortem Dump output control parameter is as follows:

```
LGO (*OP=AF)
```

Subscript Limit Specification

Subscript limits can be specified on the execution control statement to control the printing of arrays by Post Mortem Dump facility. This has the same effect as a PMDARRY call (section 10). The subscript limit parameter has the following format:

```
*DA=i+j+k+l+m+n+o
```

The integers i, j, k, l, m, n, and o specify the maximum values of the subscripts of arrays to be printed; i through o represent the first through seventh dimensions respectively. Subscript limits can be omitted from the list to control printing. If only i is specified, only 1-dimensional arrays are dumped; if i, j, and k are specified, then 1-, 2-, and 3-dimensional arrays are dumped and so forth, with 7-dimensional arrays dumped only when

parameters 1 through o are specified. For example, if the control statement:

```
LGO (*DA=2+5)
```

is used, only 1- and 2-dimensional arrays will be dumped. If the statement

```
DIMENSION ARAY(20,20)
```

appears in the source program, then the following elements will be printed by Post Mortem Dump:

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


The first part of this section contains sample deck structures, including control statements, illustrating compilation and execution of FORTRAN programs. The second part contains sample executable programs illustrating various features of FORTRAN. Examples of input and output are included.

SAMPLE DECK STRUCTURES

Following are some typical deck structures that can be used for compiling and executing FORTRAN programs. Refer to the operating system reference manual for details of control statements.

FORTRAN SOURCE PROGRAM WITH CONTROL STATEMENTS

Figure 12-1 shows a deck structure for compiling and executing a FORTRAN program that contains a function and a subroutine.

COMPILATION ONLY

Figure 12-2 shows a deck structure for compiling a program; the program is not executed after compilation.

OPT=0 COMPILATION

Figure 12-3 illustrates a deck structure for compiling a program in OPT=0 mode. No binary object file is produced and no execution occurs.

COMPILATION AND EXECUTION

Figure 12-4 illustrates a deck structure for compiling and executing a program that reads data from cards.

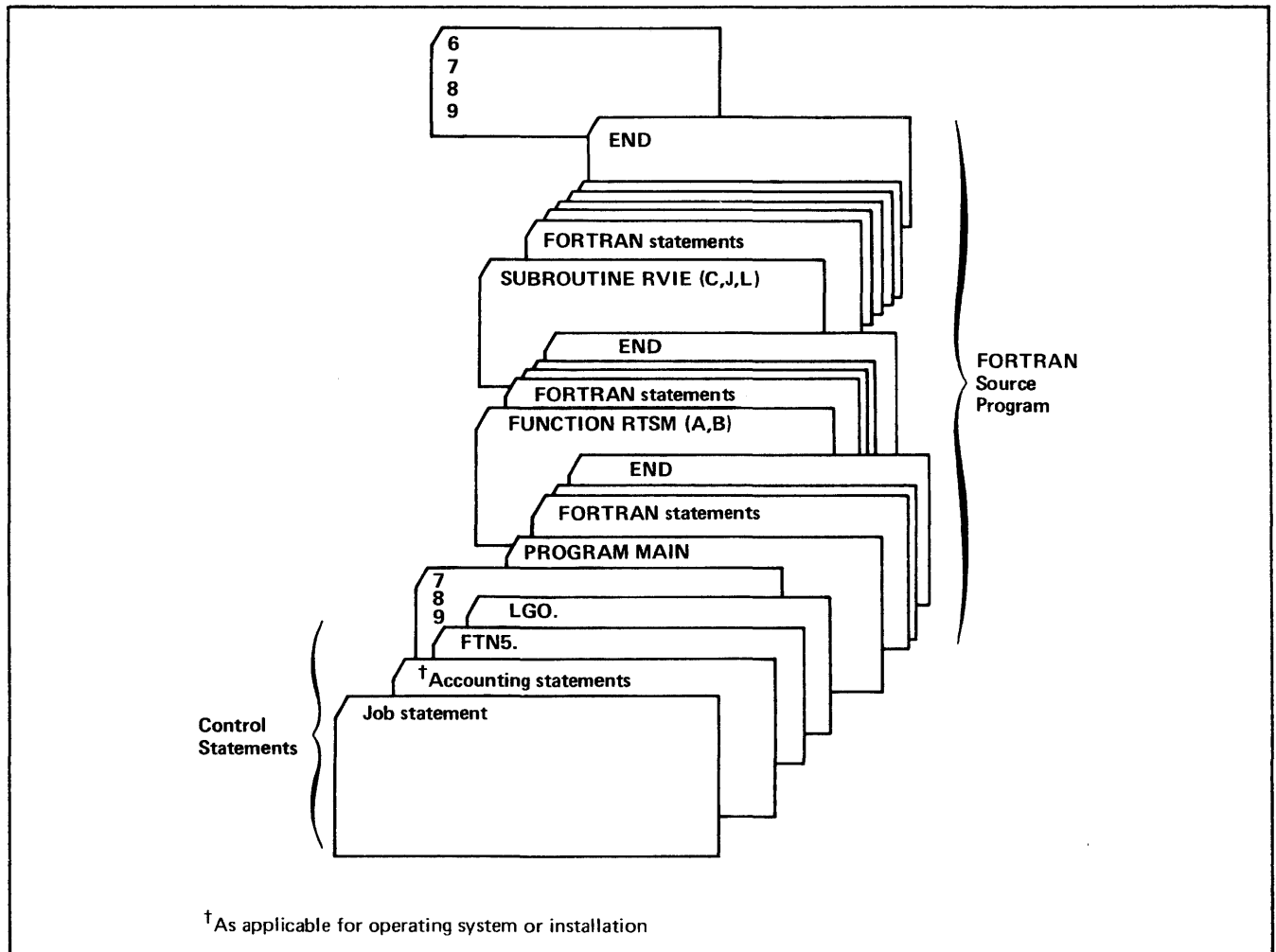


Figure 12-1. FORTRAN Source Program With Control Statements

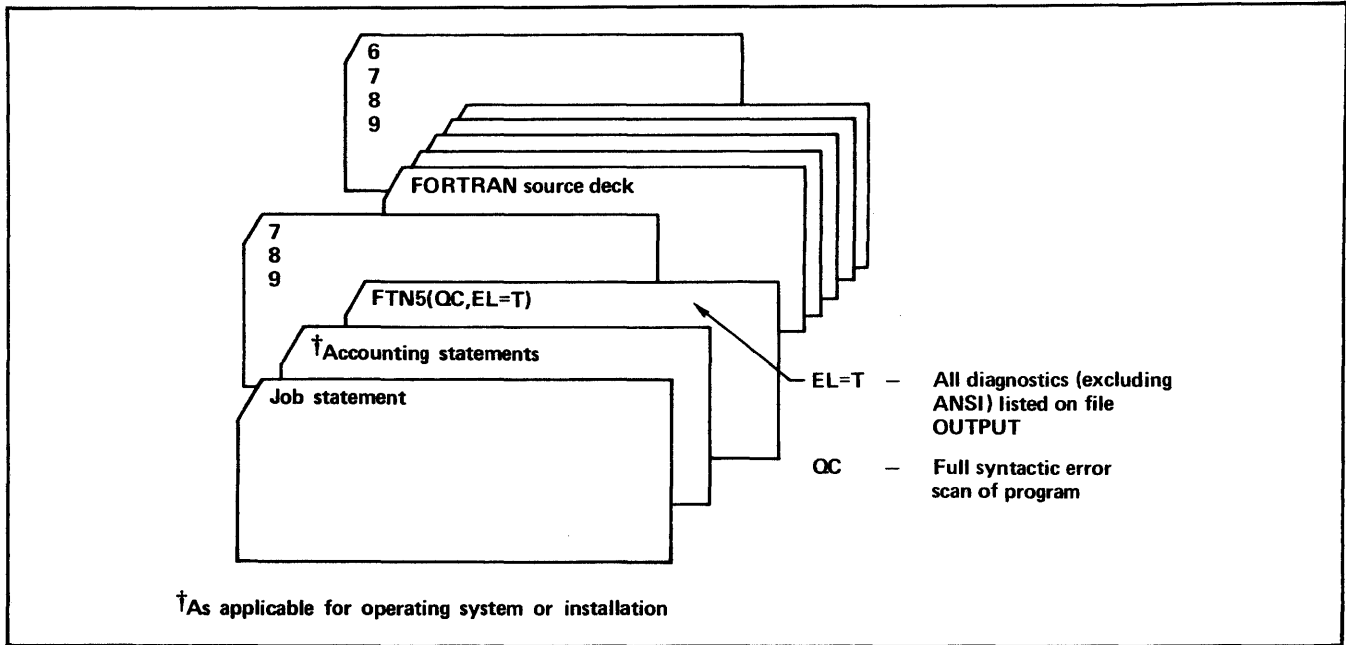


Figure 12-2. Compilation Only

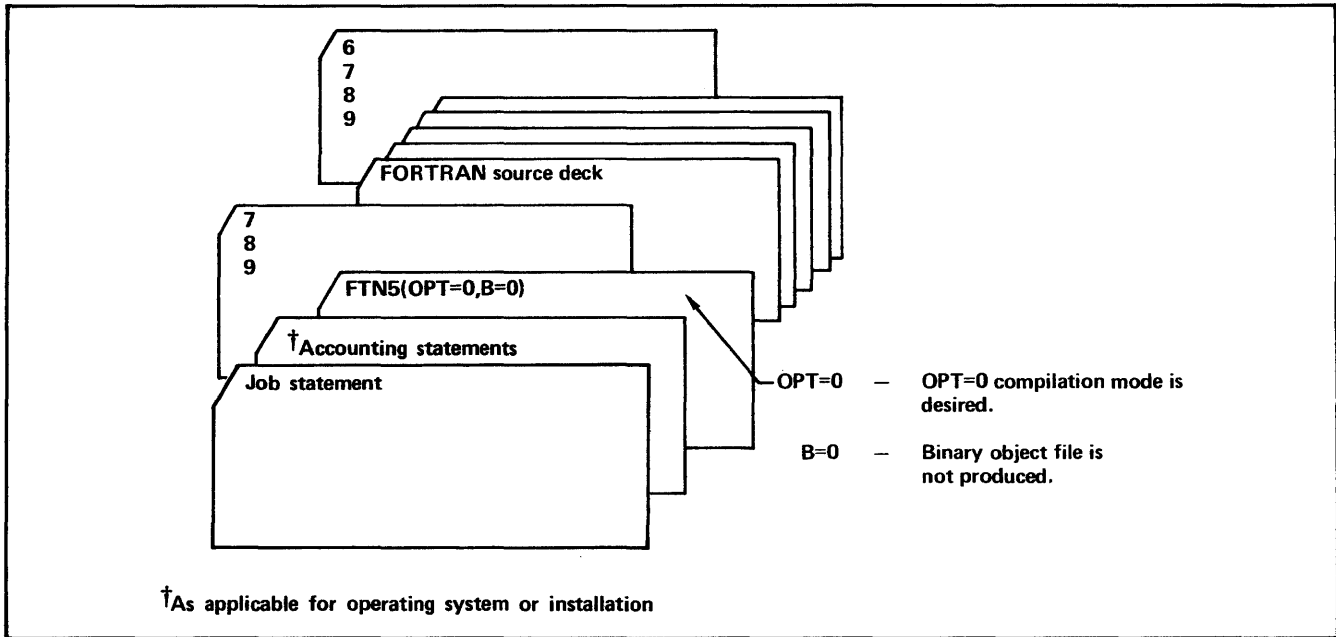


Figure 12-3. OPT=0 Compilation

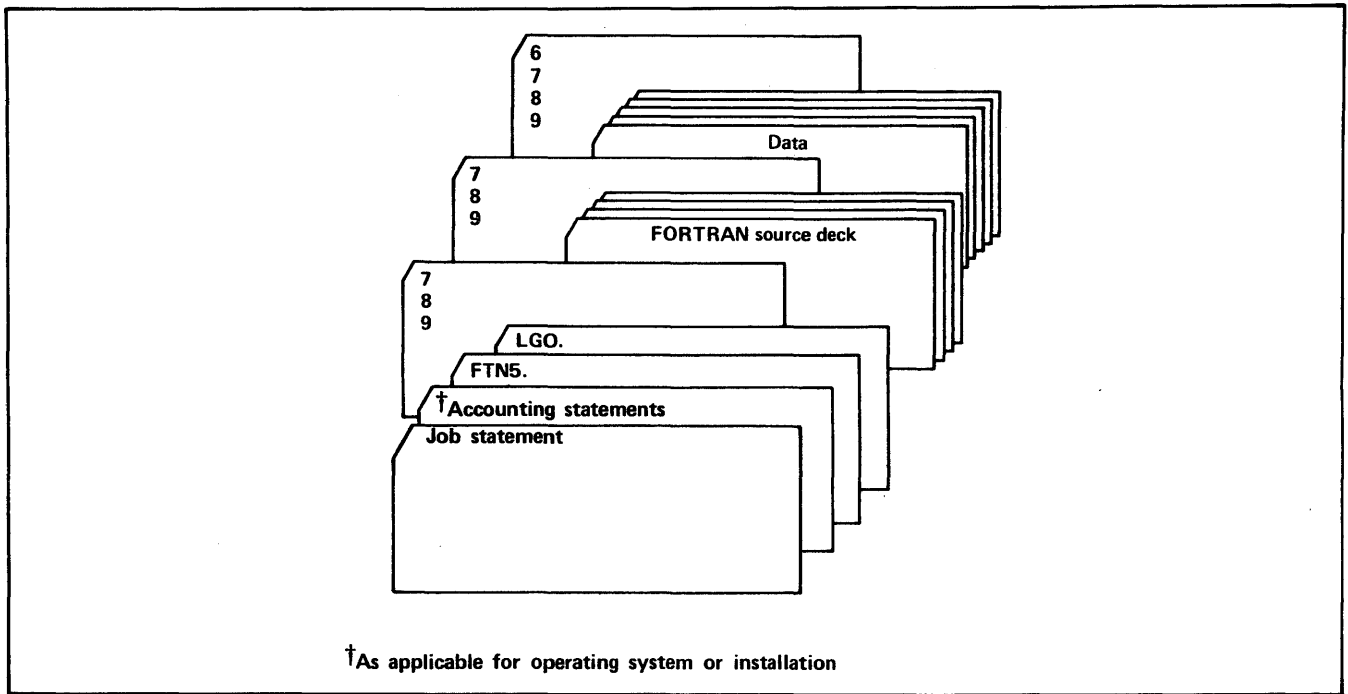


Figure 12-4. Compilation and Execution

**FORTRAN COMPILATION WITH COMPASS
ASSEMBLY AND EXECUTION**

Figure 12-5 illustrates a deck structure containing a FORTRAN and a COMPASS program unit. The FORTRAN and COMPASS source decks can be in any order.

COMPASS source decks must begin with a line containing the word:

IDENT

in columns 11 through 16. Columns 1 through 10 of the ident line must be blank.

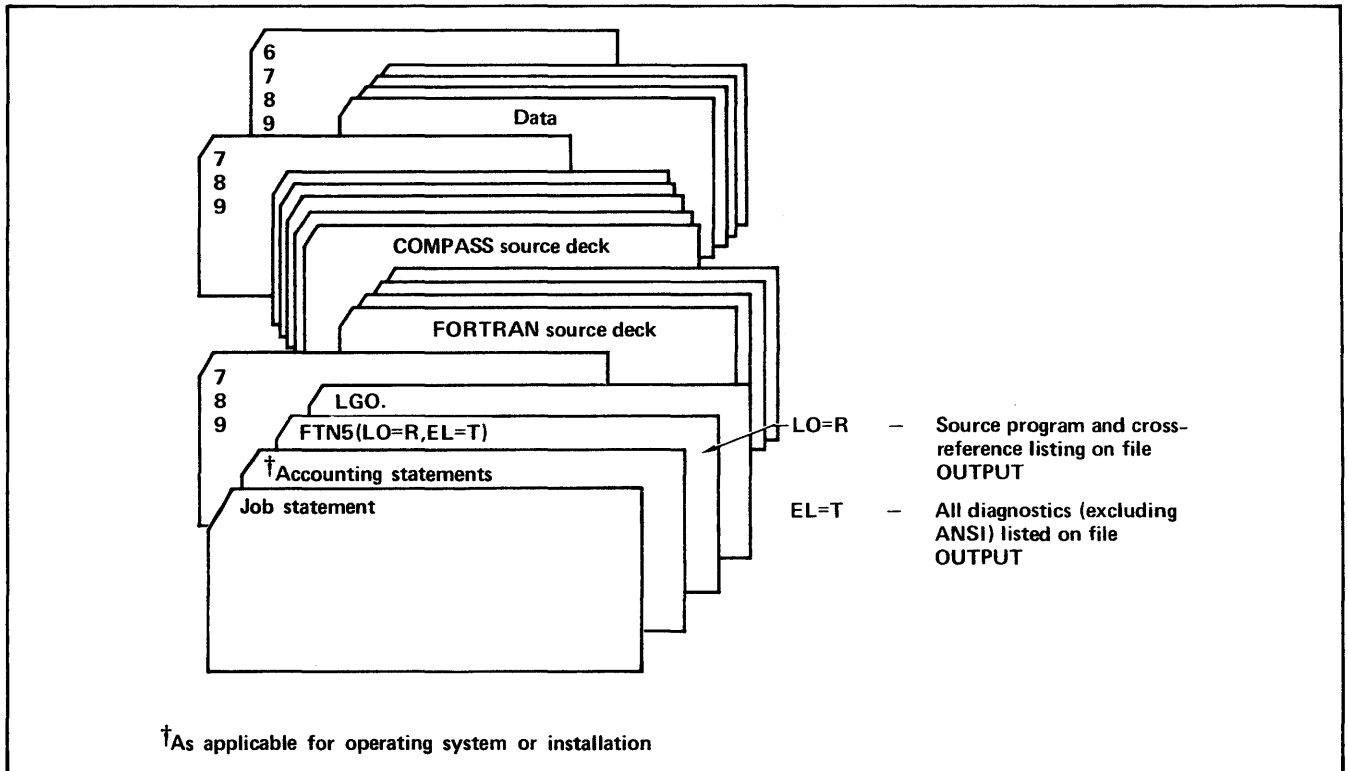


Figure 12-5. Compilation With COMPASS Assembly and Execution

COMPILE AND EXECUTION WITH FORTRAN SUBROUTINE AND COMPASS SUBPROGRAM

Figure 12-6 illustrates a deck structure containing a FORTRAN subroutine, and a COMPASS subprogram, showing the COMPASS IDENT and ENTRY statements. In this example, the LGO statement specifies the output file (as described in section 11).

COMPILE WITH BINARY CARD OUTPUT

Figure 12-7 illustrates a deck structure to compile and produce a binary object deck.

LOADING AND EXECUTION OF BINARY PROGRAM

Figure 12-8 illustrates a deck structure to load and execute a binary object program. The MAP(OFF) statement suppresses the load map.

COMPILE AND EXECUTION WITH RELOCATABLE BINARY DECK

Figure 12-9 illustrates a deck structure to compile a FORTRAN program and load and execute a binary program along with the FORTRAN program.

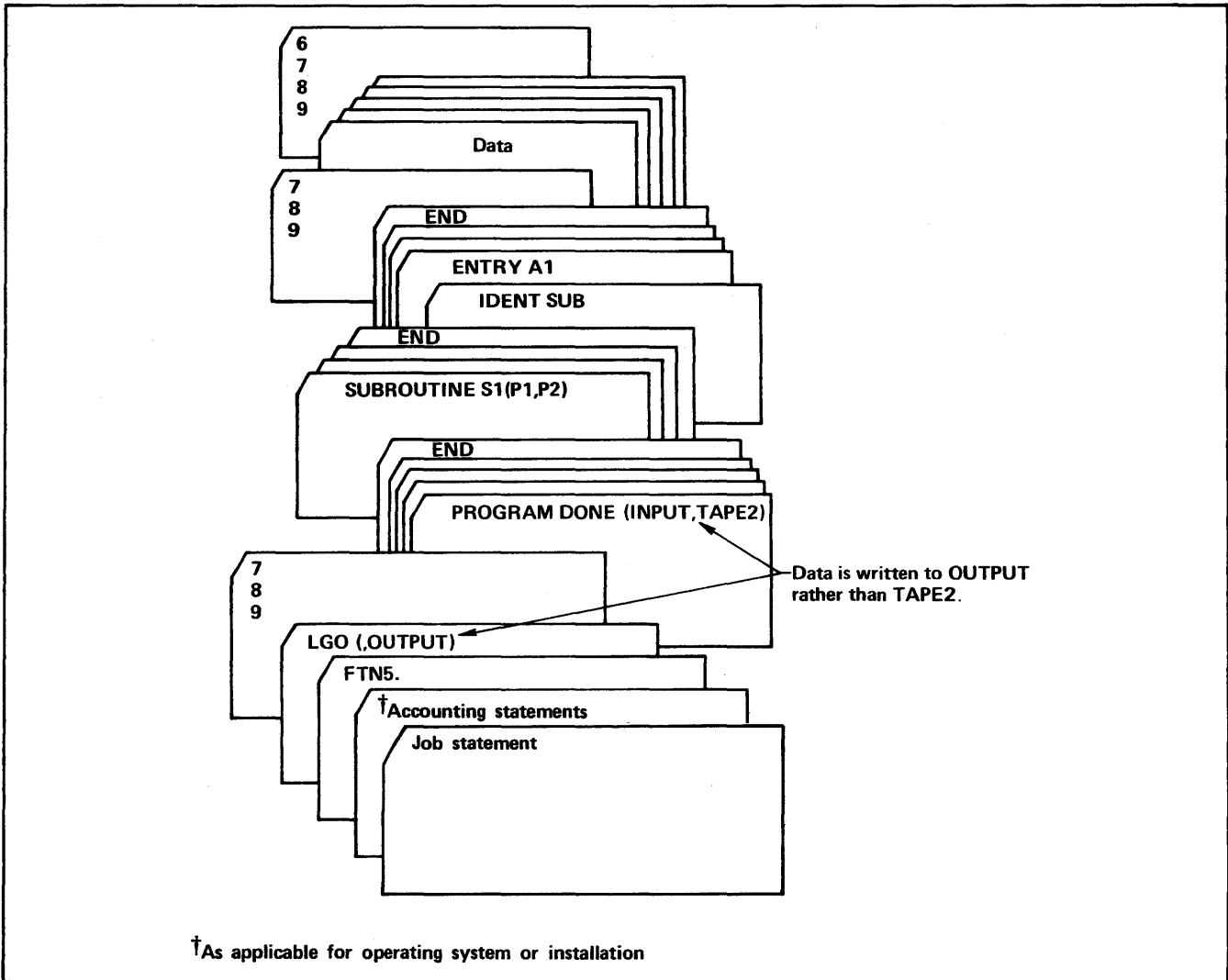


Figure 12-6. Compilation and Execution With FORTRAN Subroutines and COMPASS Subprogram

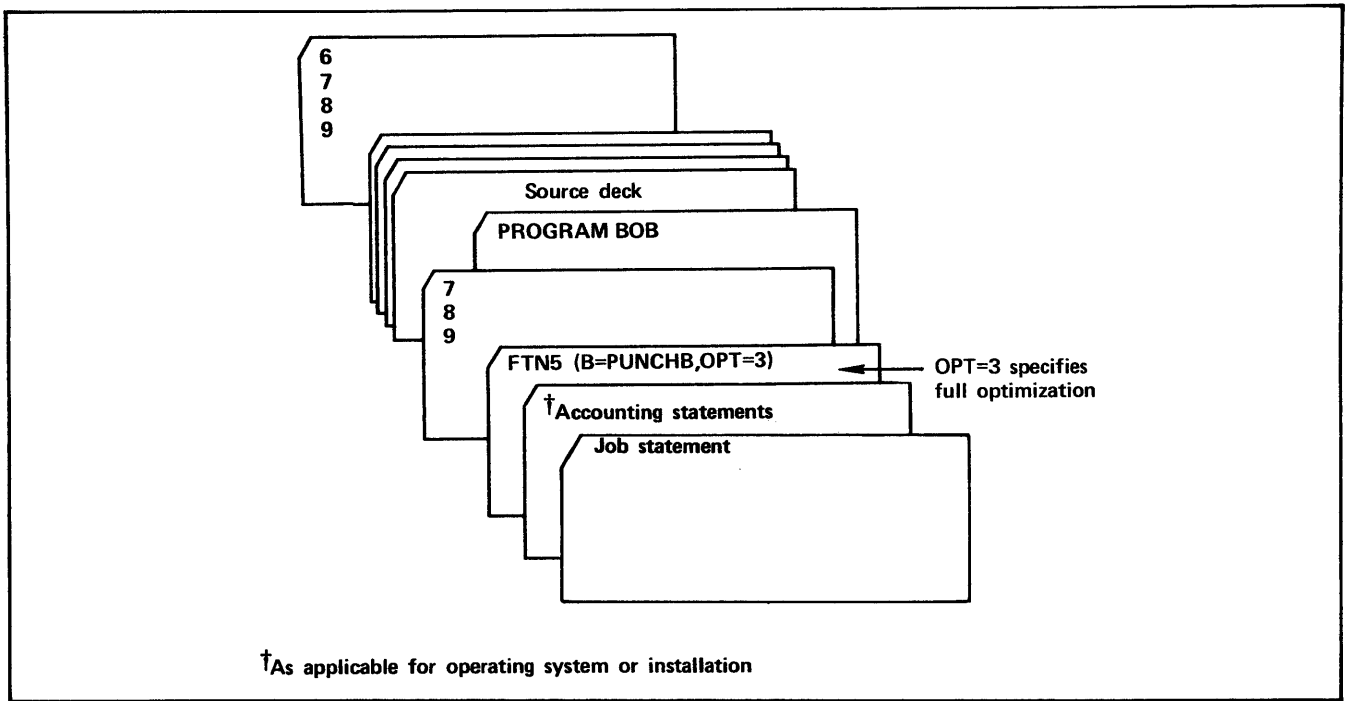


Figure 12-7. Compilation With Binary Card Output

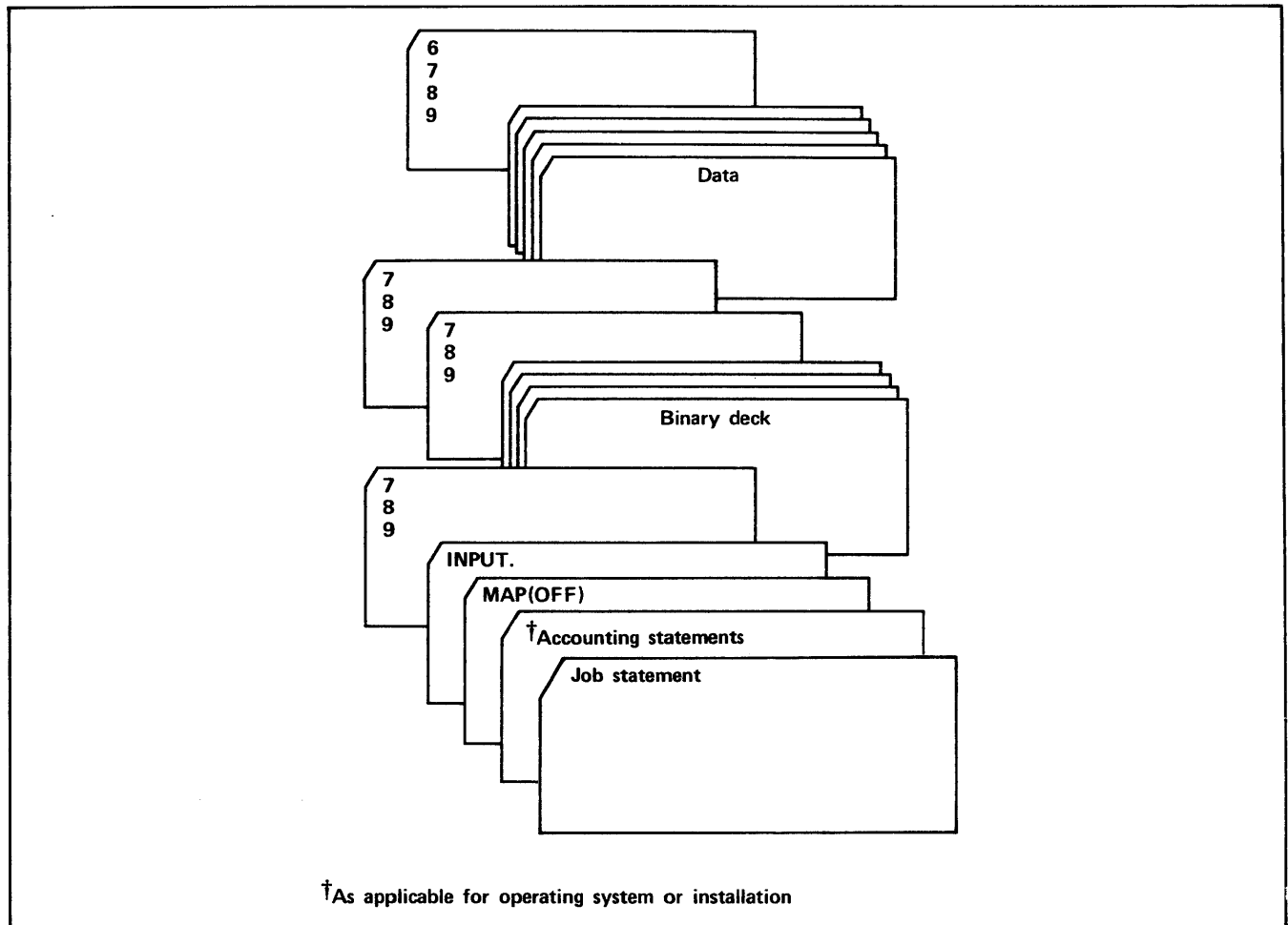


Figure 12-8. Loading and Execution of Binary Program

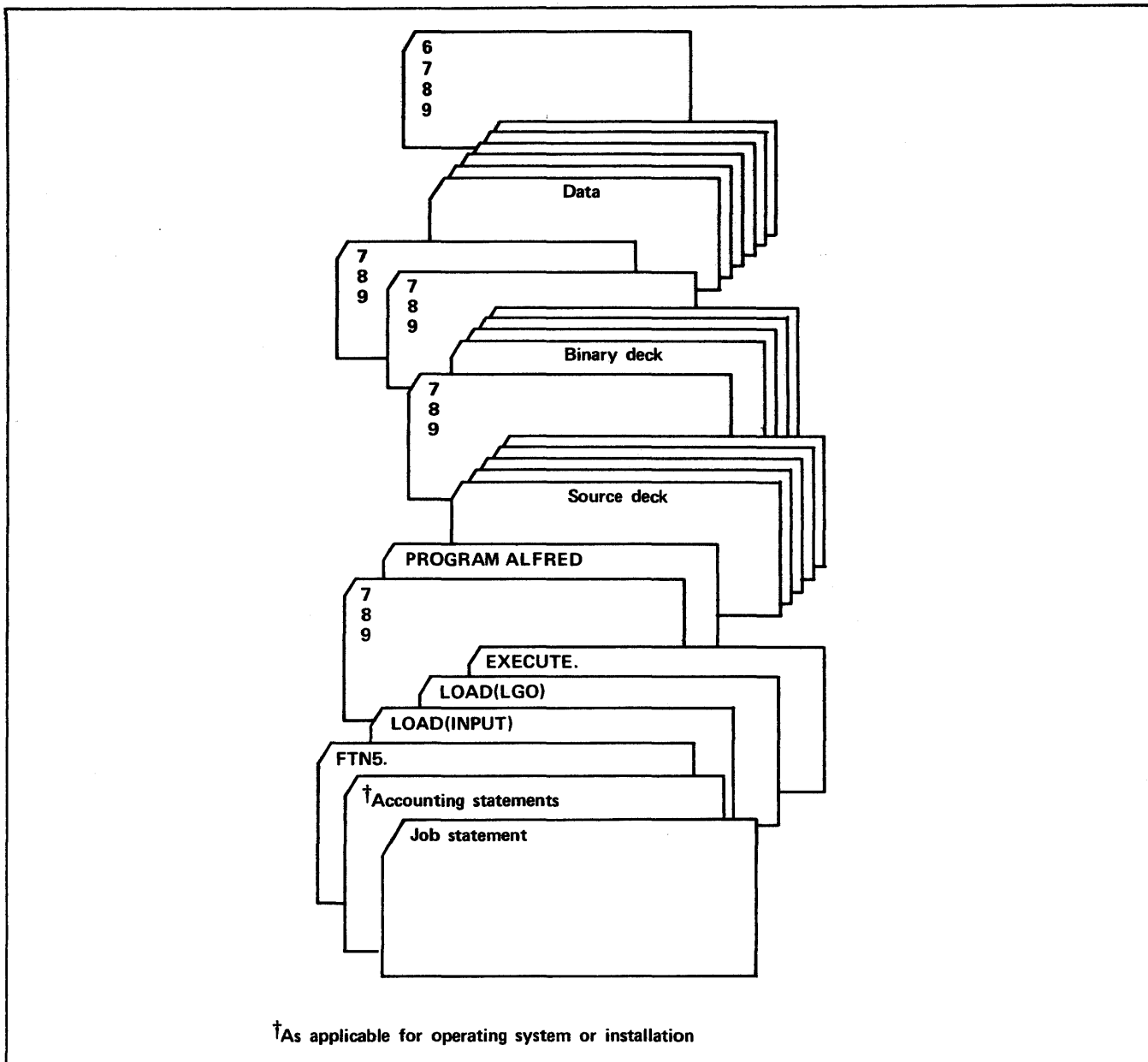


Figure 12-9. Compilation and Execution With Relocatable Binary Deck

COMPILATION AND TWO EXECUTIONS WITH DIFFERENT DATA DECKS

Figure 12-10 illustrates a deck structure to compile a program and to execute the program twice with two different data decks. Output from the two executions is sent to separate output files.

PREPARATION OF OVERLAYS

Figure 12-11 illustrates a deck structure to compile, load and execute a program containing overlays.

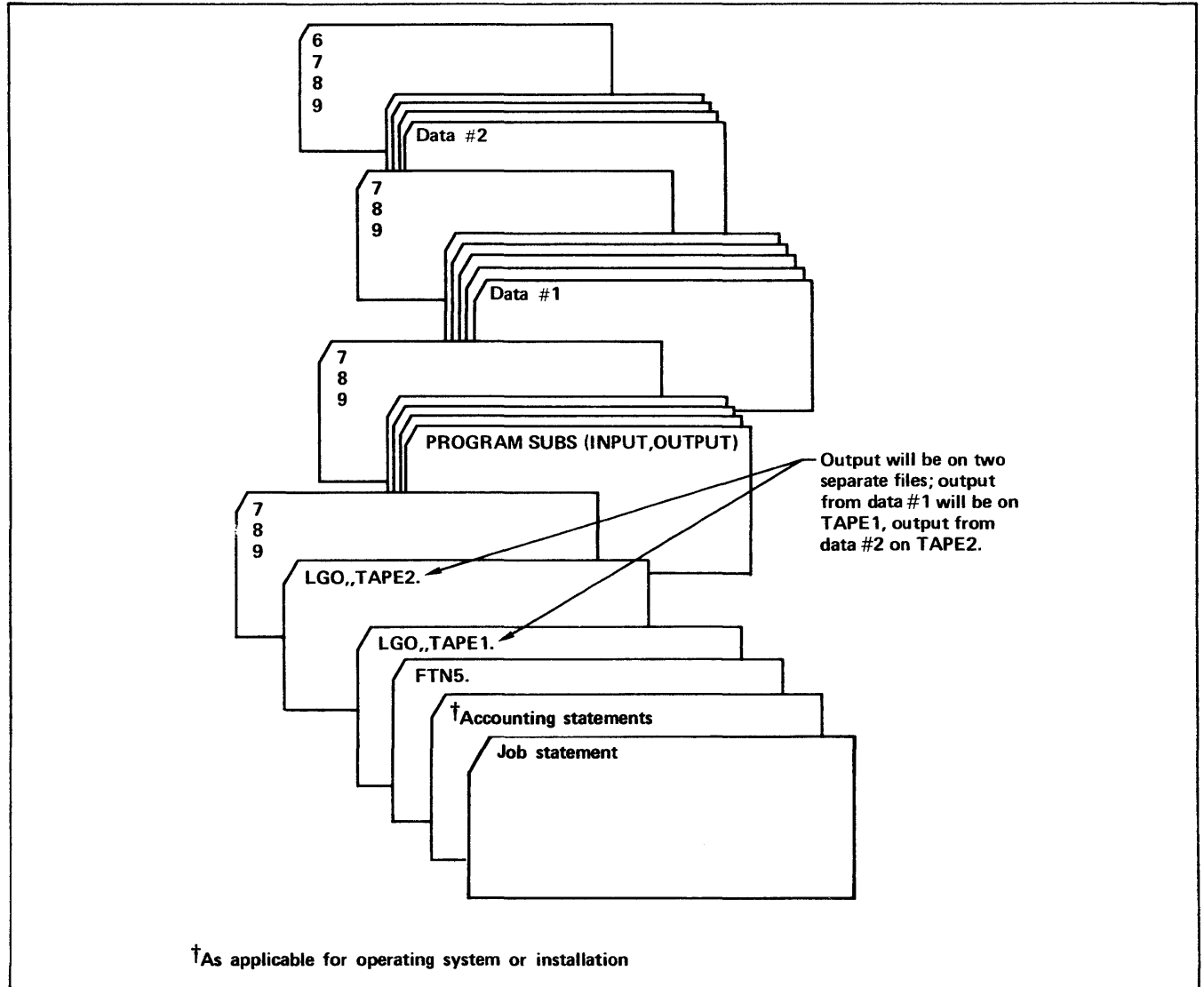
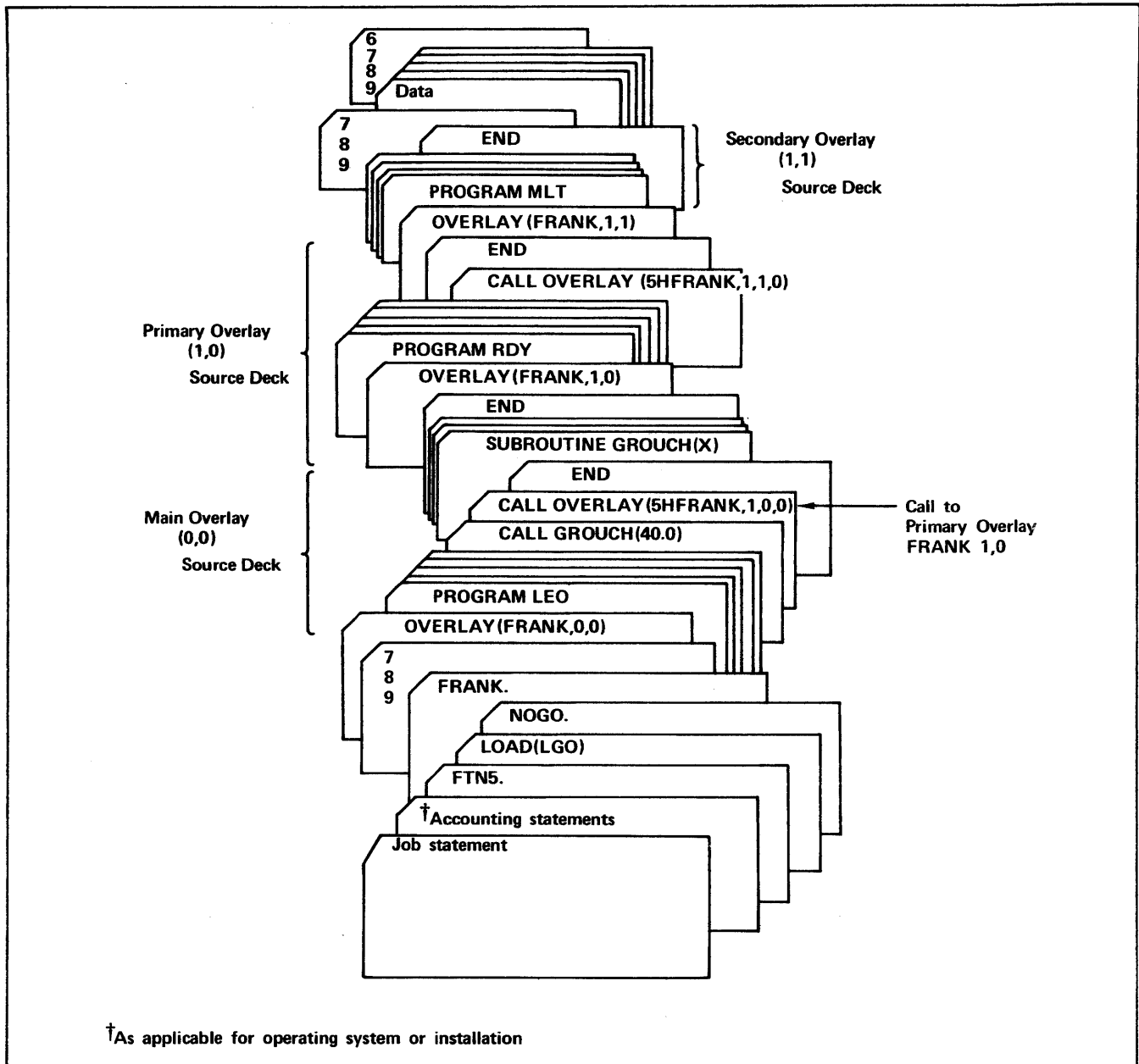


Figure 12-10. Compilation and Execution With Different Data Decks



†As applicable for operating system or installation

Figure 12-11. Preparation of Overlays

COMPILATION AND TWO EXECUTIONS WITH OVERLAYS

Figure 12-12 illustrates a deck structure to compile an overlay and to execute the overlay two times.

SAMPLE PROGRAMS

This subsection shows sample FORTRAN programs which illustrate various features of the FORTRAN language.

PROGRAM OUT

Program OUT, shown in figure 12-13, illustrates the following FORTRAN features:

Control statements for batch execution

WRITE and PRINT statements

Carriage control

PROGRAM statement

The control statement:

BIRD, T10.

is the job statement. A job statement must precede every job. BIRD is the job name. T10 specifies a maximum of 10 seconds of central processor time (can be either octal or decimal, depending on installation option).

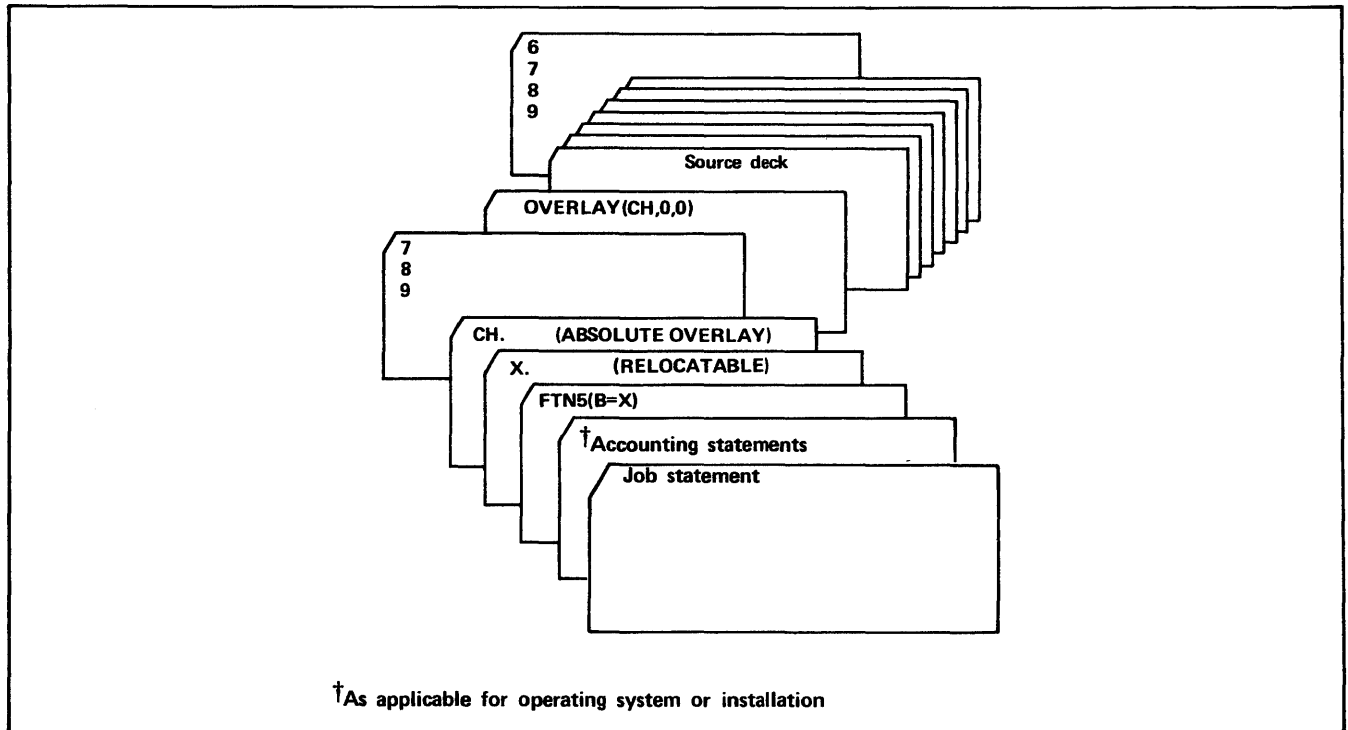


Figure 12-12. Compilation and Two Executions With Overlays

```

BIRD,T10.
FTN5.
LGO.
7/8/9 IN COLUMN 1
PROGRAM OUT
OPEN (6, FILE='OUTPUT')
PRINT 100
100 FORMAT ("1 THIS WILL PRINT AT THE TOP OF A PAGE")
INK= 2000 + 4000
WRITE (6,'(1X,I4," = INK OUTPUT BY WRITE STATEMENT)') INK
PRINT '(1H ,14, 30H = OUTPUT FROM PRINT STATEMENT)', INK
STOP
END
6/7/8/9 IN COLUMN 1

```

Figure 12-13. Program OUT With Control Statements

The statement:

FTN5.

specifies the FORTRAN compiler. The default parameters (described in section 10) are used. Since no alternative files are specified on the FTN5 control statement, the FORTRAN compiler reads from file INPUT and outputs to files OUTPUT and LGO. Listings, diagnostics, and maps are written to OUTPUT and the relocatable object code is written to LGO.

The statement:

LGO.

causes the binary object code to be loaded and executed.

The statement:

7/8/9

separates control statements from the remainder of the job deck (file INPUT). This statement contains a 7, 8, and 9 multipunched in column 1; it follows the control statements in every batch job.

The OPEN statement (line 2) associates unit 6 with file OUTPUT.

The WRITE statement (line 6) outputs the variable INK to file OUTPUT. The format specification is

included in the WRITE statement. If the following PRINT statement had been used instead of WRITE:

```
PRINT '(15, "= INK OUTPUT BY PRINT",  
*"STATEMENT")', INK
```

the OPEN statement would not be needed. The specification uses quotes to delimit the literal and the carriage control character 1 to cause the line to be printed at the top of a page.

Lines 6 and 7 print the variable INK. In both output statements, a blank carriage control character is specified to cause single spacing. Line 6 uses the specification 1X to produce a blank in column 1; line 7 uses the specification 1H for the same effect.

The 6/7/8/9 card contains the characters 6, 7, 8, and 9 multipunched in column 1. It is the last card in every job deck (INPUT file), indicating to the system the end of the job.

Output from program OUT is shown in figure 12-14.

```
THIS WILL PRINT AT THE TOP OF A PAGE  
6000 = INK OUTPUT BY WRITE STATEMENT  
6000 = OUTPUT FROM PRINT STATEMENT
```

Figure 12-14. Program OUT Output

PROGRAM B

Program B, shown in figure 12-15, generates a table of 64 characters. The internal bit configuration of any character can be determined by its position in the table. Each character occupies six bits.

Features illustrated in this example include:

Octal constants

Simple DO loop

PRINT statement

FORMAT with /,I,X and A editing

Character constant as a format specifier

The PRINT statement (line 2) has no output list; it prints out the heading at the top of the page using the information provided by the format specification. The 1 is the carriage control character, and the two slashes cause one line to be skipped before the next string is printed. The slash at the end of the format specification skips another line before the program output is printed.

The DO loop (lines 4 through 6) generates numbers 0 through 7 (note that a DO index can be zero). The PRINT statement (line 5) prints 0 through 7 (the value of J) on the left and the 8 characters in NCHAR on the right. The first iteration of the DO loop prints NCHAR as it appears on line 3. The octal value 01 is a display code A, 02 is a B, 03 is a C, etc. Line 6 adds the octal constant 10101010101010100000 to NCHAR; when this is printed on the second iteration of the DO loop, the octal value 10 is printed as a display code H, 11 as I, 12 as J, etc. Compare these values with the character set listed in appendix A.

Output from program B is shown in figure 12-16.

PROGRAM STATES

Program STATES, shown in figure 12-17, reads employee names and home states, ignoring all but the first two letters of the state name. If the state name starts with the letters CA, the name is printed. This program illustrates character handling.

The first PRINT statement (line 3) directs the printer to start a new page, print the heading NAME, and skip 3 lines.

The READ statement (line 5) reads the last name into LNAME, first name into FNAME, home state into STATE, and tests for end-of-file.

```
PROGRAM B
PRINT '( "TABLE OF INTERNAL VALUES", //, " 01234567", // )'
NCHAR= 0"00 01 02 03 04 05 06 07 00 00"
DO 3 J = 0,7
PRINT '(13, 1X, A8)', J, NCHAR
3 NCHAR= NCHAR + 0"10 10 10 10 10 10 10 00 00"
STOP
END
```

Figure 12-15. Program B

TABLE OF INTERNAL VALUES

01234567

```
0 :ABCDEFGH
1 HIJKLMNO
2 PQRSTUUV
3 XYZ01234
4 56789+ -*
5 /()$= ,.
6 #[]% "- !&
7 '?<>@\^;
```

Figure 12-16. Program B Output

```
PROGRAM STATES
CHARACTER*10 FNAME, LNAME, STATE
PRINT 1
1 FORMAT (1H1, 5X, 4HNAME, ///)
3 READ (*, '(3A)', END=99) LNAME,
X FNAME, STATE
C
C IF FIRST TWO CHARACTERS OF STATE ARE CA
C PRINT LAST NAME AND FIRST NAME
C
IF (STATE(1:2) .EQ. 'CA') THEN
PRINT '(5X, 2A)', LNAME, FNAME
ENDIF
GO TO 3
99 STOP
END
```

Figure 12-17. Program STATES

The relational operator .EQ. tests to determine if the first two letters read into variable STATE match the two letters of the constant 'CA'. If a match occurs, FNAME and LNAME are printed.

Sample input and output for program STATES are shown in figure 12-18.

PROGRAM EQUIV

Program EQUIV, shown in figure 12-19, places values in variables that have been equivalenced and prints these values using the NAMELIST statement. The following features are illustrated:

EQUIVALENCE statement

NAMELIST statement

```

Input:
BROWN,    PHILLIP M.CA
BICARDI,  R. J.      KENTUCKY
CROWN,    SYLVIA     CAL
HIGENBERF,ZELDA    MAINE
MUNCH,    GARY G.    CALIF
SMITH     SIMON      CA
DEAN,     ROGER     GEORGIA
RIPPLE    SALLY     NEW YORK
JONES     STAN       OREGON
HEATH     BILL      NEW YORK

Output:

NAME

BROWN,    PHILLIP M.
CROWN,    SYLVIA
MUNCH,    GARY G.
SMITH     SIMON

```

Figure 12-18. Sample Input and Output for Program STATES

```

PROGRAM EQUIV
EQUIVALENCE (X,Y), (Z,I)
NAMELIST /OUT/ X, Y, Z, I
OPEN (6, FILE='OUTPUT')
X= 1.
Y= 2.
Z= 3.
I= 4
WRITE (6,OUT)
STOP
END

```

Figure 12-19. Program EQUIV

Line 2 equivalences two real variables X and Y; the two variables share the same location in storage, which can be referred to as either X or Y. Any change made to one variable changes the value of the others in an equivalence group as illustrated by the output of the WRITE statement, in which both X and Y have the value 2. . The storage location shared by X and Y contains first 1. (X=1.), then 2. (Y=2.).

The real variable Z and the integer variable I are equivalenced, and the same location can be referred to as either real or integer. Since integer and real internal formats differ, however, the output values will not be the same.

For example, the storage location shared by Z and I contained first 3.0 (real value), then 4 (integer value). When I is output, no problem arises; an integer value is referred to by an integer variable name. However, when this same integer value is referred to by a real variable name, the value 0.0 is output, because the internal formats of real and integer values differ. The integer and real internal formats are shown in figure 12-20.

Although a value can be referred to by names of different types, the internal bit configuration does not change. An integer value output as a real variable has a zero exponent and its value is zero.

When variables of different types are equivalenced, the value in the storage location must agree with the type of the variable name, or unexpected results might be obtained.

This NAMELIST WRITE statement (line 10) outputs both the name and the value of each member of the NAMELIST group OUT defined in the statement NAMELIST/OUT/X,Y,Z,I. The NAMELIST group is preceded by the group name, OUT, and terminated by the characters \$END. Output is shown in figure 12-21.

PROGRAM COME

Program COME, shown in figure 12-22, places variables and an array in common and declares another variable and array equivalent to the first element in common. It places the numbers -1 through -12 in each element of the array IA and outputs values in common using the NAMELIST statement. Features illustrated include:

- COMMON and EQUIVALENCE statements
- NAMELIST statement
- Negative subscript
- Negative DO loop parameters

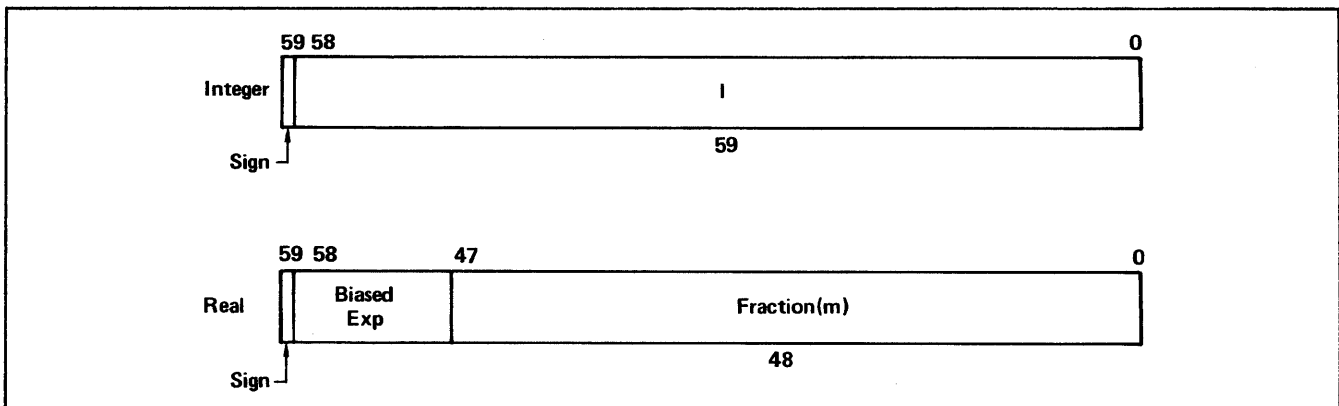


Figure 12-20. INTEGER and REAL Internal Formats

Variables are stored in common in the order of appearance in the COMMON statement: A, B, C, D, F, G, H. All variables with the exception of G are declared integer. G is implicitly typed real.

The EQUIVALENCE statement assigns the first element of the arrays IA and E to the same storage location as the variable A. The subscript of IA has a lower bound of -12. Since A is in common, E and IA will be in common. Variables and array elements are assigned storage as shown in figure 12-23.

The DO loop places values -1 through -12 in IA using a negative DO index. The first element of IA (indexed by -12) shares the same location as the first element of E. This location is also shared by A. IA(-11) is equivalent to E(2,1) and B; IA(-10) is equivalent to E(3,1) and C, and so forth.

Any change made to one member of an equivalence group changes the value of all members of the group. When -12 is stored in IA(-12), both E(1,1) and A have the value -12. When -11 is stored in IA(-11), B and E(2,1) have the value -11. Although B and E(2,1) are not explicitly equivalenced to IA(-11), equivalence is implied by their position in common.

The implied equivalence between the array elements and variables is illustrated by the output shown in figure 12-24.

```

$OUT
X      = .2E+01,
Y      = .2E+01,
Z      = 0.0,
I      = 4,
$END

```

Figure 12-21. Program EQUIV Output

```

PROGRAM COME
COMMON A,B,C,D, F,G,H
INTEGER A,B,C,D,E(3,4),F, H,IA(-12:-1)
EQUIVALENCE (A, E, IA)
NAMelist /V/ A,B,C,D,E,F,G,H,IA
C
OPEN (6, FILE='OUTPUT')
DO 2 J=-1, -12, -1
2  IA(J)= J
WRITE (6,V)
C
STOP
END

```

Figure 12-22. Program COME

Relative Address	0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11
I												
E(1,1)	E(2,1)	E(3,1)	E(1,2)	E(2,2)	E(3,2)	E(1,3)	E(2,3)	E(3,3)	E(1,4)	E(2,4)	E(3,4)	
A	B	C	D	F	G	H						
IA(-12)	IA(-11)	IA(-10)	IA(-9)	IA(-8)	IA(-7)	IA(-6)	IA(-5)	IA(-4)	IA(-3)	IA(-2)	IA(-1)	

Figure 12-23. Storage Layout for Variables in Program COME

```

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

```

Figure 12-24. Program COME Output

The NAMELIST statement is used for output. A NAMELIST group, V, containing the variables and arrays A, B, C, D, E, F, G, H, IA is defined. The NAMELIST WRITE statement, WRITE(6,V), outputs all the members of the group in the order of appearance in the NAMELIST statement. Array E is output on one line in the order in which it is stored in memory. There is no indication of the number of rows and columns (3,4).

G is equivalent to E(3,2) and yet the output for E(3,2) is 6 and G is 0.0. G is type real and E is type integer. When two names of different types are used for the same element, their values will differ because the internal bit configuration for type real and type integer differ. (Refer to Program EQUIV.)

Output from program COME is shown in figure 12-24.

PROGRAM LIBS

Program LIBS, shown in figure 12-25, illustrates the following features:

Use of FORTRAN library subroutines and intrinsic functions

EXTERNAL used to pass a library subroutine name as a parameter to another library routine

INTRINSIC used to pass an intrinsic function name as a parameter to another library routine

Division by zero

LEGVAR function used to test for overflow or divide error conditions

The following functions and subroutines are used in LIBS:

DATE

TIME

SECOND

RANGET

SQRT

SIN

DATE is a library function which returns the date entered by the operator from the console.

SQRT is an intrinsic function that calculates the square root of its argument. SIN is an intrinsic function that calculates the sine of its argument. These functions are declared INTRINSIC so that they can be passed as arguments to a subprogram.

The PRINT statement in line 10 prints the date and time. The arguments TODAY and CLOCK are declared character with length 10 because the DATE and TIME functions each return 10 characters. The leading and trailing blanks appear with the 10 characters returned by the subroutine DATE, because the operating system formats the date in this manner. (The date format is system and installation dependent.) The value returned by TIME is changed by the system once a second, and the position of the digits remains fixed; a leading blank always appears.

```

C      PROGRAM LIBS
C      CHARACTER*10 TODAY, CLOCK, DATE, TIME
C      EXTERNAL DATE
C      INTRINSIC SQRT, SIN
C
C      TODAY= DATE()
C      CLOCK= TIME()
C
C      PRINT 2, TODAY, CLOCK
2      FORMAT ('1TODAY= ', A, ' CLOCK= ', A)
C
C      TYME= SECOND()
C      CALL RANGET (SEED)
C      Y= FUNC(SQRT)
C      Y1= FUNC(SIN)
C
C      PRINT 3, TYME, Y, Y1, SEED, SEED
3      FORMAT (' THE ELAPSED CPU TIME IS',G14.5,' SECONDS.'//,' SQRT(2.4
* )/PI = ',G14.5,/' SIN(2.4)/PI = ',G14.5,/' THE INITIAL VALUE OF T
*HE RANF SEED IS',022,', OR',/G30.15,' IN G30.15 FORMAT.')
C
C      Y= 0.0
C      WOW= 7.2/Y
C      IF (LEGVAR(WOW) .NE. 0) PRINT 4, WOW
4      FORMAT (1H0,50(2H*-)/' DIVIDE ERROR, WOW PRINTS AS:',G10.2)
C      STOP
C      END
C      FUNCTION FUNC(F)
C      FUNC= F(2.4)/3.14159
C      RETURN
C      END

```

Figure 12-25. Program LIBS

Read

A formatted READ statement for an external file places the image of each record read into an input buffer. Compiler routines convert the character string in the record into floating-point, integer, or logical values, as specified by the FORMAT statement, and store these values in the locations associated with the variables named in the list.

With internal files, the specified file (character variable, substring, or array) is used as the input buffer. The record length is equal to the length, in characters, of the variable if the file is a character variable, of a single array element if the file is an array, or of the substring.

With external files, when the format specification indicates a new record is to be processed (by a slash or the final right parenthesis of the FORMAT statement), a new record is read into the input buffer.

With internal files, when the format specification indicates a new record is to be processed (by a slash or final right parenthesis), the next element of the array is used as the input buffer.

Write

A formatted WRITE statement for external files causes the output buffer to be cleared. Data in the WRITE statement list is converted into a character string according to the format specified in the format specification, and placed in the output buffer. When the format specification indicates the end of a record with either a slash or the final right parenthesis, the character string is passed from the output buffer to the output system; the output buffer area is reset, and the next string of characters is placed in the buffer.

The WRITE statement for internal files is processed by compiler routines in the same way as for external files, but with the internal file specified within the WRITE statement used as the output buffer. The number of words per record in the array is determined by the length of an element.

In the sample program, the format of data on input is specified in column 1 of each input card. If column 1 contains a one or zero or blank, each of the remaining columns contains a data item. If column 1 is a two, each pair of the remaining columns is a data item. If column 1 contains a number equal to or greater than 3, each triplet of the remaining columns is a data item. Based on the information in column 1, the correct format specification is selected. The program then totals and prints out the items in each input record.

CARD is a character variable 79 characters long, which is to receive the characters in columns 2 through 80 of the input record. IN is dimensioned 79 to receive the converted input items. FM is a character array which contains three elements, each six characters long. The DATA statement (line 4) loads a format specification into each element of FM.

The READ statement in line 8 reads the first column of an input record into KEY under I format and the remaining 79 characters into CARD under A format. When an end-of-file is encountered, control transfers to statement 100, a STOP statement.

Line 9 ensures that the value of KEY is between 1 and 3; this value is stored in N.

Line 10 calculates the number of values to be transferred to IN.

The READ statement in line 12 transmits the characters in CARD to IN, converting them to integers according to the format specification stored in FM; N selects the array element containing the correct format specification.

Lines 14 through 20 sum the values in IN, print the input and output values, and branch back to process the next input record.

Sample input and output records for program ADD are shown in figure 12-28.

PROGRAM PASCAL

Program PASCAL, shown in figure 12-29, produces a table of binary coefficients (Pascal's triangle). The following features are illustrated:

Nested DO loops

Implied DO loop

The DO loop in lines 6 and 7 initializes the integer array LROW to 1. The PRINT statement in line 8 prints a heading and the first two rows of the triangle.

The nested DO loops (lines 11 through 15) calculate the remaining elements of the triangle. These statements illustrate the technique of going backward through an array by using a negative incrementation parameter.

Each pass through the inner DO loop generates one row of the triangle. The row elements are written in line 14 using an implied DO loop.

Output from program PASCAL is shown in figure 12-30.

PROGRAM PIE

Program PIE, shown in figure 12-31, calculates an approximation of the value of π . This program illustrates the use of the intrinsic function RANF.

The random number generator, RANF, is called twice during each iteration of the DO loop, and the values obtained are stored in the variables X and Y.

The DATA statement (line 2) initializes the variable circle with the value 0.0.

Each time RANF is called, a random number, uniformly distributed over the range 0 through 1, is returned. A random number is stored in X and in Y.

Input:

```
21322554766988775533210332245666877965541233322112365478965412365547896541236028
3021445669987745663221445566665233655222144455663325566699885666554778854887029
55566663223666552332214455666998877655222144455611223303324456669988774558896030
10234566688899887789965554444556665533222111233023333669985555222114444777885031
```

Output:

```
1900 IS THE TOTAL OF THE 39 NUMBERS ON THE CARD
21322554766988775533210332245666877965541233322112365478965412365547896541236028
THE NUMBERS ARE
 13 22 55 47 66 98 87 75 53 32 10 33 22 45 66 68 77 96 55 41
 23 33 22 11 23 65 47 89 65 41 23 65 54 78 96 54 12 36 2
```

```
14380 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
3021445669987745663221445566665233655222144455663325566699885666554778854887029
THE NUMBERS ARE
 21 445 669 987 745 663 221 445 566 665 523 365 522 214 445 566 332 556 669 988
566 655 477 885 488 702
```

```
13840 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
35566663223666552332214455666998877655222144455611223303324456669988774558896030
THE NUMBERS ARE
556 666 322 366 655 233 221 445 566 699 887 765 522 214 445 561 122 330 332 445
666 998 877 455 889 603
```

```
370 IS THE TOTAL OF THE 79 NUMBERS ON THE CARD
10234566688899887789965554444556665533222111233023333669985555222114444777885031
THE NUMBERS ARE
 0 2 3 4 5 6 6 6 8 8 8 9 9 8 8 7 7 8 9 9
 6 5 5 5 4 4 4 4 5 5 6 6 6 5 5 3 3 2 2 2
 1 1 1 2 3 3 0 2 3 3 3 3 6 6 9 9 8 5 5 5
 5 2 2 2 1 1 4 4 4 4 7 7 7 8 8 5 0 3 1
```

Figure 12-28, Program ADD Input and Output

```
PROGRAM PASCAL
C
C THIS PROGRAM PRODUCES A PASCAL TRIANGLE WITH 15 ROWS
C
INTEGER LROW(15)
DO 10 I=1,15
LROW(I)= 1
PRINT ('1 PASCAL TRIANGLE "//1X, 15,/1X, 215)', LROW(15),
* LROW(14), LROW(15)
C
DO 50 J = 14, 2, -1
DO 40 K=J,14
LROW(K)= LROW(K) + LROW(K+1)
PRINT '(1X, 15I5)', (LROW(M), M=J-1,15)
50 CONTINUE
C
STOP
END
```

Figure 12-29. Program PASCAL

PASCAL TRIANGLE

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1

```

Figure 12-30. Program PASCAL Output

```

PROGRAM PIE
DATA CIRCLE /0.0/
C
DO 1 I = 1,1000
X= RANF()
Y= RANF()
IF (X*X + Y*Y .LE. 1.0) CIRCLE= CIRCLE + 1.0
1 CONTINUE
C
PI= 4.0*CIRCLE/1000.0
PRINT*, ' PI = ', PI
C
STOP
END

```

Output:

PI = 3.148

Figure 12-31. Program PIE and Output

The IF statement and the arithmetic expression $4.0 * \text{CIRCLE} / 1000.0$. calculate an approximation of the value of π . The value of π is calculated using Monte Carlo techniques. The IF statement counts those points whose distance from the point (0., 0.) is less than or equal to one. The ratio of the number of points within the quarter circle to the total number of points approximates $1/4$ of π . The value PI is printed by the list directed output statement PRINT*, 'PI=', PI.

PROGRAM X

Program X, shown in figures 12-32 and 12-33, references a function EXTRAC which squares the number passed as an argument. This program illustrates the following features:

Referencing user-defined functions

Function type

Program X illustrates that a function type must agree with the type associated with the function name in the calling program.

In the example shown in figure 12-32, the first letter of the function name EXTRAC is E and the function is therefore implicitly typed real. EXTRAC is referenced, and the value 7 is passed to the function as an argument. However, the function subprogram is explicitly defined integer, INTEGER FUNCTION EXTRAC(K), and the conflicting types produce erroneous results.

The argument 7 is type integer which agrees with the type of the dummy argument K in the subprogram. The result 49 is correctly computed. However, when this value is returned to the calling program, the integer value 9 is returned to the real name EXTRAC; and an integer value in a real variable produces an erroneous result. (Refer to program EQUIV.)

This problem arises because the programmer and the compiler regard a program from different viewpoints. The programmer often considers a complete program to be one unit, whereas the compiler treats each program unit separately. To the programmer, the statement:

```
INTEGER FUNCTION EXTRAC(K)
```

```

PROGRAM X
C IF EXTRAC IS DECLARED TYPE INTEGER THE RESULT IS 49, OTHERWISE IT IS
C ZERO
C
      K= EXTRAC(7)
      PRINT ((''K = ', 15)', K
      STOP
      END
C

```

Function EXTRAC:

```

      INTEGER FUNCTION EXTRAC (K)
      EXTRAC= K*K
      RETURN
      END

```

Output:

K = 0

Figure 12-32. Program X, Function EXTRAC, Output: INTEGER Declaration Omitted From Main Program

```

PROGRAM X
C IF EXTRAC IS DECLARED TYPE INTEGER THE RESULT IS 49, OTHERWISE IT IS
C ZERO
C
      INTEGER EXTRAC
      K= EXTRAC(7)
      PRINT ((''K = ', 15)', K
      STOP
      END
C

```

Function EXTRAC:

```

      INTEGER FUNCTION EXTRAC (K)
      EXTRAC= K*K
      RETURN
      END

```

Output:

K = 49

Figure 12-33. Program X, Function EXTRAC, Output: INTEGER Declaration Included in Main Program

defines the function EXTRAC integer. The compiler, however, compiles integer function EXTRAC and the main program separately. In the subprogram, EXTRAC is declared integer; in the main program it is declared real. Information (in this instance the type of the function) which the main program needs regarding a subprogram, must be supplied in the main program.

There is no way for the compiler to determine if the type of a program unit agrees with the type of the name in the calling program; therefore, no diagnostic help can be given for errors of this kind.

In figure 12-33, EXTRAC is declared integer in the calling program, and the correct result is obtained.

PROGRAM ADIM

Program ADIM, shown in figure 12-34, illustrates the use of adjustable dimensions to allow a subroutine to operate on arrays of various sizes. The following features are included in this example:

Passing an array to a subroutine as a parameter

Specifying an array name, with no dimension information, in an argument list

Specifying an array with a negative lower subscript bound

```

        PROGRAM ADIM
        COMMON X(4,3), Z(-2:3)
        REAL Y(6)
C
        CALL IOTA (X,12)
        CALL IOTA (Y,6)
        CALL IOTA (Z,6)
C
        PRINT 100, X, Y, Z
100    FORMAT ('1ARRAY X = ',12F6.0/' ARRAY Y = ', 6F6.0,
* /' ARRAY Z = ',6F6.0)
C
        DO 8 I = -2,3
        8    Z(I)= I
        PRINT 110, Z
110    FORMAT (' ARRAY Z = ',6F6.0)
C
        STOP
        END
C
        SUBROUTINE IOTA (A,M)
C
C    IOTA STORES CONSECUTIVE INTEGERS IN EVERY ELEMENT OF THE ARRAY A
C    STARTING AT 1
C
        DIMENSION A(M)
        DO 1 I = 1,M
        1    A(I)= I
        RETURN
        END

```

Figure 12-34. Program ADIM and Subroutine IOTA

Two arrays, X and Z, are dimensioned and placed in common. Z is dimensioned (-2:3). This means that Z has six elements; the lower subscript bound is -2 and the upper subscript bound is 3. The elements are: Z(-2), Z(-1), Z(0), Z(1), Z(2), Z(3).

The array Y is dimensioned (6) and is explicitly typed real. It is not in common.

In subroutine IOTA, the adjustable dimension for array A is indicated by M. Whenever the main program calls IOTA, it can provide the name and the dimensions of the array; since A and M are dummy arguments, IOTA can be called repeatedly with different dimensions replacing M at each call. IOTA contains a DO loop which stores consecutive integers into the array A.

The main program calls subroutine IOTA three times. In the first call, the first argument is array X and the second argument is the number of elements in the array, 12. Consecutive integers are stored into the 12 elements of X.

In the second call to IOTA, the arguments (Y,6) are passed. Consecutive integers are stored into the six elements of Y.

In the third call to IOTA, the arguments (Z,6) are passed. The subscript bounds specified in the subroutine need not be the same as the ones specified in the calling program. Although Z is dimensioned (-2:3) in the main program, it can be dimensioned (6) in IOTA.

The PRINT statements output the arrays X, Y, and Z. The second PRINT statement illustrates the use of a negative DO index to output the array Z. The output is shown in figure 12-35.

PROGRAM ADIM2

ADIM2, shown in figure 12-36, is an extension of program ADIM. Subroutine IOTA is used; in addition, another subroutine and two functions are used. The following features are illustrated:

Parameter statement

Negative array subscripts

Negative DO parameters

Use of an expression for an array dimension

Multiple entry points

Adjustable dimensions

EXTERNAL statement

Passing values through COMMON

Use of intrinsic functions ABS and REAL

Calling functions through several levels

Passing a subprogram name as an argument

ARRAY X =	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
ARRAY Y =	1.	2.	3.	4.	5.	6.						
ARRAY Z =	1.	2.	3.	4.	5.	6.						
ARRAY Z =	-2.	-1.	0.	1.	2.	3.						

Figure 12-35. Program ADIM Output

```

PROGRAM ADIM2
C
C THIS PROGRAM USES ADJUSTABLE DIMENSIONS, NEGATIVE ARRAY BOUNDS,
C AND MANY SUBPROGRAM CONCEPTS
C
PARAMETER (I=4, J=3, K=-2, M=12, N=6)
COMMON X(I,J)
REAL Y(K:J)
EXTERNAL MULT, AVG
NAMelist /V/ X, Y, AA, AM
C
CALL SET (Y, N, 0.)
CALL IOTA (X, M)
CALL INC (X, M, -5.0)
AA = PVAL (M, AVG)
AM = PVAL (M, MULT)
PRINT V
STOP
END

C
SUBROUTINE SET (A, M, V)
C
C SET PUTS THE VALUE V INTO EVERY ELEMENT OF THE ARRAY A
C
DIMENSION A(*)
DO 1 I = 1, M
1 A(I) = 0.0
C
ENTRY INC (A, M, V)
C
C INC ADDS THE VALUE V TO EVERY ELEMENT IN THE ARRAY A
C
DO 2 I = 1, M
2 A(I) = A(I) + V
RETURN
END

C
SUBROUTINE IOTA (A, M)
C
C IOTA PUTS CONSECUTIVE NEGATIVE INTEGERS STARTING AT -1 INTO EVERY
C ELEMENT OF THE ARRAY A
C
DIMENSION A(-M:-1)
DO 1 I = -1, -M, -1
1 A(I) = I
RETURN
END

```

Figure 12-36. Program ADIM2 (Sheet 1 of 2)

```

C
      FUNCTION PVAL (ISIZE, WAY)
C
C   PVAL COMPUTES THE ABSOLUTE VALUE OF THE REAL VALUE OF A FUNCTION
C   PASSED TO PVAL.  ISIZE IS AN INTEGER WHICH PVAL PASSES TO THE
C   FUNCTION
C
      PVAL = ABS (WAY(ISIZE))
      RETURN
      END

C
      FUNCTION AVG (J)
C
C   AVG COMPUTES THE AVERAGE OF THE FIRST J ELEMENTS OF COMMON
C
      PARAMETER (M=4, N=3)
      COMMON A(M*N)
      AVG = 0.
      DO 1 I = 1, J
1     AVG = AVG + A(I)
      AVG = AVG/REAL(J)
      RETURN
      END

C
      REAL FUNCTION MULT (J)
C
C   MULT MULTIPLIES THE FIRST AND TWELFTH ELEMENTS OF COMMON AND
C   SUBTRACTS FROM THIS THE AVERAGE (COMPUTED BY THE FUNCTION AVG)
C   OF THE FIRST J/2 WORDS IN COMMON
C
      COMMON ARRAY(-1:10)
      MULT = ARRAY(10) * ARRAY(-1) - AVG(J/2)
      RETURN
      END

```

Figure 12-36. Program ADIM2 (Sheet 2 of 2)

Program ADIM2 illustrates the method of a main program calling subprograms and subprograms calling each other. Since the program is necessarily complex, each subprogram is described separately followed by a description of the main program.

Subroutine SET

Subroutine SET places the value V into every element of the array A. The dimension of A is specified by M.

Subroutine SET has an alternate entry point INC. When SET is entered at ENTRY INC, the value V is added to each element of the array A. The dimension of A is specified by M.

The first DO loop in subroutine SET clears the array to zero.

Subroutine IOTA

Subroutine IOTA is as described for program ADIM except that the input array A is given negative upper and lower subscript bounds. The DO loop uses negative control variables and places consecutive negative integers in A.

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 intrinsic function ABS.

The main program first calls PVAL with the statement AA=PVAL(M,AVG), passing the integer M (assigned the value 12 in the PARAMETER statement) and the function AVG as parameters. The type of the argument in the main program (INTEGER M) agrees with the corresponding dummy argument (ISIZE) in the subprogram.

The value of PVAL is computed in line 7. This value will be returned to the main program through the function name PVAL. Two functions are referenced by this statement; the intrinsic function ABS and the user-written function AVG. The actual arguments M and AVG replace ISIZE and WAY. The second time PVAL is called, the actual arguments M and MULT replace ISIZE and WAY.

Function AVG

This function computes the average of the first J elements of common. J is a value passed by the main program through the function PVAL.

This function subprogram is an example of a main program and a subprogram sharing values in common. The main program and function AVG declare common to be a total of 12 words. Values placed in common by the main program are available to the function subprogram.

The number of values to be averaged is passed to function PVAL by the statement AA=PVAL(12,AVG) and function PVAL passes this number (in ISIZE) to function AVG: PVAL=ABS(WAY(ISIZE)).

AVG uses a PARAMETER statement to assign symbolic names to the constants 4 and 3. These constants are then used in an expression that calculates the dimension for A. The expression itself is used as the dimension for A. AVG declares a total of 12 locations for common.

Lines 4 through 6 sum the 12 elements and divide by the number of elements to calculate the average. The intrinsic function REAL is used to convert the integer 12 to a real number to avoid mixed mode arithmetic, although in this case mixed mode is permissible and produces the same result.

The average is returned to the statement PVAL=ABS(WAY(ISIZE)) in function PVAL.

Function MULT

MULT multiplies the first and twelfth words in COMMON and subtracts the product from the average (computed by the function AVG) of the first J/2 words in common.

The declaration COMMON ARRAY (-1:10) assigns 12 elements to ARRAY and places it in common. The 12 elements are referenced by a subscript in the range -1 through 10. Line 8 multiplies the first element (ARRAY(-1)) by the twelfth element (ARRAY(10)) and subtracts the average (computed by function AVG) of the first J/2 elements in common.

Main Program: ADIM2

The main program calls the subroutines and functions described.

```

$V
X      = -.17E+02, -.16E+02, -.15E+02, -.14E+02, -.13E+02,
        -.12E+02, -.11E+02, -.1E+02,  -.9E+01,  -.8E+01,
        -.7E+01, -.6E+01,
Y      = 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
AA     = .115E+02,
AM     = .1165E+03,
$END

```

Figure 12-37. Program ADIM2 Output

The array Y has six elements, with subscript bounds of (-2:3). MULT and AVG appear in an EXTERNAL statement so that they can be passed to subprograms as arguments.

Lines 12 through 16 call the user-written subprograms SET, IOTA, and PVAL; CALL INC calls subroutine SET through the alternate entry point INC. The calls to PVAL pass a symbolic constant and a function name. Results are returned to AA and AM, respectively.

The namelist PRINT statement outputs the values calculated by the subprograms. The output is shown in figure 12-37.

PROGRAM CIRCLE

Program CIRCLE, shown in figure 12-38, finds the area of a circle which circumscribes a rectangle with short sides of length 3 and long sides of length 4. This example illustrates the use of FUNCTION subprograms and of statement functions. The program contains an error.

```

Program CIRCLE:

PROGRAM CIRCLE
A= 4.0
B= 3.0
AREA= 3.1416/4.0 * DIM(A,B)**2
PRINT 1, AREA
FORMAT (' AREA = ', G20.10)
STOP
END

Function DIM:

C
FUNCTION DIM(X, Y)
DIM= SQRT(X*X + Y*Y)
RETURN
END

Output:

AREA = .7854000000

```

Figure 12-38. Program CIRCLE, Function DIM, Output

Figure 12-39 shows a rectangle and circumscribed circle. The area of a circle is given by $\pi * R^2$, which is approximated by the FORTRAN expression:

$$3.1416/4.0*D**2$$

where R is the radius and D is the diameter of the circle.

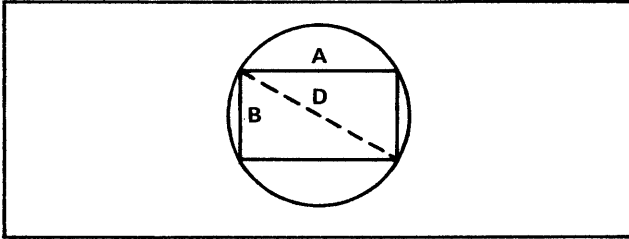


Figure 12-39. Rectangle and Circumscribed Circle

The user-written function DIM computes the diameter of the rectangle given the lengths of the sides using the relation:

$$DIM=SQRT(X*X + Y*Y)$$

The result shown in figure 12-38 is incorrect. The area of a circle circumscribing a rectangle with sides 3 and 4 is clearly greater than .785.

The error occurred because the function DIM has the same name as an intrinsic function. If the name of an intrinsic function is used for a user-written function, the user-written function is ignored.

There are several ways of correcting this error:

Change the function name so that it is not the same as an intrinsic function name.

Declare DIM external; in this case, the user-written external function will be used.

Write the function DIM as a statement function; the function name can be the same as an intrinsic function name, and the user-written function is used. This is the most efficient method. Since FORTRAN compiles statement functions in-line, the program executes much faster because no function call is used. This solution is limited to functions of a single statement.

A corrected version of the program, in which DIM is written as a statement function, is shown in figure 12-40.

PROGRAM BOOL

Program BOOL, shown in figures 12-41 and 12-42, illustrates some problems that can occur when Boolean constants are used in expressions. The program in figure 12-41 contains no type declaration for the variables A, B, C, D, and E. The program in figure 12-42 declares these variables type Boolean.

Program CIRCLE:

```
PROGRAM CIRCLE
DIM(X,Y)= SQRT(X*X + Y*Y)
A= 4.0
B= 3.0
AREA= 3.1416/4.0*DIM(A,B)**2
PRINT 1, AREA
1 FORMAT ('1AREA IS ',G20.10)
STOP
END
```

Output:

AREA IS 19.63500000

Figure 12-40. Program Circle With Correction and Output

Program BOOL:

```
PROGRAM BOOL
LOGICAL F
NAMELIST /OUT/A,B,C,D,E,F
A= 0*20
B= 0*10 + 0*10
C= B + 0*10
I= 5
D= I + 0*10
E= B + I + 0*10
F= A .EQ. 0*77
PRINT OUT
STOP
END
```

Output:

```
$OUT
A = 0.0,
B = .16E+02,
C = .16E+02,
D = .13E+02,
E = .21E+02,
F = T,
$END
```

Figure 12-41. Program BOOL and Output

Boolean constants include octal, hexadecimal, and Hollerith constants. When Boolean constants are used in expressions with operands of another type, no mode conversion occurs, and the result has the type of the other operand. Boolean operands used in arithmetic expressions are treated as type integer. For example, referring to figure 12-41, the statement:

B=0*10+0*10

```

Program BOOL:

PROGRAM BOUL
LOGICAL F
BOOLEAN A,B,C,D,E
NAMelist /OUT/A,B,C,D,E,F
A= 0"20"
B= 0"10" + 0"10"
C= B + 0"10"
I= 5
D= I + 0"10"
E= B + I + 0"10"
F= A .EQ. 0"77"
PRINT OUT
STOP
END

Output:

$OUT

A      = 0"20",
B      = 0"20",
C      = 0"30",
D      = 0"15",
E      = 0"35",
F      = F,

$END

```

Figure 12-42. Program BOUL With Correction and Output

is evaluated using integer arithmetic. Furthermore, for subsequent operations, the result of integer arithmetic is treated as true integer. Thus, in the above example, the expression on the right is evaluated using integer arithmetic; and the integer result is converted to real before the value is stored in B. Comparing the values produced for A and B illustrates this effect.

With floating-point arithmetic, whenever the left 12 bits of the computer word are all zeros or all ones, the value of that number is zero. (See Program EQUIV.) This explains why the output value of A in figure 12-41 is zero.

The remaining expressions are evaluated as follows:

$$C=B+0"10"$$

Floating-point arithmetic is used to evaluate the expression; the octal constant 0"10" is used without type conversion, making its value zero. Note in the output from BOUL the values of B and C are equal.

$$D=I+0"10"$$

No problem arises in the preceding expression as it is evaluated with integer arithmetic; then the result is converted to real and stored in D.

$$E=B+I+0"10"$$

The compiler, in scanning the preceding expression left to right, encounters the real variable B and uses real arithmetic to evaluate the expression. Again, the octal constant 0"10" has the real value of zero.

If the expression were written as:

$$E=0"10"+I+B \text{ or } E=I+0"10"+B$$

the first two terms would be added using integer arithmetic; then that result would be converted to real and added to B. In this case, the octal constant 0"10" would effectively have the value eight.

This is similar to the mode conversion which occurs in:

$$X=Y*3/5 \text{ or } Z=3/5*Y$$

These expressions would give different values for X and Z. More information on the evaluation of mixed mode expressions is presented in section 3.

$$F=A .EQ. 0"77"$$

Real arithmetic is used to compare the value because A is a type real name. The value in A and the constant 0"77" both have all zeros in the leftmost 12 bits; both have value zero for real arithmetic; therefore, the value assigned to F is .TRUE.

To avoid the confusion illustrated in this example, simply use a type Boolean declaration for variable names whose values come from octal, hexadecimal, or Hollerith constants. Figure 12-42 shows the same program with the names A, B, C, D, and E all as type Boolean.

All these examples use octal constants; however, the same problem occurs with Hollerith, especially when it is right-justified, and with hexadecimal. The program segment in figure 12-43 illustrates the point.

```

REAL ANS
.
.
.
READ 2, ANS
2 FORMAT (R3)
IF(ANS .EQ. R"NO ") PRINT 3
3 FORMAT (' NEGATIVE RESPONSE')
.
.
.

```

Figure 12-43. Hollerith Examples

PROGRAM EASY IO

Program EASY IO, shown in figure 12-44, illustrates the use of list directed input/output.

```

      PROGRAM EASY IO
C
C   GIVEN THE SIDES OF A TRIANGLE, COMPUTE THE AREA AND RADIUS OF THE
C   INSCRIBED CIRCLE
C
      REAL SIDES(3)
      EQUIVALENCE (SIDES(1),A), (SIDES(2),B), (SIDES(3),C)
      NAMELIST /OUT/ SIDES, AREA, RADIUS
3     READ (*, *, END=50) SIDES
      S= (A + B + C)/2.0
      AREA= SQRT(S*(S-A) * (S-B) * (S-C))
      RADIUS= AREA/S
      WRITE (*, OUT)
      GO TO 3
50    STOP
      END

```

Figure 12-44. Program EASYIO

List directed input/output eliminates the need for fixed data fields. It is especially useful for input since the user need not be concerned with punching data in specific columns. List directed input does not require the user to name each item as does NAMELIST input.

Used in combination, list directed input and NAMELIST output simplify program design. Such a program is easy to write, even for persons just learning the language; knowledge of the format specifications is not required. This feature is particularly useful when FORTRAN programs are being run from a remote terminal.

Program EASY IO calculates the area and radius of a circle inscribed in a triangle, given the lengths of the sides of the triangle. A list directed READ statement is used for input, and NAMELIST is used for output. Figure 12-45 shows some sample input and output.

The user can enter the three input values in whatever way is convenient, such as: one item per line (or card), one item per line with each item followed by a comma, all items on a single line with spaces separating each item, all items on a line with a comma and several spaces separating each item, or any combination of the foregoing. Furthermore, even though all input items are real, the decimal point is not required when the input value is a whole number.

PROGRAM BLOCK

Program BLOCK, shown in figure 12-46, illustrates block IF structures.

Block IF structures allow the user to specify alternate paths of execution, based on the outcome of IF tests. Block IF structures eliminate the need for branching when IF tests are performed. This feature can make programs simpler and more readable.

Program BLOCK reads an integer into the variable K, and two sets of real numbers into the arrays A and B. K is tested and the following action is taken:

K=1

Calculate $C(I)=A(I)**2 + B(I)**2$.

K=2

Calculate $C(I)=A(I)*B(I)$.

All other values of K

Set array C to zero.

These tests could be performed by conventional methods, using logical IF and GO TO statements. However, with block IF structures the program is much clearer.

The program includes a block IF statement (line 7), and ELSE IF statement (line 11), and an ELSE statement (line 15). These statements provide for three alternate paths of execution. After the appropriate block has been executed, control transfers to the WRITE statement following END IF. The program then branches back to process the next input record.

Sample input and output are shown in figure 12-47.

PROGRAMS ONE AND TWO

Programs ONE and TWO, shown in figure 12-48, illustrate internal file usage.

Program ONE writes a single record to an internal file. The array A and the variables B and C are declared type character of length 10. The character variable ALPHA, to be used as the internal file, has length 40. The DATA statement loads character data into A, B, and C.

```

Input:
3 4 5
6,7,8
3*1
4
5
6

Output:
$OUT
SIDES = .3E+01, .4E+01, .5E+01,
AREA = .6E+01,
RADIUS = .1E+01,
$END

$OUT
SIDES = .6E+01, .7E+01, .8E+01,
AREA = .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 = .1322875655323E+01,
$END

```

Figure 12-45. Sample Input and Output for Program EASYIO

The WRITE statement defines ALPHA to be an internal file and writes the values of A, B, and C to the file according to the format specification (2A4, A5, A6). The following formatting is performed:

Characters ABCD from A(1) are transmitted to positions 1 through 4 of ALPHA.

Characters KLMN from A(2) are transmitted to positions 5 through 8 of ALPHA.

Characters UVWXY from B are transmitted to positions 9 through 13 of ALPHA.

```

PROGRAM BLOCK
PARAMETER (M=5)
DIMENSION A(M), B(M), C(M)
NAMelist /OUT/ K, A, B, C
C
2 READ (*, *, END=100) K, A, B
IF (K .EQ. 1) THEN
DO 5 I = 1,M
5 C(I)= A(I)**2 + B(I)**2
C
ELSE IF (K .EQ. 2) THEN
DO 10 I = 1,M
10 C(I)= A(I)*B(I)
C
ELSE
DO 15 I = 1,M
15 C(I)= 0.0
C
END IF
WRITE (*, OUT)
GO TO 2
100 STOP
END

```

Figure 12-46. Program BLOCK

Characters Z12345 from C are transmitted to positions 14 through 19 of ALPHA.

Positions 20 through 40 of ALPHA are blank filled.

Program TWO is identical to program ONE except that ALPHA is dimensioned 2 and the format specification is changed to cause two records to be written to ALPHA. The characters in A(1) and A(2) are transmitted to ALPHA(1) as before. The slash, however, causes subsequent data to be transmitted to ALPHA(2). Unused portions of both records are blank filled.

PROGRAM PMD2

Program PMD2, shown in figure 12-49, illustrates the use of the Post Mortem Dump facility. In this example, the dump is triggered by a program abort.

Program PMD2 consists of a main program and a subroutine. The main program contains an error: in the CALL statement, the subroutine name SETCOM is misspelled as SETCM. This error causes the program to abort when the statement CALL SETCM is executed.

Subroutine SETCOM tests the logical variable L. If L contains the value .TRUE., data is read from unit 1 into the array B. If L contains the value .FALSE., B is set to zero.

Note that the program contains no calls to Post Mortem Dump routines. In this case, if the program aborts and DB=PMD was selected, a dump occurs automatically.

The Post Mortem Dump output for program PMD2 is shown in figure 12-50. The dump includes an error analysis, a description of current file status, and an analysis of variables in the main program (in which the error occurred).

```

Input:
5 9.0 9.0 8.0 8.0 7.0 5.0 3.0 3.0 2.0 6.0
1 1.0 0.0 0.0 7.0 7.0 4.0 0.0 0.0 0.0 0.0
4 4.0 4.0 4.0 7.0 8.0 5.0 0.0 0.0 3.0 2.0
3 3.0 3.0 2.0 2.0 1.0 6.0 8.0 0.0 1.0 1.0

```

```

Output:
$OUT
K      = 1,
A      = .1E+01, 0.0, 0.0, .7E+01, .7E+01,
B      = .4E+01, 0.0, 0.0, 0.0, 0.0,
C      = .17E+02, 0.0, 0.0, .49E+02, .49E+02,
$END
$OUT
K      = 4,
A      = .4E+01, .4E+01, .4E+01, .7E+01, .8E+01,
B      = .5E+01, 0.0, 0.0, .3E+01, .2E+01,
C      = 0.0, 0.0, 0.0, 0.0, 0.0,
$END
$OUT
K      = 3,
A      = .3E+01, .3E+01, .2E+01, .2E+01, .1E+01,
B      = .6E+01, .8E+01, 0.0, .1E+01, .1E+01,
C      = 0.0, 0.0, 0.0, 0.0, 0.0,
$END

```

Figure 12-47. Sample Input and Output for Program BLOCK

Example 1:

```

PROGRAM ONE
CHARACTER A(2)*10,B*10,C*10,ALPHA*40
DATA A,B,C /'ABCDEFGH IJ', 'KLMNOPQRST', 'UVWXY', 'Z123456'/
WRITE (ALPHA, '(2A4,A5,A6)') A.R.C
PRINT 2,ALPHA
2 FORMAT ('CONTENTS OF ALPHA = ', /1X, A40)
STOP
END

```

Output:

```

CONTENTS OF ALPHA =
ABCDKLMNUVWXYZ12345Δ ----- Δ
          SECONDS EXECUTION TIME.

```

A single record is written to the internal file ALPHA.

Figure 12-48. Programs ONE and TWO (Sheet 1 of 2)

Example 2:

```
PROGRAM TWO
CHARACTER A(2)*10,B*10,C*10,ALPHA(2)*40
DATA A,B,C /'ABCDEFGH IJ','KLMNOPQRST','UVWXY','Z123456'/
WRITE (ALPHA,'(2A4/A5,A6)') A,B,C
PRINT 2,ALPHA
2 FORMAT ('1CONTENTS OF ALPHA = ', /1X, 2A40)
STOP
END
```

Output:

```
CONTENTS OF ALPHA =
| ABCDKLMNΔ ----- Δ UVWXYZ12345Δ --- |
```

record 1

record 2

Two records are written to the internal file ALPHA.

Figure 12-48. Programs ONE and TWO (Sheet 2 of 2)

```
PROGRAM PMD2
C
C THIS PROGRAM CONTAINS AN ERROR WHICH ACTIVATES POST MORTEM DUMP
C IF DB=PMD IS SELECTED
C
CHARACTER*10 FILE, IFG
LOGICAL LVAR
COMMON /CBLCK/ ARR(3,3)
C
OPEN (UNIT=6,FILE='OUTPUT')
LVAR = .TRUE.
CALL SETCM (LVAR, IFG)
WRITE (6,*) IFG, ARR
STOP
END
C
SUBROUTINE SETCOM (L, IFG)
LOGICAL L
CHARACTER*10 IFG
COMMON /CBLCK/ B(3,3)
C
IF (L) THEN
IFG = 'FIRST'
READ (1,END=999) ((B(I,J),I=1,3),J=1,3)
ELSE
IFG = 'SECOND'
DO 10 I=1,3
DO 10 J=1,3
10 B(I,J) = 0.0
ENDIF
RETURN
999 END
```

Figure 12-49. Program PMD2

FTN POST MORTEM DUMP

ERROR REPORT

79/08/20. 12.57.28.

*** YOUR JOB HAS THE FOLLOWING NON-FATAL LOAD ERROR(S):
 UNSATISFIED EXTERNAL REF -- SETCM

/// EXECUTION WAS TERMINATED BECAUSE YOUR PROGRAM CALLED A MISSING ROUTINE AT LINE NUMBER 12 OF PROGRAM PMD2

... ARRAYS WILL BE PRINTED BY DEFAULT PARAMETERS (20, 2, 1, 1, 1, 1, 1)

... YOUR PROGRAM REQUIRED 26300B WORDS TO LOAD, 10315B WORDS TO RUN

... FILE STATUS AT TIME OF TERMINATION

FILE NAME -OUTPUT	FURTRAN NAMES TAPE6	LAST OP OPENED	STATUS	FILE TYPE SQ	BLOCKING TYPE C	REC TYPE Z	RECORD COUNT 0
----------------------	------------------------	-------------------	--------	-----------------	--------------------	---------------	-------------------

... VARIABLES IN NAME	PROGRAM TYPE	PMD2 RELOCATION	CURRENT VALUE	COMMENTS	NAME
--------------------------	-----------------	--------------------	---------------	----------	------

ARR	REAL	/CBLOCK/	ARRAY		ARR
... DIMENSIONED AS - ARR(1:3,1:3)					
*** THE NEXT ITEM IS NEVER DEFINED	FILE	CHARACT	PMD2 :AW		FILE
	IFG	CHARACT	:::E/DU5A%		IFG
	LVAR	LOGICAL	.TRUE.		LVAR

... ARRAYS IN PROGRAM PMD2

REAL	ARRAY	ARR(1:3,1:3)		
(ARR(N,1))				
N=1	NOT INITIALIZED	NOT INITIALIZED	NOT INITIALIZED	
(ARR(N,2))				
N=1	NOT INITIALIZED	NOT INITIALIZED	NOT INITIALIZED	

... TRACEBACK SUCCESSFULLY COMPLETED

/// END OF ERROR REPORT

Figure 12-50. Post Mortem Dump Output for Program PMD2

PROGRAM PMD

Program PMD, shown in figure 12-51, illustrates the use of the Post Mortem Dump. In this example, Post Mortem Dump calls are used to trigger a dump. Post Mortem Dump routines illustrated are:

PMDARRY
 PMDLOAD
 PMDDUMP

```

PROGRAM PMD
DIMENSION A(50), B(50), C(50)
DATA A/50*2.0/, B/50*4.0/
C
CALL PMDARRY(5)
C
DO 10 I = 1,50,2
10  A(I) = A(I) + B(I)
CALL SUBT (A,B,C,50)
C
CALL PMDLOAD
C
STOP
END
C
SUBROUTINE SUBT (X,Y,Z,M)
DIMENSION X(M), Y(M), Z(M)
DO 16 I = 1,M
16  Z(I) = SQRS(X(I),Y(I))
C
CALL PMDDUMP
C
RETURN
END
C
FUNCTION SQRS(R,S)
SQRS = R*R + S*S
C
CALL PMDDUMP
C
RETURN
END
  
```

Figure 12-51. Program PMD

Program PMD consists of a main program, a subroutine, and a function subprogram. These program units perform some simple operations on values stored in an array. The call to PMDARRY in the main program specifies that only 1-dimensional arrays are to be dumped and that dumps of arrays are to be limited to the first five elements, although the arrays are dimensioned 50. The call to PMDLOAD in line 11 causes a dump of variables in the main program and in any routines that have called PMDDUMP.

Subroutine SUBT and function SQRS each contain a call to PMDDUMP. After these calls are executed, the call to PMDLOAD in the main program causes variables in SUBT and SQRS to be dumped following the variables of the main program.

The Post Mortem Dump output is shown in figure 12-52. The dump includes an analysis of variables and traceback information for each program unit.

PROGRAM DBUG

Program DBUG, shown in figure 12-53, illustrates the use of CYBER Interactive Debug (CID) to conduct an interactive debug session (not supported on SCOPE 2). The CID commands illustrated are:

SET,BREAKPOINT
 GO
 PRINT
 QUIT

Program DBUG stores numbers into an array A and stores a character string into a variable CHAR. The program is compiled and executed interactively in debug mode.

The terminal session for NOS/BE is shown in figure 12-54 (CID and system output are in uppercase, user input is in lowercase). The DEBUG control statement establishes debug mode. When the program is compiled in debug mode, special tables are generated for use by CID. The execution control statement LGO initiates the debug session. CID responds with:

CYBER INTERACTIVE DEBUG
 ?

allowing the user to enter CID commands. The SET,BREAKPOINT command sets a breakpoint that causes execution to be suspended when line 9 is reached. The GO command initiates execution of the program. The message:

*B #1, AT L.9
 ?

indicates that a breakpoint has suspended execution at line 9 and that CID is waiting for user input. Note that execution is suspended before the statement in line 9 is executed (the PRINT command shows that CHAR still contains the value assigned by the DATA statement).

The GO command is then entered to resume program execution. The message:

*T #17, END IN L.10
 ?

is a trap message indicating that the program has terminated at line 10 and that CID commands can be entered.

The QUIT command ends the debug session. Debug mode, however, remains in effect until DEBUG(OFF) is entered.

FTN POST MORTEM DUMP

ERROR REPORT

79/08/20. 12.59.04.

*** YOUR JOB HAS THE FOLLOWING NON-FATAL LOAD ERROR(S):
 UNSATISFIED EXTERNAL REF -- SETCM

/// EXECUTION WAS INTERRUPTED BECAUSE YOUR PROGRAM CALLED PMDLOAD AT LINE NUMBER 11 OF PROGRAM PMD
 ... ARRAYS WILL BE PRINTED BY REQUESTED PARAMETERS (5, 0, 0, 0, 0, 0, 0)
 ... YOUR PROGRAM REQUIRED 27200B WORDS TO LOAD, 11105B WORDS TO RUN

NAME	TYPE	RELOCATION	CURRENT VALUE	COMMENTS	NAME
A	REAL		ARRAY		A
... DIMENSIONED AS - A(1:50)					
B	REAL		ARRAY		B
... DIMENSIONED AS - B(1:50)					
C	REAL		ARRAY		C
... DIMENSIONED AS - C(1:50)					
I	INTEGER		51 = 1R%		I

... ARRAYS IN PROGRAM PMD

REAL (A(N))	REAL (B(N))	REAL (C(N))
REAL ARRAY A(1:50)	REAL ARRAY B(1:50)	REAL ARRAY C(1:50)
N=1 6.00000000000 2.00000000000 6.00000000000 2.00000000000 6.00000000000	N=1 4.00000000000 4.00000000000 4.00000000000 4.00000000000 4.00000000000	N=1 52.00000000000 20.00000000000 52.00000000000 20.00000000000 52.00000000000

FTN POST MORTEM DUMP

FUNCTION SQRS

79/08/20. 12.59.04.

... CURRENT SITUATION IN FUNCTION SQRS

NAME	TYPE	RELOCATION	CURRENT VALUE	COMMENTS	NAME
R	REAL	F.P. 1	2.00000000000		R
S	REAL	F.P. 2	4.00000000000		S

... CALLED FROM LINE NUMBER 4 OF SUBROUTINE SUBT

Figure 12-52. Post Mortem Dump Output for Program PMD (Sheet 1 of 2)

FTN POST MORTEM DUMP

SUBROUTINE SUBT

79/08/20. 12.59.04.

... CURRENT SITUATION IN SUBROUTINE SUBT

... VARIABLES IN SUBROUTINE SUBT

NAME	TYPE	RELOCATION	CURRENT VALUE	COMMENTS	NAME
I	INTEGER		51	= 1R%	I
M	INTEGER	F.P. 4	50	= 1R]	M
X	REAL	F.P. 1	ARRAY		X
... DIMENSIONED AS - X(1:50)					
Y	REAL	F.P. 2	ARRAY		Y
... DIMENSIONED AS - Y(1:50)					
Z	REAL	F.P. 3	ARRAY		Z
... DIMENSIONED AS - Z(1:50)					

... ARRAYS IN SUBROUTINE SUBT

REAL ARRAY X(1:50)
 VARIABLE SPAN IN SUBSCRIPTS 1
 (X(N))

N=1	6.0000000000	2.0000000000	6.0000000000	2.0000000000	6.0000000000
-----	--------------	--------------	--------------	--------------	--------------

REAL ARRAY Y(1:50)
 VARIABLE SPAN IN SUBSCRIPTS 1

ALL REQUESTED ELEMENTS OF THIS ARRAY WERE 4.0000000000

REAL ARRAY Z(1:50)
 VARIABLE SPAN IN SUBSCRIPTS 1
 (Z(N))

N=1	52.0000000000	20.0000000000	52.0000000000	20.0000000000	52.0000000000
-----	---------------	---------------	---------------	---------------	---------------

... CALLED FROM LINE NUMBER 9 OF PROGRAM PMD

/// END OF ERROR REPORT

Figure 12-52. Post Mortem Dump Output for Program PMD (Sheet 2 of 2)

```

1          PROGRAM DEBUG      74/74  OPT=0

1          PROGRAM DEBUG
2          DIMENSION A(-1:4)
3          CHARACTER*5 CHAR
4          DATA CHAR /'ABCDE'/
5          C
6          DO 12 I=-1,4
7          12  A(I) = I
8          C
9          CHAR = 'XYZ12'
10         STOP
11        END

```

Figure 12-53. Program DEBUG

```

COMMAND- debug ← Activate debug mode
COMMAND- ftn5,i=prog,l=list ← Compile program
57300 CM STORAGE USED.
0.112 CP SECONDS.
COMMAND- lgo ← Initiate debug session
CYBER INTERACTIVE DEBUG
?set,breakpoint,l.9 ← Set a breakpoint at line 9
?go ← Begin program execution
*B #1, AT L.9 ← Breakpoint detected at line 9, execution suspended
?print*, a,char ← Display values of A and CHAR
-1. 0. 1. 2. 3. 4. ABCDE
?go ← Resume execution
*T #17, END IN L.10 ← Program terminates at line 10
?
STOP
12000B MAX FL DURING EXECUTION.
.167 CP SECONDS EXECUTION TIME
print*, char ← Display value of CHAR
XYZ12
?quit ← Terminate debug session
DEBUG TERMINATED
COMMAND- debug(off) ← Turn off debug mode
COMMAND-

```

Figure 12-54. Debug Session

PROGRAM GOTO

Program GOTO, shown in figure 12-55, illustrates the computed GO TO feature.

Program GOTO reads records containing a single integer each and keeps a running total of the number of integers falling within the intervals 0 through 25, 26 through 50, 51 through 75, and 76 through 100. If the integer does not fall within any of these intervals an appropriate message is printed. When all records have been read, the total for each interval is printed.

In the computed GO TO statement in line 13, the control index is an expression $(NUM + 24)/25$. If the input value NUM is in the range 1 through 100, the value of the expression is in the range 1 through 4. The computed GO TO transfers control to the label 20, 30, 40, or 50, if the value of the expression is 1, 2, 3, or 4 respectively. The appropriate counter is then incremented. If the value of the expression is less than 1 or greater than 4, control passes to the PRINT statement following the GO TO.

Sample input and output are shown in figure 12-56.

```

PROGRAM GOTO
C
C PROGRAM GOTO READS INTEGERS RANGING FROM 1 TO 100, DIVIDES THEM INTO
C FOUR GROUPS, AND DETERMINES THE NUMBER IN EACH GROUP
C
      NGRP1 = 0
      NGRP2 = 0
      NGRP3 = 0
      NGRP4 = 0
C
10  READ (*, *, END=100) NUM
      IF (NUM .EQ. 0) NUM = 1
      GO TO (20,30,40,50), (NUM + 24)/25
      PRINT (' " NUMBER ",14," IS OUT OF CORRECT RANGE"'), NUM
      GO TO 10
20  NGRP1 = NGRP1 + 1
      GO TO 10
30  NGRP2 = NGRP2 + 1
      GO TO 10
40  NGRP3 = NGRP3 + 1
      GO TO 10
50  NGRP4 = NGRP4 + 1
      GO TO 10
C
100 PRINT 200, NGRP1, NGRP2, NGRP3, NGRP4
200 FORMAT (' 0 - 25 :', 14,/1X,'26 - 50 :', 14,/1X,'51 - 75 :', 14,
* /1X, '76 - 100:',14)
      STOP
      END

```

Figure 12-55. Program GOTO

Input:

```

56
30
110
2
25
-10
0
100
81

```

Output:

```

NUMBER 110 IS OUT OF CORRECT RANGE
NUMBER -10 IS OUT OF CORRECT RANGE
0 - 25 : 3
26 - 50 : 1
51 - 75 : 1
76 - 100: 2

```

Figure 12-56. Sample Input and Output for Program GOTO

```

PROGRAM ASK
LOGICAL E, O
CHARACTER*10 N,A,S,F*11
DIMENSION BUF(50)

C
DO 10 I = 1,50
10  BUF(I) = I
C
OPEN (2, ERR=99, FILE='N123', STATUS='NEW', ACCESS='DIRECT',
* RECL=100 )
K=1
DO 15 I = 1,41,10
15  WRITE (2, REC=K, ERR=99) (BUF(J),J=I,1+9)
C      K = K + 1
C
INQUIRE (UNIT=2, ERR=99, EXIST=E, OPENED=O, NAME=N, ACCESS=A,
* SEQUENTIAL=S, FORM=F, RECL=L, NEXTREC=M)
C
PRINT 50, E,O,N,A,S,F,L,M
50  FORMAT ('1', 'UNIT EXISTS? ', L1, /' UNIT ASSOCIATED WITH FILE? ',
* L1, /' FILE NAME IS ', A, /' ACCESS METHOD IS ', A,
* /' SEQUENTIAL ACCESS PERMITTED? ', A,
* /' OPENED FOR ', A, ' I/O', /' RECORD LENGTH IS ', 15,
* /' NEXT RECORD IS ', 15)
C
CLOSE (2, ERR=99, STATUS='DELETE')
STOP
99  PRINT*, ' FATAL I/O ERROR'
STOP
END

```

Figure 12-57. Program ASK

PROGRAM ASK

Program ASK, shown in figure 12-57, illustrates the OPEN, INQUIRE, and CLOSE statements. The program creates a file, writes information to the file, inquires about the status of the file, and closes the file.

The OPEN statement in lines 9 and 10 creates a file named N123 and associates the file with unit 2. File N123 is declared to be a direct access file with a record length of 100 words.

The DO loop in lines 12 through 15 writes 5 records to file N123. One record is written on each pass through the loop. Each record consists of ten consecutive words from the array BUF followed by blank fill. Since N123 is a direct access file, the REC parameter is specified on the WRITE statement to assign a number to each record. A counter K is incremented on each pass through the loop, and the value of K is used for the record number.

The INQUIRE statement in line 16 performs an inquire on unit 2. INQUIRE returns information in the variables supplied for the specified parameters. The variables O and E are declared type logical because INQUIRE returns a logical value (T or F) for the EXIST and OPENED parameters. Variables N, A, S, F, and B are declared type character because INQUIRE returns a character string for the NAME, ACCESS, SEQUENTIAL, and FORM parameters.

Program output is shown in figure 12-58. The FORMAT statement formats the output so that it is self-explanatory. Note that sequential access is

```

UNIT EXISTS? T
UNIT ASSOCIATED WITH FILE? T
FILE NAME IS N123
ACCESS METHOD IS DIRECT
SEQUENTIAL ACCESS PERMITTED? NO
OPENED FOR UNFORMATTED I/O
RECORD LENGTH IS 100
NEXT RECORD IS 6

```

Figure 12-58. Program ASK Output

not permitted on file N123. The file is opened for unformatted output (default for direct access files), the next record is 6 (5 records have been written), and blanks within a record are ignored (default). The NAME, ACCESS, and RECL parameters reflect information specified on the OPEN statement.

The CLOSE statement in line 26 specifies the STATUS='DELETE' parameter so that the file is destroyed after execution of the CLOSE. If this statement were omitted, an implicit CLOSE(2,STATUS='KEEP') would occur.

PROGRAM SCORE

Program SCORE, shown in figure 12-59, reads student names and test scores from input records and calls subroutine AVG to compute the average of the scores on each record and to determine which of the students qualify for honors. Program SCORE illustrates the use of an alternate return.


```

PROGRAM SCORE
CHARACTER*10 NAME
DIMENSION ISCORE(4)
DATA XLIM/90.0/, N/4/

C
6   READ (*, 100, END=12) NAME, (ISCORE(I),I=1,4)
100 FORMAT (A10, 4I3)
    CALL AVG (ISCORE, N, XLIM, AV, *8, *10)
8   PRINT '(21X, "HONORS")'
10  PRINT '("+", A, 3X, F6.2, /)', NAME, AV
    GO TO 6

C
12  STOP
    END

C
SUBROUTINE AVG(IARR, N, XLIM, AV, *, *)
DIMENSION IARR(N)

C
SUM = 0
DO 20 I = 1,N
20  SUM = SUM + IARR(I)
AV = SUM/N
IF (AV .GE. XLIM) RETURN 1
RETURN 2
END

```

Figure 12-59. Program SCORE and Subroutine AVG

Each input record contains a name and four test scores. After reading a record, the main program calls subroutine AVG which computes the average of the four scores. The actual arguments passed to AVG are an array ISCORE containing the four scores, an integer variable N containing the number of scores, a real variable XLIM, a real variable AV in which AVG returns the computed average, and two statement labels indicated by *8 and *10.

The variables XLIM and N are initialized by the DATA statement in line 4.

Subroutine AVG computes the average of the values in ISCORE and tests the average against XLIM to determine if the student qualifies for honors. The IF statement in line 8 performs the test and returns control to the statement label represented by the first asterisk in the SUBROUTINE statement (label 8) if the test has a value that is true. If the test is not true, control passes to the next statement which returns control to the statement label represented by the second asterisk in the SUBROUTINE statement (label 10).

In the main program, the statement labeled 8 prints the string "HONORS". The statement labeled 10 prints the name and the computed average; the + carriage control character causes these values to appear on the same line as "HONORS".

The program continues to process input records until an end-of-file is detected, at which time control passes to the statement labeled 99 and execution terminates.

Sample input and output for program SCORE are shown in figure 12-60.

Input:	
SMITH	98 85 89 92
JONES	75 83 80 89
DOE	85 92 95 89
DOAKES	85 89 80 91
Output:	
SMITH	91.00 HONORS
JONES	81.75
DOE	90.25 HONORS
DOAKES	86.25

Figure 12-60. Sample Input and Output for Program SCORE

STANDARD CHARACTER SETS

A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation is specified when the operating system is installed. The standard character sets are shown in table A-1.

Depending on another installation option, NOS and NOS/BE assume an input deck has been punched either in 026 or 029 mode, regardless of the character set in use. Under NOS, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Under NOS/BE, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified alternate mode remains in effect throughout the job unless reset by another alternate mode specification.

Graphic character representation on a terminal or printer depends on the installation character set and the device type. CDC graphic characters in table A-1 are applicable to BCD terminals. ASCII subset graphic characters are applicable to ASCII-CRT, ASCII-TTY terminals, and line printers that have the ASCII graphics print-train.

Under SCOPE 2, the alternate modes are: 026, 029, and blank.

The 026 and 029 modes are specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The 26 and 29 codes convert 026 and 029 coded input to display code. Blank entries in columns 79 and 80 indicate that the following section is coded or binary and the next card should be checked according to these alternatives:

If the next card is a free-form flag card, the section following is free-form binary. (See the SCOPE 2 reference manual.)

If the next card has 7/9 punched (only) in column 1, the following section is SCOPE 2 binary. (See the SCOPE 2 reference manual.)

In any other case, the following section is coded with the last requested conversion mode.

When a 63-character set is in use, display code 00 under A or R edit descriptor conversion in a formatted I/O statement, ENCODE statement, or DECODE statement is converted to display code 55 octal (blank). No conversions occur when a 64-character set is in use.

FORTRAN programs can be written to handle 95-character or 128-character ASCII. In general, NOS handling of 95-character or 128-character ASCII involves 6-bit and 12-bit codes, with characters represented in a single display code or double display code combination. The NOS character codes are shown in table A-2. In general, NOS/BE and INTERCOM handling of 95-character or 128-character ASCII involves 8-bit and 12-bit codes, with the 8-bit ASCII code right-justified in a 12-bit field. The ASCII character set is shown in table A-3. See the appropriate operating system manual.

TABLE A-1. FORTRAN AND STANDARD CHARACTER SETS

FORTRAN	Display Code (octal)	CDC			ASCII		
		Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
:	00 [†]	: (colon) ^{††}	8-2	00	: (colon) ^{††}	8-2	072
A	01	A	12-1	61	A	12-1	101
B	02	B	12-2	62	B	12-2	102
C	03	C	12-3	63	C	12-3	103
D	04	D	12-4	64	D	12-4	104
E	05	E	12-5	65	E	12-5	105
F	06	F	12-6	66	F	12-6	106
G	07	G	12-7	67	G	12-7	107
H	10	H	12-8	70	H	12-8	110
I	11	I	12-9	71	I	12-9	111
J	12	J	11-1	41	J	11-1	112
K	13	K	11-2	42	K	11-2	113
L	14	L	11-3	43	L	11-3	114
M	15	M	11-4	44	M	11-4	115
N	16	N	11-5	45	N	11-5	116
O	17	O	11-6	46	O	11-6	117
P	20	P	11-7	47	P	11-7	120
Q	21	Q	11-8	50	Q	11-8	121
R	22	R	11-9	51	R	11-9	122
S	23	S	0-2	22	S	0-2	123
T	24	T	0-3	23	T	0-3	124
U	25	U	0-4	24	U	0-4	125
V	26	V	0-5	25	V	0-5	126
W	27	W	0-6	26	W	0-6	127
X	30	X	0-7	27	X	0-7	130
Y	31	Y	0-8	30	Y	0-8	131
Z	32	Z	0-9	31	Z	0-9	132
0	33	0	0	12	0	0	060
1	34	1	1	01	1	1	061
2	35	2	2	02	2	2	062
3	36	3	3	03	3	3	063
4	37	4	4	04	4	4	064
5	40	5	5	05	5	5	065
6	41	6	6	06	6	6	066
7	42	7	7	07	7	7	067
8	43	8	8	10	8	8	070
9	44	9	9	11	9	9	071
+ (plus)	45	+	12	60	+	12-8-6	053
- (minus)	46	-	11	40	-	11	055
* (asterisk)	47	*	11-8-4	54	*	11-8-4	052
/ (slash)	50	/	0-1	21	/	0-1	057
((left paren)	51	(0-8-4	34	(12-8-5	050
) (right paren)	52)	12-8-4	74)	11-8-5	051
\$ (currency)	53	\$	11-8-3	53	\$	11-8-3	044
= (equals)	54	=	8-3	13	=	8-6	075
blank	55	blank	no punch	20	blank	no punch	040
, (comma)	56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
. (decimal point)	57	. (period)	12-8-3	73	. (period)	12-8-3	056
	60	≡	0-8-6	36	#	8-3	043
	61	[8-7	17	[12-8-2	133
	62]	0-8-2	32]	11-8-2	135
	63	% ^{††}	8-6	16	% ^{††}	0-8-4	045
" (quote)	64	"	8-4	14	" (quote)	8-7	042
	65	⎵	0-8-5	35	⎵ (underline)	0-8-5	137
	66	∇	11-0	52	!	12-8-7	041
	67	∧	0-8-7	37	&	12	046
' (apostrophe)	70	'	11-8-5	55	' (apostrophe)	8-5	047
	71	↓	11-8-6	56	?	0-8-7	077
	72	<<	12-0	72	<	12-8-4	074
	73	>>	11-8-7	57	>	0-8-6	076
	74	<	8-5	15	@	8-4	100
	75	∇	12-8-5	75	\	0-8-2	134
	76	∩	12-8-6	76	˘ (circumflex)	11-8-7	136
	77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

[†]Twelve zero bits at the end of a 60-bit word in a zero byte record are an end-of-record mark rather than two colons.
^{††}In installations using a 63-graphic set, display code 00_g has no associated graphic or card code; display code 63_g is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55_g).

TABLE A-2. CODES (6/12-BIT) FOR NOS

Display Code (6/12-Bit Octal)	Char.	ASCII Code (7-Bit Octal)	ASCII Code (Hexadecimal)	Display Code (6/12-Bit Octal)	Char.	ASCII Code (7-Bit Octal)	ASCII Code (Hexadecimal)
00 [†]	:	072	3A	7604	d	144	64
01	A	101	41	7605	e	145	65
02	B	102	42	7606	f	146	66
03	C	103	43	7607	g	147	67
04	D	104	44	7610	h	150	68
05	E	105	45	7611	i	151	69
06	F	106	46	7612	j	152	6A
07	G	107	47	7613	k	153	6B
10	H	110	48	7614	l	154	6C
11	I	111	49	7615	m	155	6D
12	J	112	4A	7616	n	156	6E
13	K	113	4B	7617	o	157	6F
14	L	114	4C	7620	p	160	70
15	M	115	4D	7621	q	161	71
16	N	116	4E	7622	r	162	72
17	O	117	4F	7623	s	163	73
20	P	120	50	7624	t	164	74
21	Q	121	51	7625	u	165	75
22	R	122	52	7626	v	166	76
23	S	123	53	7627	w	167	77
24	T	124	54	7630	x	170	78
25	U	125	55	7631	y	171	79
26	V	126	56	7632	z	172	7A
27	W	127	57	7633	{	173	7B
30	X	130	58	7634		174	7C
31	Y	131	59	7635	}	175	7D
32	Z	132	5A	7636	~	176	7E
33	0	060	30	7637	DEL	177	7F
34	1	061	31	7640	NUL	000	00
35	2	062	32	7641	SOH	001	01
36	3	063	33	7642	STX	002	02
37	4	064	34	7643	ETX	003	03
40	5	065	35	7644	EOT	004	04
41	6	066	36	7645	ENO	005	05
42	7	067	37	7646	ACK	006	06
43	8	070	38	7647	BEL	007	07
44	9	071	39	7650	BS	010	08
45	+	053	2B	7651	HT	011	09
46	-	055	2D	7652	LF	012	0A
47	*	052	2A	7653	VT	013	0B
50	/	057	2F	7654	FF	014	0C
51	(050	28	7655	CR	015	0D
52)	051	29	7656	SO	016	0E
53	\$	044	24	7657	SI	017	0F
54	=	075	3D	7660	DLE	020	10
55	(space)	040	20	7661	DC1	021	11
56	,	054	2C	7662	DC2	022	12
57	.	056	2E	7663	DC3	023	13
60	#	043	23	7664	DC4	024	14
61	[133	5B	7665	NAK	025	15
62]	135	5D	7666	SYN	026	16
63 ^{††}	%	045	25	7667	ETB	027	17
64	"	042	22	7670	CAN	030	18
65	<u>†††</u>	137	5F	7671	EM	031	19
66	!	041	21	7672	SUB	032	1A
67	&	046	26	7673	ESC	033	1B
70	'	047	27	7674	FS	034	1C
71	?	077	3F	7675	GS	035	1D
72	<	074	3C	7676	RS	036	1E
73	>	076	3E	7677	US	037	1F
74	@	100	40	7400	null	---	--
75	\	134	5C	7401	@	100	40
76	^	136	5E	7402	^	136	5E
77	;	073	3B	7403	null	---	--
7600	null	---	--	7404	:	072	3A
7601	a	141	61	7405	null	---	--
7602	b	142	62	7406	null	---	--
7603	c	143	63	7407	,	140	60

[†]In the 63-character set, this display code represents a null character. Also, use of the colon in program and data files may cause problems. This is particularly true when it is used in PRINT and FORMAT statements.

^{††}In the 63-character set, this display code represents a colon (:), 7-bit ASCII code 072, 7-bit hexadecimal code 3A.

^{†††}On TTY models having no underline, the backarrow (←) takes its place.

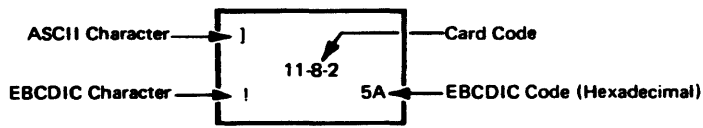
TABLE A-3. CODES (8-BIT) FOR NOS/BE

Bits	Hex Digits	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
b4 b3 b2 b1	Hex Digits	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL 12-0-9-8-1 NUL 00	DLE 12-11-9-8-1 DLE 10	SP no-punch SP 40	0 0 F0	@ 8-4 @ 7C	P 11-7 P D7	\ 8-1 \ 79	p 12-11-7 p 97
0 0 0 1	1	SOH 12-9-1 SOH 01	DC1 11-9-1 DC1 11	 12-8-7 4F	1 1 F1	A 12-1 A C1	Q 11-8 Q D8	a 12-0-1 a 81	q 12-11-8 q 98
0 0 1 0	2	STX 12-9-2 STX 02	DC2 11-9-2 DC2 12	" 8-7 " 7F	2 2 F2	B 12-2 B C2	R 11-9 R D9	b 12-0-2 b 82	r 12-11-9 r 99
0 0 1 1	3	ETX 12-9-3 ETX 03	DC3 11-9-3 TM 13	# 8-3 # 7B	3 3 F3	C 12-3 C C3	S 0-2 S E2	c 12-0-3 c 83	s 11-0-2 s A2
0 1 0 0	4	EOT 9-7 EOT 37	DC4 9-8-4 DC4 3C	\$ 11-8-3 \$ 5B	4 4 F4	D 12-4 D C4	T 0-3 T E3	d 12-0-4 d 84	t 11-0-3 t A3
0 1 0 1	5	ENQ 0-9-8-5 ENQ 2D	NAK 9-8-5 NAK 3D	% 0-8-4 % 6C	5 5 F5	E 12-5 E C5	U 0-4 U E4	e 12-0-5 e 85	u 11-0-4 u A4
0 1 1 0	6	ACK 0-9-8-6 ACK 2E	SYN 9-2 SYN 32	& 12 & 50	6 6 F6	F 12-6 F C6	V 0-5 V E5	f 12-0-6 f 86	v 11-0-5 v A5
0 1 1 1	7	BEL 0-9-8-7 BEL 2F	ETB 0-9-6 ETB 26	' 8-5 ' 7D	7 7 F7	G 12-7 G C7	W 0-6 W E6	g 12-0-7 g 87	w 11-0-6 w A6
1 0 0 0	8	BS 11-9-6 BS 16	CAN 11-9-8 CAN 18	(12-8-5 (4D	8 8 F8	H 12-8 H C8	X 0-7 X E7	h 12-0-8 h 88	x 11-0-7 x A7
1 0 0 1	9	HT 12-9-5 HT 05	EM 11-9-8-1 EM 19) 11-8-5) 5D	9 9 F9	I 12-9 I C9	Y 0-8 Y E8	i 12-0-9 i 89	y 11-0-8 y A8
1 0 1 0	10 (A)	LF 0-9-5 LF 25	SUB 9-8-7 SUB 3F	* 11-8-4 * 5C	: 8-2 : 7A	J 11-1 J D1	Z 0-9 Z E9	j 12-11-1 j 91	z 11-0-9 z A9
1 0 1 1	11 (B)	VT 12-9-8-3 VT 0B	ESC 0-9-7 ESC 27	+ 12-8-6 + 4E	: 11-8-6 : 5E	K 11-2 K D2	[12-8-2 [4A	k 12-11-2 k 92	{ 12-0 { C0
1 1 0 0	12 (C)	FF 12-9-8-4 FF 0C	FS 11-9-8-4 IFS 1C	- 0-8-3 - 6B	< 12-8-4 < 4C	L 11-3 L D3	\ 0-8-2 \ E0	l 12-11-3 l 93	 12-11 6A
1 1 0 1	13 (D)	CR 12-9-8-5 CR 0D	GS 11-9-8-5 IGS 1D	- 11 - 60	= 8-6 = 7E	M 11-4 M D4] 11-8-2] 5A	m 12-11-4 m 94	} 11-0 } D0
1 1 1 0	14 (E)	SO 12-9-8-6 SO 0E	RS 11-9-8-6 IRS 1E	- 12-8-3 - 4B	> 0-8-6 > 6E	N 11-5 N D5	^ 11-8-7 ^ 5F	n 12-11-5 n 95	~ 11-0-1 ~ A1
1 1 1 1	15 (F)	SI 12-9-8-7 SI 0F	US 11-9-8-7 IUS 1F	/ 0-1 / 61	? 0-8-7 ? 6F	O 11-6 O D6	_ 0-8-5 _ 6D	o 12-11-6 o 96	DEL 12-9-7 DEL 07

64-character ASCII

95-character ASCII (does not include DEL)

128-character ASCII



Diagnostic messages are issued by FORTRAN 5 during both compilation and execution to inform the user of errors in the source program, input data, or intermediate results. This appendix explains the content and format of the FORTRAN 5 diagnostic messages.

COMPILE-TIME DIAGNOSTICS

When an error is detected during compilation of the source program, a diagnostic message is issued immediately after the erroneous source line. The format of the diagnostics is:

severity * message

The severity indicator tells the consequences the error will have on further processing of the program. One of the following severity indicators will accompany each error message:

FATAL

The program will not be executed.

WARNING

The error is severe, but the program will be executed. Although syntax is incorrect, the probable meaning of the source code is presumed.

TRIVIAL

A minor syntax error or omission was detected, or correct syntax was used but semantics were irregular.

ANSI

Usage does not conform to ANSI X3.9 - 1978 (FORTRAN 77) specification. Listed only if the ANSI list option is specified on the FTN5 control statement.

MDEP

The line contains a use of a machine-dependent language feature. Listed only if the MD option is specified on the FTN5 control statement.

The compile-time diagnostics issued by FORTRAN 5 are summarized in alphabetical order in table B-1. Ellipses, denoted by ..., are replaced by items from the relevant source statement.

SPECIAL COMPILATION DIAGNOSTICS

When a compilation is aborted or prematurely terminated for internal reasons, one or more of the messages shown in table B-2 are issued. This table also includes messages that appear only in the dayfile that are not caused by internal error.

CONTROL STATEMENT DIAGNOSTICS

When errors are detected in the control statement portion of the program the message CONTROL STATEMENT ERRORS-- followed by one or more of the messages shown in table B-3 is issued. These messages only appear in the dayfile.

COMPILER OUTPUT LISTING MESSAGES

Compiler output listing messages are printed in the source listing. They may appear before, during, or after the reference map and object code listings, depending on the error condition. The message format is different than that of the standard error summary; each message is usually left-justified on the output page, and may be preceded by several blank lines, or may be printed at the top of a page.

The compiler output listing messages are given in table B-4.

EXECUTION DIAGNOSTICS

Execution diagnostics are issued when an error occurs while a user program is running. The diagnostics are printed on the source listing in one of the following formats:

ERROR NUMBER x DETECTED BY routine
AT ADDRESS y

or

ERROR NUMBER x DETECTED BY routine
CALLED FROM routine AT ADDRESS z

or

ERROR NUMBER x DETECTED BY routine
CALLED FROM routine AT LINE d

where y and z are relative octal addresses, x is a decimal error number, and d is a decimal line number corresponding to a line number printed in the source listing.

Table B-5 summarizes the execution diagnostics by error number. In table B-5, the letters under Class mean:

F = Fatal

I = Informative, nonfatal

D = Debug (diagnostic can be issued only when in debug mode)

T = Trace (diagnostic can be issued only when in trace mode). Trace mode is active when DB=TB is specified on the FORTRAN 5 compilation control Card.

A = Always (diagnostic can always be issued)

When invalid data is entered through connected input, it can be reentered for formatted, list directed, or namelist read operations. Data that would otherwise produce error 78 (illegal data in field) or 79 (data overflow) causes this message to be displayed:

dd...dd<-ERROR IN COL. nn, RETYPE RECORD FROM THIS FIELD

where

dd..dd Is the data field right justified and blank filled, where the rightmost character is the one at which the error condition was detected.

nn Is the column number, in decimal, of the character that caused the message.

Data can then be reentered, starting at the field described and continuing through the end of the iolist. For formatted input, a field is defined by the last edit descriptor active when the error was detected; reentered data will be processed from the beginning of that descriptor. For list directed and namelist input, a field is defined as the last iolist item active when the error was detected in the constant or repetition count; the reentered data will be processed from the beginning of the iolist item.

TABLE B-1. COMPILE-TIME DIAGNOSTICS

	Message	Significance	Action
ANSI IS DEFINED TO BE INTRINSIC	The FORTRAN 5 defined intrinsic function is not supported in ANSI FORTRAN.	Supply the function for portability.
ANSI IS NON-ANSI EDIT DESCRIPTOR	Nonstandard format specification.	Replace format specification.
ANSI	7 CHARACTER SYMBOL IS NON-ANSI	ANSI allows only 6 characters.	Shorten symbol to 6 characters or less.
ANSI	CHARACTER ARRAY REQUIRED FOR FORMAT SPECIFIER	Format must be contained in character array.	Use an array of character type.
ANSI	COMMON BLOCK NAME CANNOT BE	Common block name used as another symbol name in a nonANSI manner (for example, as an entry point name or as an intrinsic function name.)	Change the common block name or, when possible, the symbol name.
ANSI	COMMON CAN BE PRESET IN BLOCK DATA ONLY	ANSI allows COMMON to be preset in block data only.	Remove presetting of COMMON.
ANSI	COMPUTED GO TO INDEX MUST BE INTEGER	Index is of incorrect type.	Change GO TO index or declare it to be integer.
ANSI	DOUBLE PRECISION AND COMPLEX OPERANDS ARE MIXED	Cannot mix DOUBLE PRECISION and COMPLEX operands.	Apply REAL function to DOUBLE PRECISION operand.
ANSI	FILE DECLARATION LIST NON-ANSI	ANSI does not permit file declaration in the PROGRAM statement.	Remove file list from PROGRAM statement.
ANSI	FUNCTION REFERENCE IN CONSTANT EXPRESSION	ANSI does not allow function reference in constant expression.	Remove function reference.
ANSI	HOLLERITH CONSTANT NON-ANSI	ANSI uses character data type.	Switch usage to character.
ANSI	I/O KEYWORD BUFL IS NON-ANSI	ANSI does not permit I/O keyword BUFL.	Remove I/O keyword BUFL.
ANSI	LIST DIRECTED OUTPUT CANNOT END WITH COMMA	ANSI does not permit comma at end of list directed output.	Remove the trailing comma.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
ANSI	MASK EXPRESSION NON-ANSI	ANSI does not permit mask expressions.	Remove mask expression.
ANSI	MULTIPLE ASSIGNMENT IS NON-ANSI	ANSI permits only one assignment per statement.	Break assignment statement into two or more statements.
ANSI	NAMelist I/O IS NON-ANSI	ANSI does not permit NAMelist I/O.	Remove NAMelist I/O.
ANSI	OBJECT OF IF IS ILLEGAL DO TERMINATOR	A logical IF, used as the last statement in a DO loop, contains a nonstandard statement.	Change object of IF. Make the last statement in the loop a CONTINUE statement.
ANSI	OCT/HEX DATA TYPE NOT DEFINED IN ANSI	ANSI does not permit octal or hexadecimal data type.	Write number as decimal.
ANSI	PAREN REPEAT LIST IS NOT PERMITTED	Repeated item list is not provided in standard FORTRAN.	Remove paren repeat list.
ANSI	RETURN IN MAIN PROGRAM -- ACTS AS END	RETURN is considered the END statement in main program.	Change RETURN to END or STOP.
ANSI	SEQUENCE MODE IS NON-ANSI	ANSI does not recognize SEQ format.	Remove sequence numbers and compile without SEQ parameter.
ANSI	STATEMENT FUNCTION ACTUAL ARGUMENT MUST AGREE IN TYPE WITH DUMMY ARGUMENT	ANSI requires that dummy and actual arguments to statement functions agree in type.	Change type declaration of dummy or actual argument.
ANSI	STATEMENT FUNCTION DUMMY ARGUMENT ... CANNOT BE AN ARRAY	Declaration of dummy argument is invalid.	Change declaration of dummy argument or name of dummy argument.
ANSI	STATEMENT IS NOT DEFINED IN ANSI	ANSI does not recognize statement.	Correct statement.
ANSI	SUBSCRIPT OF IS NOT TYPE INTEGER	ANSI requires integer subscripts.	Assign subscript expression to an integer variable and use the variable.
ANSI	SUBSTRING EXPRESSION NOT INTEGER	ANSI permits only integer substring expressions.	Check substring expression.
ANSI	SYMBOLIC CONSTANT IN COMPLEX CONSTANT NOT ANSI	Symbolic constant in complex constant is not allowed by ANSI.	Replace symbolic constant with constant.
ANSI	TRANSFER INTO RANGE OF DO	Cannot transfer into range of DO.	Rewrite loops to be closed.
FATAL EDIT DESCRIPTOR REQUIRES COUNT	Program will not execute without count.	Supply a count for the edit descriptor.
FATAL EXPRESSION NOT CONSTANT, OR NOT EVALUATABLE	Expression, which must be a constant, will not reduce.	Rewrite statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL — ILLEGAL TRANSFER TO INSIDE A CLOSED DO LOOP OR IF BLOCK	To branch inside a DO loop, a branch must previously have been made out of the loop. Branching into an IF block is illegal.	Revise program flow to remove invalid branch.
FATAL I/O CONTROL ALREADY SPECIFIED FOR THIS STATEMENT	Duplication of I/O specifier is invalid.	Remove duplicate I/O specifier.
FATAL NOT I/O CONTROL KEYWORD	I/O control keyword not recognized.	Likely to be a misspelled keyword. Correct it.
FATAL NOT LEGAL I/O CONTROL KEYWORD FOR THIS STATEMENT	Valid I/O keyword but not for this statement.	Remove I/O control keyword.
FATAL BLOCK IF(S) NOT TERMINATED	Missing ENDIF statement.	Insert ENDIF statement.
FATAL C\$ IF(S) NOT TERMINATED	Missing C\$ ENDIF statement.	Insert C\$ ENDIF statement.
FATAL CANNOT HAVE ASSUMED CHARACTER LENGTH	Only symbolic constants and dummy arguments may have (*) length.	Remove (*) length declaration.
FATAL CAUSES CHARACTER DECLARATION CONFLICT IN EQUIVALENCE GROUP	Character declaration conflict encountered in EQUIVALENCE statement.	Check declarations of equivalenced character variables.
FATAL ILLEGAL EXTENSION OF COMMON BLOCK ORIGIN	The EQUIVALENCE statement has extended the common block origin backward.	Check all EQUIVALENCE statements containing the specified variable.
FATAL ILLEGAL FIRST ELEMENT OF EXPRESSION	First element of expression found to be invalid.	Correct first element of expression.
FATAL IN INPUT LIST IS ILLEGAL	Constants and expressions cannot appear in input lists.	Remove constant or expression.
FATAL IS IN BLANK COMMON -- DATA IGNORED	Blank common variables must not be initialized.	Remove blank common variables from DATA statement.
FATAL IS NOT DEFINED AS INTRINSIC	Name is not the name of a builtin intrinsic function.	Remove name or correct spelling.
FATAL IS UNKNOWN C\$ PARAMETER FOR	C\$ parameter not recognized for this expression.	Correct C\$ parameter or expression.
FATAL MUST BE DO CONTROL VARIABLE	Expression must be a DO control variable.	Make expression a DO control variable.
FATAL MUST BE A DUMMY-ARG	Expression must be a dummy-arg.	Make expression a dummy-arg.
FATAL MUST BE INTEGER CONSTANT EXPRESSION	Expression must be an integer constant.	Make expression an integer constant.
FATAL OPERAND CANNOT BE CONVERTED TO TYPE	The operand cannot be converted to the type attempted.	Check operand and type.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL PREVIOUSLY USED IN EXECUTABLE OR CONFLICTING DECLARATIVE	Dummy argument on ENTRY statement had previous use that prohibits use as a dummy argument.	Correct the previous usage or change the name of the dummy argument.
FATAL REDEFINES A DO CONTROL INDEX	Variable redefines a current DO index.	Change variable usage. Check equivalence declarations.
FATAL SUBSCRIPT OUTSIDE OF ARRAY BOUNDS	Subscript must be inside of array bounds.	Check subscript and dimension statement.
FATAL TOO HIGH -- SHORT LOOPS SELECTED	Trip count, $ m1+m3 $ or $ m2+m3 > 2^{17}-1$.	Modify loop or select DO=LONG.
FATAL	3 BRANCH IF HAS EXPRESSION	3 branch IF expression must be integer, real, or boolean.	Change type of expression to integer, real, or boolean.
FATAL	3 BRANCH IF MISSING LABEL	Label required for 3 branch IF.	Supply a label.
FATAL	ADJUSTABLE BOUND MUST BE DUMMY-ARG OR IN COMMON	Variable used as a dimension bound must be a dummy-arg or in common.	Add variable to dummy-arg list or to common block.
FATAL	ADJUSTABLE DIMENSION BOUND IS NOT INTEGER	Adjustable dimension bound must be integer.	Declare adjustable dimension bound to be integer.
FATAL	ALTERNATE RETURN IS ILLEGAL IN A FUNCTION	Legal only in a subroutine.	Remove alternate return.
FATAL	ARGUMENT COUNT ON EXCEEDS 500	Too many arguments.	Reduce number of arguments.
FATAL	ARGUMENT COUNT ON MUST BE MORE THAN ONE	Not enough arguments.	Increase number of arguments.
FATAL	ARGUMENT COUNT ON INTRINSIC IS WRONG	Wrong number of arguments supplied for the intrinsic function.	Check syntax of the intrinsic function.
FATAL	ARGUMENT MODE ILLEGAL FOR GENERIC FUNCTION	Improper argument type.	Check definition of function to determine correct argument type.
FATAL	ARGUMENT MODE MUST AGREE WITH TYPE DEFINED FOR LIBRARY FUNCTION	Improper argument type.	Check definition of function to determine correct argument type.
FATAL	ARRAY DIMENSION -- DIMENSION BOUND EXPRESSION CONTAINS ILLEGAL OPERATION	Illegal expression in DIMENSION declaration.	Correct expression.
FATAL	ARRAY DIMENSION -- DIMENSION BOUND EXPRESSION CONTAINS NON-VARIABLE	Illegal expression in DIMENSION declaration.	Correct invalid variables in expression.
FATAL	ARRAY DIMENSION -- DIMENSION BOUND EXPRESSION CONTAINS ARRAY REFERENCE	Array references in expressions are illegal when the expression appears in a DIMENSION statement.	Remove all array references from expression.
FATAL	ARRAY DIMENSION -- LOWER BOUND EXCEEDS UPPER BOUND	Lower bound must be less than or equal to upper bound.	Correct dimension boundaries.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	ARRAY DIMENSION -- EXCEEDS 2**23-1	Dimension value too large.	Reduce dimension size.
FATAL	ARRAY DIMENSION BOUND NOT INTEGER	Dimension bounds must be integer.	Declare dimension bound to be integer.
FATAL	ARRAY EXCEEDS 7 DIMENSIONS	Too many dimensions.	Reduce number of dimensions.
FATAL	ARRAY IN COMMON CANNOT HAVE ADJUSTABLE DIMENSION	Arrays in common must have explicit bounds.	Correct dimension descriptor.
FATAL	ARRAY IN PROGRAM CANNOT HAVE ADJUSTABLE DIMENSION	Arrays in PROGRAM program unit must have explicit bounds.	Correct dimension descriptor.
FATAL	ARRAY MISSING SUBSCRIPT	Subscript required to reference an array element.	Supply subscript.
FATAL	ARRAY SIZE EXCEEDS 2**23-1	Array too large.	Reduce size of array.
FATAL	ARRAY SUBSCRIPT COUNT DOES NOT MATCH DIMENSION COUNT	Wrong number of subscripts supplied. The number of subscripts in an array reference must equal the number specified in the DIMENSION statement.	Check the number of subscripts on the DIMENSION statement.
FATAL	ARRAY DECLARATION FOR MISSING RIGHT PAREN	Right parenthesis missing.	Supply right parenthesis.
FATAL	ASSUMED CHARACTER LENGTH ILLEGAL FOR IMPLICIT	Length must be declared in the IMPLICIT statement.	Declare character length.
FATAL	ASSUMED SIZE ARRAY NOT ALLOWED IN I/O LIST	Assumed size array must have a subscript when appearing in an I/O list.	Specify a subscript.
FATAL	ASSUMED SIZE ARRAY NOT PERMITTED IN NAMELIST	Assumed size array must be subscripted when it appears in a namelist.	Specify a subscript.
FATAL	ASSUMED SIZE CAN ONLY BE ON LAST UPPER BOUND	Assumed size is not last upper bound.	Declare size when not last upper bound.
FATAL	ASSUMED SIZE OR ADJUSTABLE ARRAY MUST BE DUMMY-ARG	Assumed size or adjustable array is not dummy-arg.	Make assumed size or adjustable array dummy-arg.
FATAL	BUFFER DIRECTION SPECIFIER MUST BE IN OR OUT	BUFFER statement incorrect; correct form is BUFFER IN or BUFFER OUT.	Make BUFFER statement: BUFFERIN or BUFFEROUT.
FATAL	BUFFER I/O ADDRESS CANNOT BE CHARACTER	Buffer I/O address must not be character.	Change Buffer I/O address.
FATAL	BUFFER I/O ADDRESS CANNOT BE	Buffer I/O address is not recognized.	Correct Buffer I/O address.
FATAL	BUFFER I/O LWA MUST BE GREATER THAN OR EQUAL TO FWA	Last-word-address must be greater than or equal to first-word-address.	Correct word-address boundaries.
FATAL	BUFFER I/O PARITY SPECIFIER MUST BE INTEGER CONSTANT OR VARIABLE	Buffer I/O parity specifier not recognized.	Correct Buffer I/O parity specifier.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	BUFFER I/O PARITY INDICATOR VALUE MUST BE ZERO OR 1	Buffer I/O parity indicator not zero or 1.	Make Buffer I/O parity indicator zero or 1.
FATAL	BUFFER LENGTH FOR FILE EXCEEDS 360000B -- DEFINITION IGNORED	Buffer length too long.	Reduce buffer length.
FATAL	C\$ IF EXPRESSION MUST BE LOGICAL	C\$ IF expression is not type logical.	Make expression type logical.
FATAL	C\$ LABEL DIFFERENT FROM C\$ IF LABEL	Label on C\$ IF does not match C\$ ENDIF or C\$ ELSE label.	Make labels identical.
FATAL	CALL STATEMENT MISSING ROUTINE NAME	The correct form is CALL routine-name (parameter list).	Insert routine name between CALL keyword and parameter list.
FATAL	CHARACTER AND OTHER TYPE OPERANDS MAY NOT BE MIXED	Character operands cannot be mixed with non-character operands.	Correct operands.
FATAL	CHARACTER DECLARATION CONFLICT EXISTS IN COMMON BLOCK	Common block contains character and non-character entities.	Make all common block members either type character or type non-character.
FATAL	CHARACTER LENGTH GREATER THAN 2**15-1	Character variable too long.	Shorten character variable.
FATAL	CHAR LENGTH NOT POSITIVE CONSTANT, (POSITIVE CONSTANT EXPRESSION) OR (*)	The length on a CHARACTER or type declaration was negative or zero.	Correct the length specification.
FATAL	CHARACTER LENGTH ZERO ILLEGAL	Length must be at least 1.	Correct character length value.
FATAL	CHARACTER LENGTHS OF ENTRY AND FUNCTION CANNOT DISAGREE	Character lengths of entry and function disagree.	Correct disagreement.
FATAL	CHARACTER OPERAND USED WITH OPERATOR	Operation illegal for character variable.	Correct conflict.
FATAL	COMMA BEFORE AN I/O LIST IS ALLOWED ONLY ON SHORT FORM READ OR PRINT STATEMENT	The comma before the I/O list is not allowed here.	Remove comma.
FATAL	COMMA OR E.O.S. MUST FOLLOW LEVEL LIST NAME	Comma or end of statement expected; statement contains extraneous information.	Correct statement.
FATAL	COMMON BLOCK CANNOT BE DECLARED LEVEL 0	Wrong level declared for this block.	Declare correct level.
FATAL	COMMON BLOCK EXCEEDS MAX BLOCK LENGTH 131071	Common block too large.	Break common block into two or more common blocks.
FATAL	COMMON BLOCK EXCEEDS MAX LCM=G BLOCK LENGTH 1048568	Common block too large.	Break common block into two or more common blocks.
FATAL	COMMON ELEMENT MAY NOT APPEAR IN SAVE	Names of entities in a common block may not appear in the SAVE statement.	Correct the SAVE statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	CONCATENATION OF ASSUMED LENGTH VARIABLE NOT ALLOWED HERE	Assumed length variable cannot be concatenated in this circumstance.	Do not concatenate variable here.
FATAL	CONFLICT IN EQUIVALENCE SPECIFICATION FOR	Indicated EQUIVALENCE is inconsistent with previous EQUIVALENCE.	Check all EQUIVALENCE statements containing the specified variable.
FATAL	CONSTANT CANNOT BE CONVERTED	Constant contains syntax error.	Correct syntax error in constant.
FATAL	CONSTANT DIVIDE BY ZERO -- RESULTS SET TO INFINITE	Division by zero is an undefined operation.	Correct division error.
FATAL	DATA INTO IS ILLEGAL	DATA statement attempts to initialize something which cannot be initialized, such as a formal parameter.	Correct DATA statement.
FATAL	DATA VARIABLE LIST CONTAINS	DATA variable list contains a constant or an expression.	Correct DATA statement.
FATAL	DECIMAL POINT IS NOT SPECIFIED FOR THE EDIT DESCRIPTOR AT	Decimal point is invalid in this circumstance.	Remove decimal point.
FATAL	DECIMAL POINT REQUIRED IN EDIT DESCRIPTOR AT	Decimal point required.	Supply decimal point.
FATAL	DIMENSION ON IGNORED -- PRIOR DIMENSION RETAINED	A dimension was specified more than once; first declaration is used.	Eliminate second declaration.
FATAL	DIRECT ACCESS I/O CANNOT BE FREE FORMAT	FORMAT specification needed.	Replace * with format specification.
FATAL	DIRECT ACCESS I/O CANNOT BE NAMELIST	FORMAT specification needed.	Replace namelist name with format specification.
FATAL	DIRECT ACCESS I/O CANNOT SPECIFY END	END option is illegal.	Remove END= specifier from I/O statement.
FATAL	DO PARAMETER CANNOT BE	Type of the DO parameter is invalid.	Change the type of the parameter.
FATAL	DO-IMPLIED LOOPS IN DATA MUST BE INTEGER	DO-implied loops are required to be integer.	Make DO-implied loops integer.
FATAL	DO INDEX MUST BE SIMPLE VARIABLE	DO index is required to be a simple variable.	Make DO index a simple variable.
FATAL	DO INDEX CANNOT BE	Type of DO index is invalid.	Change the type of the DO index.
FATAL	DO LOOP CONTAINS UNCLOSED IF BLOCK	Entire IF block must be within the range of the DO loop.	Make IF block within range of DO loop.
FATAL	DO LOOP MUST TERMINATE WITHIN IF BLOCK	Entire DO loop must be within the range of the IF block.	Make DO loop within range of IF block.
FATAL	DO LOOP NOT TERMINATED BEFORE END OF PROGRAM	DO loop terminator missing.	Add DO terminator statement number where appropriate.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	DO LOOP PREVIOUSLY DEFINED -- ILLEGAL NESTING	The label was previously used.	Choose a new statement number for the DO.
FATAL	DO LOOP INCREMENT MAY NOT BE ZERO	DO loop increment is required to be nonzero.	Provide nonzero increment.
FATAL	D OR M FIELD NOT SPECIFIED FOR EDIT DESCRIPTOR AT	Edit descriptor of form Fw. was encountered.	Add the m or d field to the edit descriptor.
FATAL	DUMMY-ARG FUNCTION CANNOT HAVE ASSUMED CHARACTER LENGTH	Dummy-arg function has assumed character length.	Specify length of character dummy-arg.
FATAL	DUMMY ARGUMENT CAN OCCUR ONLY ONCE IN DEFINITION	Dummy argument previously defined in current statement function.	Remove excess dummy argument.
FATAL	DUMMY ARGUMENT CANNOT BE EQUIVALENCED	Dummy argument must not appear in EQUIVALENCE statement.	Remove dummy argument from EQUIVALENCE statement.
FATAL	DUMMY ARGUMENT MAY NOT APPEAR IN SAVE	Dummy argument must not appear in SAVE statement.	Remove dummy argument from SAVE statement.
FATAL	DUMMY ARGUMENT MUST BEGIN WITH LETTER OR STAR	Dummy argument must begin with a letter or star.	Correct dummy argument.
FATAL	E.O.S. BEFORE END OF HOLLERITH COUNT	Premature end of statement encountered.	Check for incorrect hollerith count.
FATAL	EDIT DESCRIPTOR MISSING AT	Error in FORMAT statement.	Supply edit descriptor.
FATAL	ELSEIF EXPRESSION MUST BE LOGICAL	ELSEIF expression is not type logical.	Make ELSEIF expression type logical.
FATAL	ELSEIF REQUIRES THEN	THEN is missing from ELSEIF construct.	Add THEN where appropriate.
FATAL	EMPTY COMMON BLOCK	Common block contains no elements.	Remove COMMON statement or add variable list.
FATAL	END LINE ABSENT	END statement must be last statement in source deck.	Add END statement.
FATAL	END OR ERR REQUIRES STATEMENT LABEL	The END= or ERR= in a READ statement must be followed by the label number of an executable statement.	Provide statement label.
FATAL	ENTRY INSIDE DO LOOP OR IF BLOCK IS ILLEGAL	Illegal entry into range of DO loop or IF block.	Remove ENTRY or rewrite loop or block.
FATAL	EQUAL SIGN MUST BE FOLLOWED BY NAME, NUMBER OR SLASH	Equal sign required to be followed by a name, number, or a slash.	Correct expression after equal sign.
FATAL	EQUIVALENCED ARRAY HAS SUBSCRIPT LESS THAN DIMENSION LOWER BOUND	Subscript must be greater than or equal to lower bound specified in the DIMENSION statement.	Change subscript or dimension.
FATAL	EQUIVALENCED ARRAY HAS SUBSCRIPT WHICH EXCEEDS DIMENSION BOUND	Subscript must be less than or equal to upper bound specified in DIMENSION.	Change subscript or DIMENSION statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	EXCESS LEFT PAREN IN I/O LIST	Too many left parens.	Remove excess paren(s).
FATAL	EXCESS LEFT PAREN IN I/O LIST ITEM SUBSCRIPT	Too many left parens.	Remove excess paren(s).
FATAL	EXCESS RIGHT PAREN IN I/O LIST	Too many right parens.	Remove excess paren(s).
FATAL	EXCESS SUBSCRIPTS ON EQUIVALENCE VARIABLE	EQUIVALENCE variable has more subscripts than de- clared in DIMENSION.	Change subscripts or DIMENSION statement.
FATAL	EXECUTABLE STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM	Illegal executable state- ments in block data subprogram.	Remove executable statements.
FATAL	EXPECTED C\$ DIRECTIVE LABEL -- FOUND	C\$ directive label expected.	Check C\$ directive keyword specification.
FATAL	EXPECTED C\$ PARAMETER -- FOUND	C\$ parameter expected.	Check C\$ directive keyword specification.
FATAL	EXPECTED COMMA -- FOUND	Comma expected.	Check syntax of state- ment.
FATAL	EXPECTED COMMA AFTER COUNT -- FOUND	Comma after count expected.	Check syntax of state- ment.
FATAL	EXPECTED COMMA AFTER FORMAT SPECIFIER -- FOUND	Comma after format specifier expected.	Check syntax of state- ment.
FATAL	EXPECTED COMMA OR RIGHT PAREN -- FOUND	Comma or right paren expected.	Check syntax of state- ment.
FATAL	EXPECTED COMMA OR SLASH FOUND	Comma or slash expected.	Check syntax of state- ment.
FATAL	EXPECTED DO CONTROL INDEX -- FOUND	Syntax error in DO statement.	Check syntax of DO statement.
FATAL	EXPECTED E.O.S. -- FOUND	Extraneous information follows a legal statement.	Remove extra informa- tion.
FATAL	EXPECTED E.O.S. -- FOUND AND IGNORED	End of statement expected.	Check syntax of state- ment.
FATAL	EXPECTED EQUAL SIGN -- FOUND	Equal sign expected.	Check syntax of state- ment.
FATAL	EXPECTED FORMAT SPECIFIER -- FOUND	Format specifier expected.	Check format statement.
FATAL	EXPECTED FILE NAME, FOUND	File name expected.	Correct statement.
FATAL	EXPECTED INTRINSIC FUNCTION NAME -- FOUND	Intrinsic function name expected.	Check intrinsic state- ment.
FATAL	EXPECTED LEFT PAREN -- FOUND	Left parenthesis expected.	Check syntax of state- ment.
FATAL	EXPECTED LEFT PAREN BEFORE COUNT -- FOUND	Left parenthesis before count expected.	Check syntax of state- ment.
FATAL	EXPECTED LEFT PAREN FOR AN ARGUMENT LIST FOUND	Left parenthesis for an argument list expected.	Check syntax of state- ment.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	EXPECTED LEFT PAREN OR PERIOD -- FOUND	Left parenthesis or period expected.	Check syntax of state- ment.
FATAL	EXPECTED NAME -- FOUND	Name expected.	Correct statement.
FATAL	EXPECTED RANGE INDICATOR -- FOUND	Range indicator expected.	Correct statement.
FATAL	EXPECTED RIGHT PAREN -- FOUND	Right parenthesis expected.	Check syntax of state- ment.
FATAL	EXPECTED RIGHT PAREN AFTER STRING ADDRESS -- FOUND	Right parenthesis after string address expected.	Check syntax of state- ment.
FATAL	EXPECTED RIGHT PAREN OR COMMA -- FOUND	Right parenthesis or comma expected.	Check syntax of state- ment.
FATAL	EXPECTED SLASH -- FOUND	Slash expected.	Check syntax of state- ment.
FATAL	EXPECTED SYMBOL -- FOUND	Symbol expected; scan of statement stopped.	Check syntax of state- ment.
FATAL	EXPECTED VARIABLE OR COMMON BLOCK NAME -- FOUND	Common block name, enclosed in slashes, must follow COMMON keyword for named common blocks. A variable list must follow COMMON key- word for blank common.	Correct statement.
FATAL	EXPONENT FIELD ON EDIT DESCRIPTOR AT IS ZERO OR NOT SPECIFIED	Exponent field is invalid.	Correct exponent field.
FATAL	EXPRESSION TOO COMPLICATED -- SCAN STOPPED AT	Expression too complicated; scan stopped.	Simplify expression using two or more statements.
FATAL	EXTERNAL UNIT SPECIFIER NOT INTEGER EXPRESSION	External unit specifier must be integer expression.	Make external unit specifier integer ex- pression.
FATAL	EXTRA CHARACTERS AFTER UNIT SPECIFIER IGNORED	Extraneous information follows a legal unit specifier.	Remove extra char- acters.
FATAL	EXTRANEOUS NUMERIC FIELD IN EDIT DESCRIPTOR AT	Invalid numeric field in edit descriptor.	Remove extra numeric field.
FATAL	FIELD WIDTH NOT SPECIFIED FOR EDIT DESCRIPTOR AT	Field width required.	Supply field width.
FATAL	FIELD WIDTH OF EDIT DESCRIPTOR AT	Field width is invalid.	Correct field width.
FATAL	FILE PREVIOUSLY DEFINED -- IGNORED	File already defined.	Self-explanatory.
FATAL	FILE NOT DEFINED -- DEFINITION IGNORED	File is not defined.	Define file.
FATAL	FORMAT SPECIFIER IS NAMELIST NAME	Format specifier cannot be NAMELIST name.	Correct format speci- fier.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	FORMAT LABEL PREVIOUSLY REFERENCED AS CONTROL STATEMENT LABEL	Label being referenced or defined as a format label was previously referenced as a control statement label.	Check all references to the label in question for consistent usage.
FATAL	FORMAT LABEL PREVIOUSLY REFERENCED AS DO STATEMENT LABEL	Label being referenced or defined as a format label was previously referenced as a DO statement label.	Check all references to the label in question for consistent usage.
FATAL	FORMAT MUST HAVE STATEMENT LABEL	Format is required to have statement label.	Provide a unique statement label for each FORMAT statement.
FATAL	FUNCTION ENTRY MAY NOT BE TYPE CHARACTER	Function entry must not be type character.	Make function entry noncharacter.
FATAL	FUNCTION ENTRY MUST BE TYPE CHARACTER	All entries in a character function must be of type character.	Make function entry type character.
FATAL	FUNCTION NAME OR ENTRY OF TYPE WAS NOT ASSIGNED A VALUE	The function name or entry must be assigned a value within the function.	Assign a value to the function name or entry within the function.
FATAL	FUNCTION REQUIRES EXPLICIT NULL ARGUMENT LIST	A null argument list is a left parenthesis followed immediately by a right parenthesis.	Provide null argument list after the function name in the function reference.
FATAL	GROUP NAME PREVIOUSLY DEFINED	The group name appears twice in the same NAME-LIST statement or in a previous NAMELIST statement.	Check for duplicate name-list group names.
FATAL	HEADER CARD NOT FIRST STATEMENT -- IGNORED	PROGRAM, SUBROUTINE, BLOCK DATA, or FUNCTION must be the first statement of a program.	Correct first statement of program.
FATAL	I/O CONTROL KEYWORD MUST BE POSITIVE INTEGER EXPRESSION	I/O control keyword is required to be positive integer expression.	Make I/O keyword positive expression.
FATAL	I/O CONTROL KEYWORD PARAMETER CANNOT BE	I/O control keyword parameter is invalid.	Correct I/O control keyword parameter.
FATAL	I/O CONTROL KEYWORD PARAMETER MUST BE TYPE	I/O control keyword parameter is wrong type.	Correct I/O control keyword parameter type.
FATAL	ILL-FORMED COMPLEX CONSTANT	Complex constant invalid.	Correct complex constant.
FATAL	ILLEGAL BLOCK IF STRUCTURE	ELSEIF, ELSE, or ENDIF appears, but is not associated with a block IF.	Check IF block nesting.
FATAL	ILLEGAL BLOCK NAME IN COMMON STATEMENT	Block name in COMMON statement illegal.	Correct block name.
FATAL	ILLEGAL BUFFER LENGTH FOR FILE -- DEFINITION IGNORED	Buffer length invalid.	Redefine buffer length.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	ILLEGAL CHARACTER COUNT	Character count for ENCODE or DECODE must be integer constant or simple integer variable.	Correct character count.
FATAL	ILLEGAL CONSTANT FOLLOWING + OR -	+ or - is followed by an illegal constant.	Correct illegal constant.
FATAL	ILLEGAL EXPLICIT LEVEL DECLARATION FOR COMMON MEMBER NAME	Explicit level declaration for a common member name is illegal.	Correct explicit level declaration.
FATAL	ILLEGAL FORM OF EXPONENT	Exponent is invalid.	Correct form of exponent.
FATAL	ILLEGAL FORMAT SPECIFIER	Format specifier must be a legal statement label.	Correct format specifier.
FATAL	ILLEGAL IF BLOCK NESTING WITH DO LOOP	Range of the IF block must be within the range of the DO loop.	Make range of IF block within range of DO loop.
FATAL	ILLEGAL IF STATEMENT -- OBJECT MISSING	End of statement encountered before finding object of IF.	Correct the IF statement.
FATAL	ILLEGAL NESTING OF DO LOOPS	The range of an inner DO must be within the range of an outer DO.	Restructure DO loops.
FATAL	ILLEGAL OBJECT OF IF -- TROUBLE STARTED AT	Object of IF illegal.	Correct object of IF.
FATAL	ILLEGAL OBJECT OF LOGICAL IF	Improper statement type, used as true part of a logical IF. The object must be an executable statement. It cannot be a logical IF, DO, block IF, ELSEIF, ENDIF, ELSE, or END.	Correct object of logical IF.
FATAL	ILLEGAL RANGE --	Range is illegal.	Correct range.
FATAL	ILLEGAL RECORD LENGTH FOR FILE	Record length invalid.	Redefine record length of file.
FATAL	ILLEGAL REFERENCE TO LABEL	The label specifies a non-executable statement.	Correct reference to label.
FATAL	DEFINED ON NON-EXECUTABLE STATEMENT		
FATAL	ILLEGAL REFERENCE TO STATEMENT LABEL	The label referencing a FORMAT statement appears on an executable statement.	Correct reference to statement label.
FATAL	AS A FORMAT		
FATAL	ILLEGAL REPEAT CONSTANT	Error in DATA statement.	Correct DATA statement.
FATAL	ILLEGAL SEPARATOR FOLLOWING DATA CONSTANT	The legal separators are), /, or . .	Replace with legal separator.
FATAL	ILLEGAL TRANSFER INTO RANGE OF DO	The indicated statement branches into a DO loop.	Check transfer into DO loop range.
FATAL	ILLEGAL TRANSFER TO	FORMAT statements cannot be the objects of transfers.	Correct illegal transfer.
FATAL	FORMAT		
FATAL	ILLEGAL USE OF ASSIGNMENT OPERATOR	Equal sign used improperly.	Correct use of equal sign.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	ILLEGAL USE OF ENTRY	Entry name used improperly.	Correct use of entry name.
FATAL	ILLEGAL USE OF NAMELIST GROUP NAME	Use of NAMELIST group name is invalid.	Correct use of NAMELIST group name.
FATAL	ILLEGAL USE OF OPERATOR/OPERAND --	Use of operator/operand is invalid.	Correct use of operator/operand.
FATAL	ILLEGAL USE OF PARAMETER	Use of parameter is invalid.	Use valid parameter.
FATAL	IMPLICIT MUST BE FOLLOWED BY A TYPE INDICATOR	Type information omitted.	Provide a type keyword, such as INTEGER or REAL.
FATAL	IMPLICIT STATEMENT MUST OCCUR BEFORE DECLARATIVE STATEMENTS	IMPLICIT must be the first statement after the PROGRAM statement.	Move the IMPLICIT statement.
FATAL	IMPLIED LOOP NOT TERMINATED	Implied loop must be terminated.	Check statement for syntax errors.
FATAL	IMPLIED I/O UNIT SPECIFIER NOT ALLOWED FOR THIS STATEMENT	Unit specifier must be explicit.	Explicitly specify I/O unit specifier.
FATAL	INITIAL LEFT PAREN MISSING	The initial left parenthesis is missing.	Provide left parenthesis.
FATAL	INQUIRE CANNOT SPECIFY BOTH UNIT AND FILE	Either a file name or a unit specifier must be specified in an INQUIRE statement.	Specify either unit or file.
FATAL	INQUIRE MUST SPECIFY UNIT OR FILE	INQUIRE statement is required to specify a file name or a unit specifier.	Specify either unit or file.
FATAL	INTEGER 0, 1, 2 OR 3 MUST FOLLOW LEVEL	0, 1, 2 or 3 are required to follow LEVEL in a LEVEL statement.	Correct LEVEL statement.
FATAL	INTERNAL FILE I/O CANNOT BE FREE FORMAT	Internal files are not compatible with free format.	Change unit on format specifier.
FATAL	INTERNAL FILE I/O CANNOT BE NAMELIST	Interval file I/O must not be NAMELIST.	Check NAMELIST.
FATAL	INTERNAL FILE REQUIRES A FORMAT	The internal file must have a format.	Provide format for internal file.
FATAL	INTERNAL FILE WITHOUT FORMAT OR MISSING COMMA BEFORE I/O LIST	Internal file must have format or comma missing before I/O list.	Provide format for internal file or place comma before I/O list.
FATAL	INTERNAL UNIT SPECIFIER CANNOT BE	Illegal use of internal unit specifier.	Correct illegal use.
FATAL	INTERNAL UNIT SPECIFIER CANNOT BE ASSUMED SIZE ARRAY	Internal unit specifier must not be assumed size array.	Specify size array.
FATAL	INTERNAL UNIT SPECIFIER NOT ALLOWED FOR THIS STATEMENT	Internal unit specifier invalid in this context.	Check statement.
FATAL	INTRINSIC FUNCTION	INTRINSIC function is not allowed as actual argument.	Remove intrinsic function name from argument list.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	INTRINSIC LEN MUST NOT APPEAR IN PARAMETER CONSTANT EXPRESSION	LEN intrinsic appears in PARAMETER statement.	Rewrite statement.
FATAL	INVALID FORM INVOLVING THE USE OF A COMMA OR REDUNDANT PARENS	Parenthesized form with comma(s) in error. May be badly formed complex constant or I/O list with redundant parentheses.	Correct use of comma or remove redundant parentheses.
FATAL	INVALID STATEMENT LABEL	The statement label is invalid.	Correct statement label.
FATAL	LEFT SIDE OF EQUAL SIGN IS ILLEGAL	Left side of equal sign illegal.	Correct left side of equal sign.
FATAL	LENGTH OF CHARACTER FORMAT SPECIFIER MUST BE GREATER THAN 1	The length of the character format specifier must be greater than 1.	Correct length of character format specifier.
FATAL	LEVEL 3 NAME MAY NOT OCCUR IN THIS STATEMENT	Level 3 data cannot be used in expressions.	Correct use of level 3 data.
FATAL	LOCF ARGUMENT MUST NOT BE	LOCF argument must be a variable.	Make LOCF a variable.
FATAL	LOGICAL IF EXPRESSION MUST BE LOGICAL	Logical IF expression is required to be logical.	Make logical IF expression logical.
FATAL	LOGICAL IF MUST NOT BE OBJECT OF LOGICAL IF	Logical IF cannot be object of logical IF.	Correct object of logical IF.
FATAL	MAGNITUDE OF SUBSCRIPT OF EXCEEDS 2**23-1	Subscript too large or too small.	Correct subscript.
FATAL	MISSING COMMA AT	Comma is missing in statement.	Provide comma in proper place.
FATAL	MISSING LEFT PAREN AT	Left paren is missing in statement.	Provide left paren in proper place.
FATAL	MISSING NAME IN LEVEL LIST	Name missing in LEVEL list.	Insert missing name.
FATAL	MISSING SLASH ON GROUP NAME	Group name must be enclosed by slashes.	Provide slashes on group name.
FATAL	MISSING SUBSCRIPTS SET TO LOWER BOUND FOR EQUIVALENCE VARIABLE	EQUIVALENCE variable contains fewer subscripts than declared dimension.	Check declaration of the EQUIVALENCE variable.
FATAL	MORE THAN 7 SUBSCRIPTS	Too many subscripts.	Reduce number of subscripts.
FATAL	MULTIPLE DECIMAL POINT IN EDIT DESCRIPTOR AT	Too many decimal points.	Remove extra decimal points.
FATAL	MULTIPLE DEFINITION OF CURRENT FORMAT LABEL	Format label previously defined.	Check FORMAT statements for duplicate labels.
FATAL	MULTIPLE OCCURENCES OF DUMMY ARGUMENT	Dummy argument occurs more than once in dummy-arg list.	Remove multiple occurrences of dummy arguments.
FATAL	MULTIPLY DEFINED STATEMENT LABEL	The same statement label appears on more than one statement.	Change duplicate labels.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	NAME EXCEEDS 7 CHARACTERS -- TRUNCATED TO	Names must be unique within 7 characters.	Shorten name.
FATAL	NAME IN DATA CONSTANT LIST MUST BE PARAMETER	Name must be parameter.	Remove name that is not a parameter.
FATAL	NESTING OF REPEAT COUNT IN DATA CONSTANT LIST IS ILLEGAL	Nesting of repeat count in data constant list is not allowed.	Remove nesting of repeat count.
FATAL	NO DIMENSION FOUND FOR EQUIVALENCE VARIABLE	Dimension of equivalence variable missing.	Supply dimension of equivalence variable.
FATAL	NO PREVIOUS C\$ IF DIRECTIVE	C\$ ELSE or ENDIF must be preceded by a C\$ IF.	Provide C\$ IF directive.
FATAL	NON-DUMMY ARGUMENT CANNOT BE LEVELED	Leveled name must be a dummy-arg or in common.	Add name to argument list or to a common block.
FATAL	NON-NULL LABEL FIELD ON CONTINUATION LINE	Columns 1 through 5 are not on continuation line.	Remove extraneous label.
FATAL	OBJECT OF GO TO MISSING	The GO TO does not specify an existing statement label.	Provide statement label or change object of GO TO.
FATAL	OBJECT OF GO TO DID NOT APPEAR IN ASSIGN STATEMENT	Object of GO TO must appear in ASSIGN statement.	Put object of GO TO in ASSIGN statement.
FATAL	ONLY ONE C\$ ELSE ALLOWED IN C\$ IF GROUP	More than one C\$ ELSE in C\$ IF group.	Remove excess C\$ ELSE from C\$ IF group.
FATAL	ONLY 9 PAREN LEVELS ALLOWED	Too many parenthesis levels in FORMAT statement.	Reduce number of parenthesis levels.
FATAL	ONLY 19 CONTINUATION LINES ARE PERMITTED	Too many continuation lines.	Reduce number of continuation lines.
FATAL	ONLY 500 DUMMY ARGUMENTS ARE PERMITTED -- EXCESS IGNORED	Total number of unique dummy arguments in the FUNCTION or SUBROUTINE statement and in all associated ENTRY statements exceed the allowed number.	Reduce number of dummy arguments.
FATAL	ONLY 500 COMMON BLOCKS ARE PERMITTED	Too many common blocks.	Reduce number of common blocks.
FATAL	ONLY LIST DIRECTED OUTPUT STATEMENTS MAY END WITH A COMMA	Extraneous comma found.	Remove comma.
FATAL	OPERAND HAS MODE NOT ALLOWED IN THIS CONTEXT	Wrong mode for this situation.	Correct mode.
FATAL	OPERAND OF // OPERATOR MUST BE TYPE CHARACTER	Operand is required to be type character.	Declare operand to be of type character.
FATAL	OPERAND OF ** OPERATOR MUST NOT BE TYPE CHARACTER	Exponentiation cannot be performed using character operands.	Correct operand type.
FATAL	OPERAND TO ** OPERATOR MUST NOT BE LOGICAL	Exponentiation cannot be performed using logical operands.	Correct operand type.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	OVCAP DIRECTIVE CAN APPEAR ONLY WITH SUBROUTINES	OVCAP directives can only appear with subroutines.	Revise job program type.
FATAL	OVERLAY DIRECTIVE MUST BEGIN WITH LEFT PAREN	OVERLAY directives must begin with left parenthesis.	Add left parenthesis.
FATAL	PARAMETER REQUIRES INTEGER EXPONENTIATION	Integer exponentiation is required with this parameter.	Provide integer exponentiation for this parameter.
FATAL	PARAMETER TYPE OR CHARACTER LENGTH CANNOT BE MODIFIED AFTER PARAMETER STATEMENT	Length of symbolic constant must not be changed by an IMPLICIT statement or other statements following a PARAMETER statement.	Correct usage of symbolic constant.
FATAL	PREMATURE E.O.S.	Premature end of statement.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. -- EXPECTED BLOCK NAME	End of statement encountered before a block name was found.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. -- EXPECTED SYMBOL OR SLASH	End of statement encountered before a symbol or slash was found.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN ENCODE OR DECODE	End of statement encountered; ENCODE or DECODE statement incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN I/O CONTROL LIST	End of statement encountered; I/O control list incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. IN I/O LIST ITEM SUBSCRIPT	End of statement encountered; I/O list item subscript incomplete.	Check for incomplete statement.
FATAL	PREMATURE E.O.S. OR MISSING RIGHT PAREN	End of statement encountered or right parenthesis missing.	Check for incomplete statement.
FATAL	PREVIOUS REFERENCE TO DO LABEL IS ILLEGAL	A DO label must not be referenced from outside the DO loop.	Check all previous references to the label.
FATAL	PREVIOUS REFERENCE TO FORMAT LABEL IS ILLEGAL	The label was previously defined or referenced as a FORMAT label.	Check all previous references to the label.
FATAL	PREVIOUS REFERENCE TO LABEL WAS ILLEGAL	Illegal reference to label.	Correct reference to the label.
FATAL	PROGRAM LENGTH EXCEEDS 2**17-1	Program too large.	Shorten program or break up into several routines.
FATAL	RECORD LENGTH EXCEEDS 131071 COLUMNS	Record too large. Error in FORMAT statement.	Check for incorrect repeat specification, hollerith count, and format specification.
FATAL	RECORD LENGTH FOR FILE EXCEEDS 'MAX. RECL' B -- DEFINITION IGNORED	Record length too large.	Reduce record length.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	RECURSIVE DEFINITION OF STATEMENT FUNCTION	The function name appears on both sides of an equal sign.	Remove function name from the right side of the equal sign.
FATAL	REFERENCE TO EXTERNAL REQUIRES AN ARGUMENT LIST	Function requires argument list.	Supply appropriate argument list.
FATAL	REFERENCE TO VARIABLE AS A FUNCTION OR ARRAY	The variable has a subscript or argument list, but is not declared as an array or function.	Check for missing declaration.
FATAL	REPEAT COUNT IS NOT ALLOWED BEFORE THE EDIT DESCRIPTOR	A repeat count was used with a descriptor that does not allow one.	Remove repeat count.
FATAL	SCALAR FORMAT SPECIFIER MUST BE INTEGER	Scalar format is required to be integer.	Make scalar format integer.
FATAL	'SCM' COMMON BLOCKLENGTH EXCEEDS 131071	Common block too large.	Break common block into two or more common blocks.
FATAL	SEPARATOR MISSING AT	Error in FORMAT statement.	Correct FORMAT statement.
FATAL	SEQUENCE NUMBER OUT OF ORDER	Sequence number was specified out of order.	Correct statement sequence number.
FATAL	SIGNED COUNT ALLOWED ONLY BEFORE P EDIT DESCRIPTOR	Signed count used illegally.	Correct use of signed count.
FATAL	SIZE OF ARRAY EXCEEDS 1048568	Array too large.	Reduce size of array.
FATAL	SIZE OF ARRAY EXCEEDS 131071	Array too large.	Reduce size of array.
FATAL	SLASH MUST BE FOLLOWED BY AN OCTAL OR INTEGER CONSTANT	Octal or integer constant missing after slash.	Put octal or integer after slash.
FATAL	STAR DUMMY ARGUMENT ILLEGAL IN FUNCTION	Alternate returns illegal in functions.	Remove alternate returns.
FATAL	STATEMENT FUNCTION -- MISPLACED EQUAL SIGN	Syntax error in statement function.	Correct syntax error in statement function.
FATAL	STATEMENT FUNCTION INDIRECTLY REFERENCES ITSELF	Recursive statement functions are not allowed.	Check all appropriate statement functions for indirect recursion.
FATAL	STATEMENT FUNCTION OF TYPE CANNOT ACCEPT RESULT OF TYPE	Result of statement function expansion cannot be converted to statement function type.	Check typing of statement function.
FATAL	STATEMENT FUNCTION DEFINITION MUST OCCUR BEFORE FIRST EXECUTABLE	Definition must precede first executable statement.	Move statement function definition, or check for undimensioned array.
FATAL	STATEMENT FUNCTION DUMMY ARGUMENT CANNOT BE ASSUMED LENGTH	Dummy argument name appeared in a CHARACTER*(*) declaration.	Change type declaration or dummy argument name.
FATAL	STATEMENT FUNCTION DUMMY ARGUMENT MUST BE USED AS SIMPLE VARIABLE	Dummy argument was followed by expression in parentheses.	Rewrite statement function expression.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	STATEMENT FUNCTION DUMMY PARAMETER NOT SIMPLE VARIABLE	A constant or expression appears in the parameter list of a function definition.	Check parameter list of the function def- inition.
FATAL	STATEMENT FUNCTION INVALID IN DATA VARIABLE LIST	Attempt to use statement function that is in DATA statement.	Rewrite statement.
FATAL	STATEMENT LABEL CONTAINS NON-DIGIT	Statement labels must consist of digits.	Correct statement labels.
FATAL	STATEMENT LABEL EXCEEDS 5 DIGITS	Statement labels must be five digits or less.	Correct statement labels.
FATAL	STATEMENT LABEL MUST BE NUMERIC	Statement labels must consist of digits.	Correct statement labels.
FATAL	STATEMENT LABEL REFERENCED BUT NOT DEFINED	The indicated label does not appear as a statement label anywhere in the program.	Check for missing statement.
FATAL	STATEMENT LABEL EXPECTED BUT NOT FOUND	A statement label reference is missing.	Insert label.
FATAL	STATEMENT MISPLACED	Statement in the wrong place.	Put statement in proper place.
FATAL	STRING ADDRESS CANNOT BE	Invalid string address on encode or decode.	Check string address.
FATAL	STRING ADDRESS CANNOT BE CHARACTER	String address on encode or decode cannot be type character.	Change type of string address.
FATAL	SUBROUTINE ENTRY MAY NOT APPEAR IN A DECLARATIVE STATEMENT	Subroutine entry cannot appear in a declarative statement.	Check declarative statement.
FATAL	SUBSCRIPT OF IS NOT A NUMERIC TYPE	Subscripts must be numeric.	Make subscripts numeric.
FATAL	SUBSCRIPTS IN DATA MUST BE INTEGER	Subscripts must be integer.	Make subscripts integer.
FATAL	SUBSTRING ILLEGAL FOR OPERAND	Wrong substring for operand.	Check substring.
FATAL	SUBSTRING ILLEGAL FOR PARAMETER	Wrong substring for para- meter.	Check substring.
FATAL	SUBSTRINGED VARIABLE NOT TYPE CHARACTER	Variable must be character type.	Check substring vari- able.
FATAL	SYNTAX ERROR IN BLOCK NAME	Wrong syntax in block name.	Check block name for syntax error.
FATAL	SYNTAX ERROR IN DATA CONSTANT LIST	Wrong syntax in data constant list.	Check data constant list for syntax error.
FATAL	SYNTAX ERROR IN DATA STATEMENT	Wrong syntax in data state- ment.	Check data statement for syntax error.
FATAL	SYNTAX ERROR IN DIMENSION DECLARATION	Wrong syntax in dimension declaration.	Check dimension decla- ration for syntax error.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	SYNTAX ERROR IN EQUIVALENCE STATEMENT	Wrong syntax in EQUIVALENCE statement.	Check EQUIVALENCE statement for syntax error.
FATAL	SYNTAX ERROR IN GO TO STATEMENT	Wrong syntax in GO TO statement.	Check GO TO statement for syntax error.
FATAL	SYNTAX ERROR IN I/O CONTROL LIST AT	Wrong syntax in I/O control list.	Check I/O control list for syntax error.
FATAL	SYNTAX ERROR IN I/O IMPLIED DO	Wrong syntax in I/O implied DO.	Check I/O implied DO for syntax error.
FATAL	SYNTAX ERROR IN NAMELIST	Wrong syntax in NAMELIST.	Check NAMELIST for syntax error.
FATAL	SYNTAX ERROR IN PROGRAM UNIT NAME	Wrong syntax in program unit name.	Check program unit name for syntax error.
FATAL	SYNTAX ERROR IN SUBSTRING EXPRESSION FOR	Wrong syntax in substring expression.	Check substring expression for syntax error.
FATAL	SYNTAX OF DO MUST BE I=M1,M2,M3 OR M1,M2	DO statement syntax incorrect.	Use correct syntax.
FATAL	T EDIT DESCRIPTOR FOLLOWED BY ZERO OR NON-DIGIT	T code must be followed by nonzero column number.	Correct column number.
FATAL	TABLE OVERFLOW -- INCREASE FIELD LENGTH AND RERUN	Not enough field length for compilation.	Provide more field length for compilation.
FATAL	TERMINAL DELIMITER	The terminal delimiter is missing.	Provide correct terminal delimiter.
FATAL	THE TERMINAL STATEMENT OF DO PRECEDED THE DO DEFINITION	Terminal statement of DO must not precede the DO definition.	Provide terminal statement of DO in proper place.
FATAL	THIS IS NOT A FORTRAN STATEMENT	Unrecognizable statement.	Check syntax.
FATAL	THIS STATEMENT MAY NOT BE A DO TERMINAL	A DO loop cannot end with the specified statement.	Restructure DO loop.
FATAL	THIS STATEMENT MUST BE CONTAINED ON 1 CARD	Continuation lines not allowed for this statement.	Rewrite statement to fit on 1 line.
FATAL	TOO FEW LEFT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT	Left paren missing or there is a previous syntax error.	Check parenthesis matching or correct previous syntax.
FATAL	TOO FEW RIGHT PAREN OR PREVIOUS SYNTAX ERROR -- SCAN STOPPED AT	Right paren missing or there is a previous syntax error.	Check parenthesis matching or correct previous syntax.
FATAL	TRIP COUNT IS LESS THAN ONE	Trip count must be at least one if DO=OT is selected.	Make trip count at least one.
FATAL	TRIP COUNT OF MUST BE POSITIVE	Trip count must be positive.	Make trip count positive.
FATAL	UNBALANCED PARENS	Parentheses are unbalanced.	Balance parentheses.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
FATAL	UNDECLARED INTRINSIC OR EXTERNAL FUNCTION USED AS ACTUAL ARGUMENT	Cannot use undeclared function as actual argument.	Remove undeclared function.
FATAL	UNIT SPECIFIER FILE NAME GREATER THAN 7 CHARACTERS	Illegal file name.	Check character length of unit specifier.
FATAL	UNIT SPECIFIER NOT LEGAL FILE NAME	Illegal file name.	Check all uses of the file name.
FATAL	UNIT SPECIFIER OUTSIDE RANGE 0 - 999	Illegal unit number.	Provide a unit number which is no more than 3 digits long.
FATAL	UNIT SPECIFIER MISSING	Unit specifier required.	Provide a unit number.
FATAL	UNKNOWN EDIT DESCRIPTOR	EDIT descriptor not recognized.	Check EDIT descriptor.
FATAL	UNMATCHED PARAMETER COUNT TO STATEMENT FUNCTION	The function reference and function definition contain different numbers of parameters.	Check for missing parameters.
FATAL	USAGE CONFLICT -- CANNOT BE STATEMENT FUNCTION	The indicated statement function conflicts with a previous usage.	Check all other usages; the function name might be used as a variable or array name.
FATAL	USAGE CONFLICT -- IS AND CANNOT BE	Usage conflict.	Check uses of indicated name.
FATAL	USAGE CONFLICT PREVIOUSLY USED AS	The label was previously used another way.	Check previous usage of label.
FATAL	USAGE CONFLICT -- PREVIOUSLY DEFINED AS DO TERMINAL	The label was previously defined as a DO terminal.	Check previous loops for use of the same label.
FATAL	USAGE CONFLICT -- PREVIOUSLY DEFINED AS FORMAT	The label was previously defined as a FORMAT label.	Change label.
FATAL	USAGE CONFLICT -- PREVIOUSLY USED AS A FORMAT LABEL	The label was previously used as a Format label.	Change label.
FATAL	ZERO IS SPECIFIED AS REPEAT COUNT AT	Repeat count must be greater than zero.	Make repeat count greater than zero.
FATAL	ZERO LENGTH CHARACTER OR HOLLERITH STRING	Character or hollerith string must have a positive nonzero length.	Make string positive nonzero length.
MDEP	BOOLEAN DATA TYPE IS MACHINE DEPENDENT	This data type is machine dependent.	Use the CHARACTER data type instead, for portability.
MDEP	BUFFER I/O IS MACHINE DEPENDENT	Buffer I/O is machine dependent.	Avoid using Buffer I/O if possible, especially usages that depend on the number of characters per word.
MDEP	ENCODE/DECODE ARE MACHINE DEPENDENT	ENCODE/DECODE is machine dependent.	Use internal files instead for portability.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
MDEP	LIBRARY FUNCTIONS DATE, TIME AND CLOCK ARE MACHINE DEPENDENT	These functions are machine dependent.	Do not dismantle the output of these routines, print them out as a whole.
MDEP	OVCAPS ARE MACHINE DEPENDENT	OVCAPS are machine dependent.	Do not let programs depend on certain properties of OVCAPS, such as reinitialization of variables when an OVCAP is reloaded.
MDEP	OVERLAYS ARE MACHINE DEPENDENT	OVERLAYS are machine dependent.	Do not let programs depend on certain properties of overlays, such as reinitialization of variables when an overlay is reloaded.
TRIVIAL	ARGUMENT IS NOT USED IN STATEMENT FUNCTION	Specified argument not needed.	Remove argument.
TRIVIAL	CONSTANT ** CONSTANT CANNOT BE EVALUATED	Specified operation cannot be performed at compile time.	Change the expression.
TRIVIAL	CONSTANT TOO LONG, EXCESS DIGITS TRUNCATED	Constant truncated due to excess length.	Remove excess digits.
TRIVIAL	CONTINUE WITH NO STATEMENT LABEL -- IGNORED	CONTINUE without statement label is meaningless.	Insert label or eliminate CONTINUE.
TRIVIAL	IF RESULTS IN A SIMPLE TRANSFER	The IF can be replaced by a GO TO.	Make the substitution.
TRIVIAL	IF RESULTS IN A TRANSFER TO THE NEXT LINE	Control will always transfer to the next statement, regardless of the condition specified in the IF statement.	Reexamine the IF.
TRIVIAL	INTEGER ** NEGATIVE CONSTANT -- RESULTS ZERO	Integer raised to a negative power is zero.	Change the integer to real.
TRIVIAL	LAST IF RESULTS IN A NULL TRANSFER TO THIS STATEMENT	IF acts as a do-nothing statement.	Check syntax of IF.
TRIVIAL	MISSING PROGRAM STATEMENT -- PROGRAM START. ASSUMED	The PROGRAM statement must be the first statement of the main program.	Supply PROGRAM statement.
TRIVIAL	NO PATH TO THE ENTIRE RANGE OF DO	The statements in the loop cannot be reached.	Check for logic error, in current branch.
TRIVIAL	NO PATH TO THIS STATEMENT	Statement will never be executed.	Check program logic, particularly GO TO statements and IF statements.
TRIVIAL	NULL TRANSFER STATEMENT -- TRANSFER IGNORED	A GO TO statement transfers to the next statement.	GO TO can be eliminated.
TRIVIAL	RECORD LENGTH EXCEEDS 137 COLUMNS -- MAY EXCEED I/O DEVICE	Record length may be too large for peripheral device.	Reduce record length if necessary.
TRIVIAL	STATEMENT CAN TRANSFER TO ITSELF	Infinite loop possible.	Revise statement.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
TRIVIAL	STATEMENT TRANSFERS TO ITSELF	Infinite loop results.	Change statement.
TRIVIAL	THIS DO LOOP WILL NOT EXECUTE	Condition always prohibits execution of DO loop.	Check logic of DO loop.
TRIVIAL	TL EDIT DESCRIPTOR BACKSPACED BEYOND 1st COLUMN -- COLUMN POINTER RESET AT 1	Value of TL code is too large.	Check TL code.
TRIVIAL	VARIABLE ** ZERO -- RESULT ASSUMED ONE	Variable raised to zero power is equal to one.	Check expression.
TRIVIAL	ZERO ** ZERO -- RESULTS INDEFINITE	Zero raised to zero power is indefinite.	Check expression.
WARNING	*TO* ASSUMED FOR	Syntax error in ASSIGN statement.	Check ASSIGN statement for syntax error.
WARNING PREVIOUSLY DECLARED INTRINSIC -- IGNORED	Function already declared.	Check declaration of functions.
WARNING PREVIOUSLY DECLARED EXTERNAL -- IGNORED	Function already declared.	Check declaration of functions.
WARNING PREVIOUSLY TYPED NON-CONFORMING -- PREVIOUS TYPE OVERRIDDEN	Most recent declaration used.	Check declarations.
WARNING REDUNDANTLY DECLARED IN SAVE	The indicated name appears more than once in a SAVE statement.	Eliminate redundancy.
WARNING	ARGUMENT TO MASK MUST BE BETWEEN 0 AND 60	Argument to mask is not between 0 and 60.	Make argument to mask between 0 and 60.
WARNING	C\$ PARAMETER VALUE FOR ON MUST BE 0 OR 1	C\$ parameter must be zero or one.	Check C\$ parameter.
WARNING	COMMA AFTER STATEMENT LABEL IGNORED	Comma is not needed.	Remove comma.
WARNING	COMMA MUST FOLLOW LEVEL NUMBER	Comma missing after level number.	Insert comma.
WARNING	CONFLICT IN RANGE INDICATOR -- FIRST RETAINED	Overlap of ranges in IMPLICIT statement.	Check for overlap of ranges in IMPLICIT statement.
WARNING	CONSTANT EXCEEDS 5 DIGITS -- TRUNCATED	Constant too long.	Reduce number of digits in constant to 5 or less.
WARNING	CONSTANT MISSING EXPONENT FIELD -- ZERO ASSUMED	Exponent field missing in constant; zero assumed.	Provide constant with an exponent field.
WARNING	DO CONCLUSION NOT COMPILED -- DO DEFINITION ERROR	Error in DO definition; DO conclusion not compiled.	Correct previous errors.
WARNING	ENTRY MUST NOT BE DECLARED EXTERNAL -- IGNORED	The entry must not be declared external.	Correct declaration of entry.
WARNING	ENTRY STATEMENT IGNORED IN MAIN PROGRAM	An ENTRY statement in the main program has no purpose.	Remove ENTRY statement.
WARNING	EXCESS CONSTANTS IGNORED	Too many constants.	Reduce excess number of constants.
WARNING	EXPECTED COMMA AFTER I/O CONTROL -- FOUND	Comma should have followed I/O control statement.	Provide comma after I/O control statement.
WARNING	EXPECTED E.O.S. -- FOUND AND IGNORED	Extraneous information follows a legal statement.	Remove extra characters.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
WARNING	EXPECTED LEFT PAREN -- FOUND	Left parenthesis not found.	Check syntax of statement.
WARNING	EXTRANEOUS COMMA IGNORED	Comma unrecognized and ignored.	Remove extraneous comma.
WARNING	FIELD WIDTH IS LESS THAN MINIMUM REQUIRED ON EDIT DESCRIPTOR AT	Field width too small.	Increase field width.
WARNING	FUNCTION REFERENCED AS SUBROUTINE	Function names must not be the object of CALL statements.	Use function reference syntax.
WARNING	FWA AND LWA NOT IN SAME ARRAY, EQUIVALENCE CLASS, OR COMMON BLOCK	First-word-address and last-word-address must be in the same common block, equivalence class, or array.	Check declarations for inconsistencies involving FWA and LWA.
WARNING	GENERIC ONLY INTRINSIC TYPED-- TYPING IGNORED	Name of intrinsic function, which is not specific function, appears in type	Remove attempted typing.
WARNING	HOLLERITH CONSTANT EXCEEDS 10 CHARACTERS	Self-explanatory.	Reduce constant to 10 characters or less.
WARNING	I/O LIST IGNORED WHEN USING NAMELIST	Namelist I/O does not use an I/O list.	Eliminate I/O list.
WARNING	ILLEGAL NAME -- ENTRY STATEMENT IGNORED	Name invalid.	Provide legal name.
WARNING	INTRINSIC TYPED NON-CONFIRMING -- TYPE IGNORED	Declared type of intrinsic disagrees with implicit type.	Change type declaration.
WARNING	LOCAL IN BLOCK DATA -- IGNORED	Variable appears in BLOCK DATA subprogram, but not in a common statement.	Check common block for missing variables.
WARNING	MISSING NAME -- ENTRY STATEMENT IGNORED	ENTRY statement needs a name.	Provide name for ENTRY statement.
WARNING	MULTIPLY DEFINED LEVEL FOR NAME -- IGNORED	Too many levels defined for name.	Check defined levels of name.
WARNING	MULTIPLY DEFINED LEVEL FOR COMMON BLOCK NAME -- IGNORED	Too many levels defined for common block name.	Check defined levels of common block name.
WARNING	NAME PREVIOUSLY DEFINED -- ENTRY STATEMENT IGNORED	Too many definitions of ENTRY name.	Check for another usage of the ENTRY name.
WARNING	NON-OCT/HEX DIGIT IN OCT/HEX CONSTANT -- IGNORED	Digit must be 0-7 for octal; 0-9, A-F for hexadecimal.	Rewrite octal or hexadecimal constant.
WARNING	NUMBER OF ARGUMENTS IN REFERENCE TO IS NOT CONSISTENT	Number of arguments in reference must be the same as the number of arguments in the FUNCTION or SUBROUTINE statement.	Check arguments.
WARNING	OBJECT OF GO TO NOT INTEGER VARIABLE	Object of GO TO must be a simple integer variable.	Make object of GO TO integer variable.
WARNING	ONLY 49 FILES ARE ALLOWED -- EXCESS IGNORED	Too many files were specified in the PROGRAM statement.	Reduce number of excess files.
WARNING	PREMATURE E.O.S. -- EXPECTED VARIABLE AT	End of statement encountered; statement incomplete.	Check syntax.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
WARNING	PREMATURE E.O.S. OR EXTRA TRAILING SEPARATOR	End of statement encountered or an extraneous separator found.	Check statement or eliminate extra separator.
WARNING	PREVIOUS DEFINITION OF STATEMENT FUNCTION IS OVERRIDDEN	The function was defined more than once; the most recent definition is used.	Change second definition.
WARNING	RANGE INDICATOR NOT 1 LETTER -- TRUNCATED TO	Implicit statement range indicator not 1 letter.	Change the range indicator to 1 letter.
WARNING	REDUNDANT EQUIVALENCE SPECIFICATION FOR	EQUIVALENCE specification used before.	Check for occurrence of indicated symbol in previous EQUIVALENCE statement.
WARNING	SHIFT COUNT MUST BE BETWEEN -60 and 60	SHIFT count is not between -60 and 60.	Make SHIFT count between -60 and 60.
WARNING	STATEMENT FUNCTION HAS NULL DEFINITION -- IGNORED	Statement function expansion reduces to a null code sequence.	Check for error in function definition statement.
WARNING	SUBROUTINE APPEARED IN TYPE DECLARATION	Subroutine named on CALL statement was typed.	Remove type declaration.
WARNING	SUBROUTINE CANNOT BE TYPED -- TYPING IGNORED	Subroutine entry name appeared in type statement.	Remove type declaration.
WARNING	SUBROUTINE REFERENCED AS FUNCTION	Subroutines are referenced with the CALL statement.	Use CALL statement.
WARNING	SUBSCRIPT OF VIOLATES LOWER DIMENSION BOUND	Subscript less than declared lower bound.	Correct subscript.
WARNING	SUBSCRIPT OF VIOLATES UPPER DIMENSION BOUND	Subscript greater than declared upper bound.	Correct subscript.
WARNING	TARGET/SOURCE OVERLAP IN CHARACTER ASSIGNMENT STATEMENT	None of the elements in a character expression can overlap the target variable that the expression is being assigned to.	Eliminate all instances of overlap in the assignment.
WARNING	TERMINAL CHARACTER CONVERTED TO RIGHT PAREN	The indicated character appeared where a right parenthesis was expected.	Compiler assumes a right parenthesis.
WARNING	THIS STATEMENT HAS NO INITIAL LINE -- INITIAL ASSUMED	Initial line missing from statement.	Provide initial line.
WARNING	TOO FEW CONSTANTS -- VARIABLES FROM NOT INITIALIZED	Not enough constants in data constant list.	Initialize the variables; uninitialized variables can cause run-time errors.
WARNING	TRIVIAL EQUIVALENCE GROUP WITH ONLY 1 MEMBER IS IGNORED	An EQUIVALENCE must contain at least 2 members.	Check EQUIVALENCE statement.
WARNING	TRIVIAL RANGE -- SAME AS	Implicit range is trivial.	Check range.
WARNING	TYPING OF IGNORED -- PRIOR TYPING RETAINED	The symbol appeared in more than one type statement; first type is used.	Eliminate second type declaration.
WARNING	UNIVERSAL SAVE DECLARED -- OTHER SAVE STATEMENTS ARE REDUNDANT	When universal SAVE declared, other SAVE statements are not necessary.	Eliminate redundant SAVE statements.

TABLE B-1. COMPILE-TIME DIAGNOSTICS (Contd)

	Message	Significance	Action
WARNING	UNKNOWN FORM -- BLANK ASSUMED	Unrecognizable form of STOP or PAUSE statement.	Check STOP or PAUSE statement.
WARNING	VARIABLE HAS NO DIMENSION BOUND -- IGNORED	Variable label must have dimension bound.	Provide dimension bound for variable label.
WARNING	VARIABLE NOT INTEGER	Variable must be integer.	Make variable integer.

TABLE B-2. SPECIAL COMPILATION DIAGNOSTICS

Message	Significance	Action
<p>COMPILING program LAST STATEMENT BEGAN AT LINE nnnnn ERROR AT aaaaa IN ddddddd LAST OVERLAY LOADED - (p,s)</p>	<p>Compiler, operating system, or hardware error has occurred while compiling program.</p> <p>program Name of source program unit.</p> <p>nnnnn Approximate compiler-assigned source line number where the difficulty arose. During transitions from one phase of compilation to another, the END line number might be displayed.</p> <p>ddddddd Name of compiler internal deck where abort occurred. Might be RA+0 if control was accidentally transferred to the control point job communications area.</p> <p>aaaaaa Address relative to origin of internal deck where abort occurred.</p> <p>p,s Primary and secondary level numbers of overlay last loaded before abort occurred:</p> <p>0,0 - Control statement cracker; global communication and control</p> <p>1,0 - (OPT=0) compilation overlay</p> <p>2,0 - (OPT>0) compilation batch controller</p> <p>2,1 - (OPT>0) compilation normal pass 1 (lexical scan, parse, intermediate language generation)</p> <p>2,2 - (OPT>0) compilation pass 2 (global and local optimization, object code generation)</p> <p>2,3 - (OPT>0) compilation reference map generation and object code assembly phase</p>	<p>Follow site-defined procedures for reporting software errors or operational problems.</p>
<p>DEAD CODE IN program</p>	<p>A section of code is unreachable and cannot be processed (can be issued only when OPT>2).</p>	<p>Same as STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED.</p>
<p>EMPTY INPUT FILE. NO COMPILATION.</p>	<p>An end-of-partition or end-of-section was encountered on the first read of the input.</p>	<p>Check for extra EOP or EOS, or mispositioned input file.</p>

TABLE B-3. CONTROL STATEMENT DIAGNOSTICS

Message	Significance	Action
..... .GT. 10 CHAR	Indicated parameter length greater than allowable. Check spelling.	Correct indicated portion of control statement.
..... ANSI MUST BE O, T OR F	Invalid level for ANSI diagnostics.	Choose a correct level.
..... EL MUST BE T, W, F, OR C	Invalid error level selected.	Choose a correct level.
..... ET MUST BE O, T, W, F, OR C	Invalid error termination level selected.	Choose a correct level.
..... HAS ILLEGAL BINARY VALUE	Indicated parameter has invalid binary value.	Check and select a correct form.
..... ILLEGAL FILENAME	Indicated filename is invalid. Can be system dependent.	Select a correct filename. Change other control statements which use that filename.
..... ILLEGAL NUMERIC	Indicated value was used, numerical value was expected.	Choose a numeric value.
..... LCM OPTION MUST BE G, I, OR D	Invalid LCM access specified.	Choose a correct access.
..... MD MUST BE O, T, OR F	Invalid machine dependent diagnostic level selected.	Choose a correct level.
..... TOO MANY OCCURRENCES	Indicated parameters present more times than allowed.	Eliminate excess occurrences.
..... TOO MANY OPTIONS	Indicated parameter has more than the allowable number of options.	Reduce to a valid number of options
..... UNRECOGNIZABLE	Indicated attempt is not a valid parameter.	Check spelling; consult section 11 for valid parameters.
..... UNKNOWN OPTION	Indicated option is not valid for that parameter.	Check section 11 for valid options of the parameter.
ARG CANT BE BOTH COMMON AND FIXED	Attempt made to specify both types of argument list styles.	Choose one or the other or neither.
B=O AND GO IS INCONSISTENT	GO cannot work without a binary file.	Drop GO or select a binary file.
CS MUST BE USER OR FIXED	Invalid syntax for collate sequence parameter.	Choose a correct value.
E=O IS ILLEGAL	Error level, but not error file can be suppressed.	Provide named file or use default.
FILE USE CONFLICT	Multiple use of file name.	Select distinct file names.

TABLE B-3. CONTROL STATEMENT DIAGNOSTICS (Contd)

Message	Significance	Action
I=0 IS NOT ALLOWED	A file name for input must be provided or default used.	Use default or supply file name.
ILLEGAL CHARACTER	Invalid character found on control statement.	Check control statement.
ILLEGAL G OPTION SYNTAX	Bad G file syntax.	Correct the syntax.
ILLEGAL S OPTION SYNTAX	Gad S file syntax.	Correct the syntax.
INTERACTIVE DEBUG REQUIRES OPT=0	DB=ID and OPT other than 0 were selected.	Change OPT to 0 or do not request interactive debug.
ONLY 7 SYSTEMS TEXTS ALLOWED	The sum of the S and G text names cannot exceed 7.	Reduce the number of texts. May require splitting into two or more separate compilations.
OPT LEVEL NOT 0,1,2, OR 3	Invalid optimization level selected.	Choose a valid level.
PD ARGUMENT NOT 6 OR 8	Invalid print density selected.	Choose a valid density.
PL MUST NOT EXCEED 999999999	Print limit too high.	Select a lower limit.
PS .LT. 4	Page size too small to print listings.	Increase page size.
PW .LT. 50 OR .GT. 136	Page width out of valid range.	Choose a width in the valid range.
TERMINATOR MISSING	A terminator is missing.	Provide the valid terminator.
X=0 IS NOT ALLOWED	External text must be named, if used.	Select a valid name.

TABLE B-4. COMPILER OUTPUT LISTING MESSAGES

Message	Significance	Action
STATEMENTS BEGINNING AT THE BELOW LINE NUMBERS ARE UNREACHABLE (DEAD CODE), AND WILL NOT BE PROCESSED.	Executable statements in the source program can never be executed, due to program flow of control. No object code is compiled for dead statements. Accompanied by dayfile message DEAD CODE IN program. Detected only when OPT=2 has been selected. Note that due to code movement by the OPT=2 optimizer, line numbers named are approximate. This is especially true of unreachable code within block if blocks.	Check flow control of program.

TABLE B-5. EXECUTION-TIME DIAGNOSTICS

No.	Class	Message	Significance	Action	Issued By
1	F A	ERROR IN COMPUTED GO TO STATEMENT - INDEX VALUE INVALID	Value .LT. 1 or .GT. number of statement numbers. Occurs only if FORTRAN Extended 4 binary is used in a FORTRAN 5 job.	Recompile using FORTRAN 5 com- piler.	GOTOER=
2	I A	ARGUMENT ABS VALUE .GT. 1 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ACOSIN=(ACOS)
3	I A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ALOG
4	I A	ARGUMENT ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ALOG10
5	I A	ARGUMENT ABS VALUE .GT. 1 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ACOSIN=(ASIN)
6	I A	ARGUMENT INDEFINITE	Note 1	Note 2	ATAN
7	I A	ARGUMENT VECTOR ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	ATAN2
8	I A	ARGUMENT TOO LARGE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	CABS
9	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ZTOI (Z**I)
10	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CCOS
11	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CEXP
12	I T	ZERO ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	CLOG
13	I A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SINCOS=(COS)
14	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT ABS (REAL PART) TOO LARGE ABS (IMAG PART) TOO LARGE	Note 1	Note 2	CSIN
15	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	CSQRT

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
16	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE BASE IN EXPONENTIATION INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOX (D**X)
17	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DATAN
18	I A	ARGUMENT VECTOR 0,0 ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DATAN2
19	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOD (D**D)
20	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOI (D**I)
21	I T	FLOATING OVERFLOW IN D** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)LOG(D) TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DTOZ (D**Z)
22	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DCOS
23	I A	ARGUMENT TOO LARGE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	DEXP
24	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DLOG
25	I T	ZERO ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DLOG10
26	I T	DP INTEGER EXCEEDS 96 BITS 2ND ARGUMENT ZERO INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DMOD
28	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DSIN
29	I T	NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	DSQRT

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
30	I A	ARGUMENT TOO LARGE, FLOATING OVERFLOW ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	EXP
31	I T	INTEGER OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER	Note 1	Note 2	IT0J (I**J)
33	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOD (X**D)
34	I T	ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOI (X**I)
35	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	XTOY (X**Y)
36	I A	ARGUMENT TOO LARGE, ACCURACY LOST ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SINCOS=(SIN)
39	I A	ARGUMENT NEGATIVE ARGUMENT INFINITE ARGUMENT INDEFINITE	Note 1	Note 2	SQRT
40	I T	ILLEGAL SENSE SWITCH NUMBER	Number not in range 1-6; return parameter set to 2.		SSWTCB
41	I T	ARGUMENT TOO LARGE, ACCURACY LOST INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	TAN
42	I T	INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	TANH
44	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE DOUBLE POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ITOD (I**D)
45	I T	FLOATING OVERFLOW ZERO TO THE ZERO POWER ZERO TO THE NEGATIVE POWER NEGATIVE TO THE REAL POWER INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	ITOX (I**X)
46	I T	FLOATING OVERFLOW IN I** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(I) TOO LARGE INFINITE ARGUMENT INDEFINITE ARGUMENT	Note 1	Note 2	IT0Z (I**Z)

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
47	I T	FLOATING OVERFLOW IN X** REAL(Z) ZERO TO THE ZERO OR NEGATIVE POWER NEGATIVE TO THE COMPLEX POWER IMAG(Z)*LOG(X) TOO LARGE INFINITE OR INDEF ARGUMENT	Note 1	Note 2	XTOZ(X**Z)
49	I A I A I A	COMMA MISSING AT END OF RECORD - COMMA ASSUMED NAMELIST DATA TERMINATED BY EOF NOT \$ CONSTANTS MISSING AT END OF RECORD - NEXT RECORD READ	Error occurred during NAMELIST processing.	Check NAMELIST input data for errors.	NAMIN=
50	F A	FATAL ERROR IN LOADER.	Error occurred during load.	Inspect load map to determine cause of error.	OVERLA=
51	I A	Set by user via subroutine SYSTEM or SYSTEMC.	Defined by user.	Defined by user.	USER
52	F A	Set by user via subroutine SYSTEM or SYSTEMC. Error numbers larger than any listed in this table become error 52.	Defined by user.	Defined by user.	USER
53	F A	NOT ENOUGH FL FOR SORT/MERGE.	More memory required for Sort/Merge processing.	Extend program field length.	SMxxxx=
54	F A	MIXED IO MODES - { DIRECT SEQUENTIAL } { CODED } { BINARY } AFTER { DIRECT SEQUENTIAL } { BUFFER } { CODED } { BINARY } ON LFN-xxxxxxx. { BUFFER }	User is trying to switch I/O mode on a file without an intervening REWIND, ENDFILE, or CLOSE/ OPEN sequence.	Continue using previous I/O mode or perform a REWIND, END- FILE, or CLOSE/ OPEN sequence.	FORSYS=
55	F A	END-OF-FILE ENCOUNTERED, FILENAME - - - - xxxxxxxx.	Attempt to read past end-of-file.	Rewind before reading or cor- rect program logic.	BUFIN=
56	F A	WRITE FOLLOWED BY READ, FILENAME - - - - xxxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	BUFIN=
57	F A	AREA SPECIFICATION SPANS SCM/LCM.	In a buffered I/O statement the first and last word addresses must be in the same level of memory.	Check word addresses in buffered I/O statement.	BUFIO=
58	F A	BUFFER DESIGNATION BAD -- FWA.GT.LWA.	First-word address must be LE last word address.	Check buffer designation.	BUFIO=
59	F A	BUFFER SPECIFICATION BAD -- FWA.GT.LWA	First-word address must be LE last word address.	Check first and last word address.	BUFOUT=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
60	F A	BFS EXCEEDS ALLOCATED STATIC SIZE, LFN-xxxxxxx.	User has specified a larger file card BFS than is available in a statically compiled program.	Decrease file card BFS parameter or recompile program with a larger buffer size on the PROGRAM statement file declaration.	FORSYS=
61	F A	PARAMETERS ON EXPLICIT OPEN INCONSISTENT WITH USE OF LFN-xxxxxxx.	User is attempting to change I/O modes on a file that was explicitly opened without first explicitly closing the file.	Issue an explicit close on the file.	FORSYS=
62	F A	INVALID UNIT	Unit not recognized.	Check unit number.	GETFIT=
63	F A	END-OF-FILE ENCOUNTERED ON FILE xxxxxxxx.	Attempt to read past end-of-file.	Rewind file or correct program logic.	INPB=
64	F A	ILLEGAL BUF LEN SPECIFIED ON PROGRAM STATEMENT	For unformatted files, buffer length specified on the PROGRAM statement must be .GE.513.	Check buffer length.	FORSYS=
65	F A	END-OF-FILE ENCOUNTERED ON FILE xxxxxxxx.	Attempt to read past end-of-file.	Rewind file or correct program logic.	INPC= NAMIN=
66	F A F A F A F A F A F A F A	NAMELIST NAME NOT FOUND-xxxxxxx. INCORRECT SUBSCRIPT. TOO MANY CONSTANTS. , (\$ OR = EXPECTED, MISSING. VARIABLE NAME NOT FOUND-xxxxxxx. CONSTANT MISSING. INVALID COMPLEX CONSTANT	Error occurred during NAMELIST processing.	Check NAMELIST input data for errors.	NAMIN=
67	F A	DECODE RECORD LENGTH .LE. 0. DECODE LCM RECORD .GT. 150 CHARACTERS.	Bad first parameter to DECODE.	Check first parameter to DECODE.	DECODE=
68	F A F A	* ILL-PLACED NUMBER OR SIGN. * ILLEGAL FUNCTIONAL LETTER.	Illegal FORMAT.	Check format.	FMTAP=
69	F A	* IMPROPER PARENTHESIS NESTING.	Illegal FORMAT.	Check format.	FMTAP=
70	F A	* EXCEEDED RECORD SIZE.	The maximum record length specified on the PROGRAM, OPEN, ENCODE, DECODE or FILE control statement has been exceeded.	Change RL parameter on PROGRAM statement, MRL parameter on the FILE control statement, RECL parameter on the OPEN statement, or C parameter on the ENCODE or DECODE statement.	FMTAP=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
71	F A	* SPECIFIED FIELD WIDTH ZERO.	w=0 in FORMAT.	Check field width in FORMAT.	FMTAP=
72	F A	* FIELD WIDTH .LE. DECIMAL WIDTH.	w LE d in FORMAT.	Check width in FORMAT.	FMTAP=
73	F A	*HOLLERITH FORMAT WITH LIST.	The FORMAT has no specifiers corresponding to the I/O statement.	Change one or the other.	INCOM=
78	F A	* ILLEGAL DATA IN FIELD * ' *	Usually a nondigit in a numeric input field.	Fix input data.	INCOM=
79	F A	* DATA OVERFLOW * ' *	Input value GT 1.26501E322.	Fix input data.	INCOM=
83	F A	OUTPUT FILE LINE LIMIT EXCEEDED.	The default or specified print limit to OUTPUT was exceeded.	Specify PL on FTN5 statement, PL on execution call, or change program to print less.	OUTC= NAMOUT= OUTF= SYSERR=
85	F A	ENCODE CHARACTER/RECORD .LE. 0 ENCODE LCM RECORD .GT. 150 CHARACTERS	Bad first parameter to ENCODE.	Check first parameter to ENCODE.	ENCODE=
88	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	INPB=
89	F A	LIST EXCEEDS DATA, READ ON FILE-xxxxxxx.	More words were specified in the I/O list than existed in the record of the file.	Check for missing data or incorrect input list.	INPB=
90	F A	PARITY ERROR ON FILE xxxxxxxx DURING PREVIOUS READ.	Probable disk or tape error.	Follow site-defined procedures for reporting software error or operational problems.	INPB=
91	F A	WRITE FOLLOWED BY READ ON FILE-xxxxxxx.	A READ cannot follow a WRITE unless a REWIND intervenes.	Insert a REWIND statement.	INPC=
92	F A	PARITY ERROR READING (CODED) FILE-xxxxxxx.	Probable disk or tape error.	Follow site-defined procedures for reporting software errors or operational problems.	INPC= NAMIN=
93	F A	PARITY ERROR ON FILE-xxxxxxx DURING PREVIOUS READ.	Probable disk or tape error.	Follow site-defined procedures for reporting software errors or operational problems.	OUTB=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
94	F A	PARITY ERROR ON LAST READ ON FILE-xxxxxxx.	Probable disk or tape error.	Follow site-defined procedures for reporting software errors or operational problems.	OUTC=
95	F A	PARITY ERROR ON FILE xxxxxxx DURING PREVIOUS WRITE	Probable disk or tape error.	Follow site-defined procedures for reporting software errors or operational problems.	ODAB
96	F A	PARITY ERROR ON FILE xxxxxxx DURING PREVIOUS READ	Probable disk or tape error	Follow site-defined procedures for reporting software errors or operational problems.	IDAB

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
97	F A	INDEX NUMBER ERROR.	Nonexistent index value specified or bad file.	Check index and file.	RANMS=
98	F A	FILE ORGANIZATION ERR OR FILE NOT OPEN.		Call OPENMS.	RANMS=
99	F A	WRONG INDEX TYPE.	Wrong type specified to OPENMS.	Check index type.	RANMS=
100	F A	INDEX IS FULL.	An index is full, and an attempt is being made to add a new record to it.	Increase index size.	RANMS=
101	F A	DEFECTIVE INDEX CONTROL WORD.	Bad file.	File must be recreated.	RANMS=
102	F A	RECORD LENGTH EXCEEDS SPACE ALLOCATED.	Record length too long.	Increase space allocation.	RANMS= BUFIO=
103	F A	RECORD MANAGER ERROR xxx ON FILE xxxxxxxx, RECORD xxxxxxxx.	Record Manager error.	See Record Manager reference manual.	RANMS=
104	F A	INDEX KEY UNKNOWN.	Invalid index key.	Correct index key.	RANMS=
105	F A	RECORD LENGTH NEGATIVE.	Record length must not be negative.	Fix call.	RANMS=
107	F A	ILLEGAL PARAMETER VALUE.	Argument to Sort/Merge routine has bad value.	Check parameter value of Sort/Merge routine.	SMXXXX=
108	F A	TOO FEW OR TOO MANY PARAMETERS.	Valid number of parameters not provided.	Provide proper number of parameters.	SMXXXX=
109	F A	KEYWORD (xxxxxxx) INVALID.	Keyword not recognized.	Provide legal keyword.	SMxxxx=
110	F A	A ROUTINE CALLED OUT OF SEQUENCE.	Sequence (SMSORT, SMSORTB, SMSORTP, or SMMERGE), (other Sort/Merge calls), (SMEND or SMABT) not followed.	Check sequence of routine call.	SMxxxx=
111	F A	LCM BLOCK COPY ERROR.	Parity error.	Follow site-defined procedures for reporting software errors or operational problems.	COMIO=, DECODE=, ENCODE=, INPB=, OUTB=, READEC, WRITEC
114	F A	CONNEX CHARACTER CODE CONVERSION IS OUT OF RANGE	Bad second argument in CALL CONNEX.	Change to specify correct character set.	CONDIS=
115	I A	ARGUMENT INFINITE ARGUMENT TOO SMALL	Note 1	Note 2	EXP
116	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	HYP=(COSH)

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
117	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	HYP=(SINH)
118	I A	ARGUMENT TOO SMALL	Note 1	Note 2	DEXP
119	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DHYP=(DCOSH)
120	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DHYP=(DSINH)
121	I A	ARGUMENT INDEFINITE	Note 1	Note 2	DTANH
122	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	DTAN
123	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GT. 1.0.	Note 1	Note 2	DASNCS(DASIN)
124	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GT. 1.0	Note 1	Note 2	DASNCS(DACOS)
125	I A	ARGUMENT INDEFINITE	Note 1	Note 2	ERF(ERF)
126	I A	ARGUMENT INDEFINITE	Note 1	Note 2	ERF(ERFC)
127	I A	ARGUMENT TOO LARGE	Note 1	Note 2	ERF(ERFC)
128	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT .GE. 1.0.	Note 1	Note 2	ATANH
129	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	SIND
130	I A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE	Note 1	Note 2	COSD
131	F A	ARGUMENT INFINITE ARGUMENT INDEFINITE ARGUMENT TOO LARGE ARGUMENT ODD MULTIPLE OF 90			TAND
132	F A	DUPLICATE CHARACTER IN CSOWN CALL	Entry in collating sequence is defined twice.	Check CSOWN call.	CSOWN=
133	F A	IRRECONCILABLE STATUS OPTION	Status option irreconcilable.	Check status option.	OPECAP=
134	F A	STATUS OPTION INCOMPATIBLE WITH OLD FILE	Status option inconsistent with specified file.	Check status option and/or file name.	OPECAP=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
135	F A	FORM CHANGE ON OPENED FILE	File was previously opened for a different processing type. The valid types are formatted, unformatted, buffer I/O, and random I/O.	Check processing type.	OPECAP=
136	F A	BAD RECL VALUE		Correct record length value.	OPECAP=
137	F A	BLANK OPTION ON UNFORMATTED FILE	Blank option applies to formatted I/O only.	Remove blank option.	OPECAP=
138	F A	BAD BUFL VALUE	Negative value, or, for open files, value not equal to the file buffer length. BUFL must be at least 129 words on coded tape, 513 words on binary tape, and 65 words on disk.	For open files, verify that the BUFL value = the buffer length.	OPECAP=
139	F A	BAD OPEN OPTION	Option not allowed.	Check OPEN option.	OPECAP=
140	F A	ERROR DURING FILE CLOSING	Conflicting file attributes. CRM cannot perform close.	Check file attributes.	FORSYS
141	I A	BAD ARGUMENT TO ICHAR	Argument is not of type character, does not have a character length of 1, or is undefined for this installation.	Check argument to ICHAR.	ICCHAR=
142	F A	BAD CLOSE PARAMETER	An option is not allowed in the close.	Check close option.	CLOSE=
143	F A	ACCESS CHANGE ON OPENED FILE	File was previously opened for a different access.	Remove or change ACCESS option of OPEN statement.	OPECAP=
144	I D	xxxx SUBSCRIPT OF ARRAY nnn = yyy, DECLARED LOWER WAS llll, UPPER WAS uuuu.	The xxxx subscript of array nnn has value yyy. This value is outside the range llll to uuuu.	Correct substring limits.	CDL=
145	I D	STARTING CHARACTER POSITION OF xxxx SHOULD BE .GT. ZERO.	Substring reference outside of character string.	Correct substring limits.	CDL=
146	I D	CHARACTER LENGTH OF xxxx SHOULD BE .GT. ZERO.	A negative character length is invalid.	Correct substring limits.	CDL=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
147	I D	NEW CHARACTER LENGTH OF xxxx EXCEEDS OLD LENGTH OF xxxx	New character length must be LE old character length.	Correct substring limits.	CDL=
148	F A	INTERNAL FILE RECORD LENGTH .LE. ZERO	Record length must be positive.	Correct substring limits.	IIFC= OIFC=
149	F A	INTERNAL FILE LCM RECORD EXCEEDS 150 CHARACTERS	Internal file LCM record cannot exceed 150 characters.	Reduce LCM record length or move record to SCM.	IIFC= OIFC=
150	F A	INTERNAL FILE I/O LIST EXCEEDS FILE SIZE.	Format declared on I/O list incompatible with file (variable, substring, or array) format.	Check I/O list.	IIFC= OIFC=
151	F A	DIRECT ACCESS OPEN HAS NO RECL PARAMETER	Record length parameter missing.	Insert the parameter.	OPECAP=
152	I A	REWIND PROHIBITED ON DIRECT FILE -- IGNORED	REWIND used only for sequential files.	Remove REWIND.	REWIND=
153	F A	ARGUMENT TO CSOWN NOT TYPE CHARACTER	Noncharacter argument passed to CSOWN.	Supply a collating sequence string argument.	CSOWN=
154	F A	UNALLOCATED RECORD LENGTH GREATER THAN 150	Explicit open call attempted to make record length greater than 150 for an unallocated record in static mode.	Declare proper record length on PROGRAM statement.	OPECAP=
155	F A	SEQUENTIAL I/O ATTEMPTED ON DIRECT FILE	Sequential I/O commands used on direct access file.	Use the direct access I/O commands.	OUTB= OUTC= OUTF= INPB= INPC= INPF=
156	F A	CODED I/O ATTEMPTED ON xxxx FILE xxxx	Formatted READ or WRITE attempted on a file which was opened for unformatted, buffer, or random I/O.	Self-evident.	IDAB= ODAB=
157	F A	INVALID KEYWORD FOR COLSEQ	Attempt to specify an invalid collating sequence. The valid keywords are ASCII6, DISPLAY, INTBCD, and COBOL6.	Supply a valid keyword.	COLSEQ=
158	F A	OVER 1499 CHARACTERS IN REPEATED CHARACTER STRING	Character string with more than 1499 characters has a repeat factor for list directed input. Probably caused by missing apostrophe.	Add the terminating apostrophe or break the input into substrings.	LDIN=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
159	F A	SCRATCH FILE xxxxx CANNOT BE CLOSED WITH STATUS=KEEP	It is illegal to use status='KEEP' on a CLOSE when status='SCRATCH' was specified on the OPEN.	Correct the CLOSE or OPEN statement.	FORSYS=
160	F A	ILLEGAL USE OF ASTERISK AS STRING DELIMITER IN FORMAT	Asterisk is an invalid format string delimiter in FORTRAN 5.	Use apostrophe.	KODER= KRAKER=
161	F A	NON EXISTENT OVCAP	An attempt to load an OVCAP that does not exist.	Check that the name specified is the name of the first subroutine after an OVCAP statement.	LOVCAP or XOVCAP
162	F A	OVCAP IS ALREADY LOADED	LOVCAP has been called twice for the same OVCAP name.	Check program logic and eliminate redundant call.	LOVCAP or XOVCAP
163	F A	OVCAP WAS NEVER LOADED	A call to UOVCAP has been made specifying an OVCAP that has not been loaded.	Check program logic.	UOVCAP
164	F A	FDL ERROR xx DURING LOAD OR UNLOAD of OVCAP	A Fast Dynamic Loader error has been raised for reasons beyond user control.	Check error number in Loader reference manual. Follow site-defined procedure.	LOVCAP, XOVCAP, or UOVCAP
165	F A	INVALID SEQUENCE	SMKEY call specified a col-seg parameter without specifying a coding identifier of DISPLAY.	Ensure coding identifier is set to DISPLAY.	SMKEY
166	F A	RESERVED COL-SEQ	SMSEQ/SMEQU call specified a sequence name equivalent to one of the standard collating sequence names (ASCII6/COBOL6/DISPLAY/INTBCD) in an attempt to redefine it.	Select another name for the user-supplied collating sequence.	SMSEQ/ SMEQU
167	F A	H, ', ", ILLEGAL INPUT FORMATS	Single quote, double quote, or H format are illegal in FORTRAN 5 input.	Correct format.	KRAKER=
168	F A	* DECIMAL POINT MISSING	Decimal point required.	Supply decimal point.	FMTAP=
169	F A	FORMAT VARIABLE DOES NOT CONTAIN ASSIGNED FORMAT	Assignment of a format to a variable used for I/O is not allowed.	Use ASSIGN statement to assign statement label to variable.	INPC= OUTC= IDAC= ODAC=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
170	F A	ZERO LENGTH HOLLERITH STRING	Hollerith string must have a positive nonzero length.	Make string positive nonzero length.	KODER=
171	F A	BAD FILENAME GIVEN	Illegal character was used or character length was greater than 7.	Supply a valid file name.	OPECAP=
172	F A	SEPARATOR MISSING IN FORMAT	Error in format statement.	Supply appropriate separator.	FMTAP=
173	F A	SCALE FACTOR MISSING IN FORMAT	P edit descriptor requires count.	Supply a valid scale factor.	FMTAP=
174	F A	WTSET ARGUMENT INVALID	The first argument is not an integer or a string of length #1 or a character code is undefined for this installation.	Check the first argument.	WTSET=
175	F A	INVALID CHARACTER CODE IN CSOWN CALL	One of the character codes in the argument is undefined for this installation.	Check the argument.	CSOWN=
176-198 Reserved.					
199	F A	FORM='BUFFERED' NONCOMPATIBLE WITH DIRECT FILE	ACCESS='DIRECT' was specified for a FORM='BUFFERED' unit.	Delete ACCESS='DIRECT' specifier.	OPECAP=
200	F A	DIRECT { CODED } { READ } ON { BINARY } { WRITE } ON NONEXPLICITLY OPENED FILE xxxxx	Direct access file must be opened before I/O is allowed.	Use OPEN command.	ODAB= ODAC= IDAB= IDAC=
201	F A	DIRECT WRITE ATTEMPTED ON SEQUENTIAL FILE xxxxx	Direct I/O command used on sequential file.	Use sequential I/O command.	ODAB=
202	F A	BINARY WRITE ATTEMPTED ON xxxxx FILE xxxxx	Unformatted WRITE attempted on a file opened for formatted, buffer, or random I/O.	Check WRITE statement.	ODAB=
203	F A	ATTEMPT TO WRITE NON-POSITIVE RECORD NUMBER	Record number must be positive.	Check record number.	ODAB= ODAC=
207	F A	ATTEMPT TO READ NON-POSITIVE RECORD NUMBER	Record number must be positive.	Check record number.	IDAB= IDAC=
208	F A	LIST EXCEEDS RECORD LENGTH FOR FILE xxxxx	List too long or record length too short.	Check record list and record length.	ODAB=

TABLE B-5. EXECUTION-TIME DIAGNOSTICS (Contd)

No.	Class	Message	Significance	Action	Issued By
209	F A	CMM ERROR IN CODED DIRECT-ACCESS OUTPUT	Common Memory Manager must be available to handle a record length greater than 1500 for formatted direct access I/O.	Define proper record length on the PROGRAM statement.	ODAC=
212	F A	CMM ERROR IN CODED DIRECT-ACCESS INPUT	Common Memory Manager must be available to handle a record length greater than 1500 for formatted direct access I/O.	See Common Memory Manager reference manual.	IDAC=
215	I A	UNDEFINED WEIGHT PASSED TO CHAR	The character argument (weight) passed is not defined in the collating table.	Check character argument or current collating sequence.	CHAR= CHARF=
216	F A	SUBSTRING ERROR ON NAMELIST ITEM xxxx IN GROUP yyyy	Format of the substring is not correct.	Correct format of the substring.	NAMIN=
217	F A	NAMELIST ITEM xxxx IN GROUP yyyy, ITEM LENGTH	The substring has an upper bound greater than the length of the named character string.	Correct substring limits.	NAMIN=
<p>Note 1 Infinities can be generated by dividing a nonzero number by zero, or by an addition, subtraction, multiplication, or division whose result was greater than 10^{322} in absolute value. Indefinites are usually generated by dividing zero by zero.</p> <p>Note 2 Check for undefined argument; if argument is calculated, check for undefined or illegal operand.</p>					

This glossary does not include terms defined in the ANSI standard for FORTRAN, X3.9-1978.

Advanced Access Methods (AAM) -

A file manager that processes indexed sequential, direct access, and actual key file organizations, and supports the Multiple-Index Processor. See CYBER Record Manager.

Basic Access Methods (BAM) -

A file manager that processes sequential and word addressable file organizations. See CYBER Record Manager.

Beginning-of-Information (BOI) -

CYBER Record Manager defines beginning-of-information as the start of the first user record in a file. System-supplied information, such as an index block, control word, or tape label, exists prior to beginning-of-information.

Blank Common Block -

An unlabeled common block. No data can be stored into a blank common block at load time. The size of the block is determined by the largest declaration for it. Contrast with Labeled Common Block.

Block -

In the context of input/output, a physical grouping of data on a file that provides faster data transfer. CYBER Record Manager defines four block types on sequential files: I, C, K, and E. Other kinds of blocks are defined for indexed sequential, direct access, and actual key files. Also refers to a common block.

Buffer -

An intermediate storage area used to compensate for a difference in rates of data flow, or times of event occurrence, when transmitting data between central memory and an external device during input/output operations.

Buffer Statement -

One of the input/output statements BUFFER IN or BUFFER OUT.

Common Block -

An area of memory that can be declared in a COMMON statement by more than one relocatable program and used for storage of shared data. See Blank Common Block and Labeled Common Block.

CYBER Loader -

The system software facility that loads object code into memory and prepares it for execution.

CYBER Record Manager (CRM) -

A generic term relating to the common products AAM and BAM that run under the NOS and NOS/BE operating systems, and which allow a variety of record types, blocking types, and file organi-

zations to be created and accessed. The execution time input/output of COBOL 5, FORTRAN 5, Sort/Merge 4 and 5, ALGOL 4, and the DMS-170 products is implemented through CRM. Neither the input/output of the NOS and NOS/BE operating systems themselves, nor any of the system utilities such as COPY or SKIPP, is implemented through CRM. All CRM file processing requests ultimately pass through the operating system input/output routines.

Default Type -

The data type assumed by a variable in the absence of any type declarations for the variable. Variables whose names begin with one of the letters A through H or O through Z have a default type of real. Variables whose names begin with one of the letters I through N have a default type of integer.

Direct Access Input/Output -

A method of input/output in which records can be read or written in any order. Direct access input/output is performed by direct access READ and WRITE statements.

End-of-File (EOF) -

A particular kind of boundary on a sequential file, recognized by the END= parameter, the functions EOF and UNIT, and written by the ENDFILE statement. Any of the following conditions is recognized as end-of-file:

End-of-section (for INPUT file only)

End-of-partition

End-of-information (EOI)

W type record with flag bit set and delete bit not set

Tape mark

Trailer label

Embedded zero length level 17 block

End-Of-Information (EOI) -

The end of the last programmer record in a file. Trailer labels are considered to be past end-of-information. End-of-information is undefined for unlabeled S or L tapes.

Entry Point -

A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Equivalence Class -

A group of variables and arrays whose position relative to each other is defined as a result of an EQUIVALENCE statement.

Extended Memory -

Extended memory for the CYBER 170 Model 176 is large central memory (LCM) or large central memory extended (LCME). Extended memory for all other computer systems except for the CYBER 170 800 Series, is extended core storage (ECS) or extended semiconductor memory (ESM). Extended memory for CYBER 170 800 Series is unified extended memory (UEM).

In this manual, the acronym ECS refers to all forms of extended memory unless otherwise noted. However, in the context of a multimainframe environment or distributive data path (DDP) access, model 176 is excluded.

Programming information for the various forms of extended memory can be found in the COMPASS reference manual and in the appropriate computer system hardware reference manual. Descriptions of the various forms of extended memory can be found in the hardware manuals appropriate for equipment available at the site.

External File -

A file residing on an external storage device. An external file starts at beginning-of-information and ends at end-of-information. See File.

External Reference -

A reference in one program unit to an entry point in another program unit.

Field Length -

The area (number of words) in central memory assigned to a job.

File -

A logically related set of information; the largest collection of information that can be addressed by a file name. FORTRAN 5 recognizes two types of files, internal files and external files.

FILE Control Statement -

A control statement that contains parameters used to build the file information table for processing. Basic file characteristics such as organization, record type, and description can be specified on this statement.

File Information Table (FIT) -

A table through which a user program communicates with CYBER Record Manager. All file processing executes on the basis of fields in the table. Some fields can be set by the FORTRAN user in the FILE control statement.

Generic Function Name -

The name of an intrinsic function that can have arguments of any data type. Except for data type conversion generic functions, the data type of the result is the same as the data type of the arguments.

Implicit Type -

The type of a variable as declared in an IMPLICIT statement.

Indefinite -

An indefinite value results from a calculation that cannot be resolved, such as dividing zero by zero. The internal representation of an indefinite value does not correspond to any number; therefore, an I is printed for the value.

On a CYBER 70 Model 71, 72, 73, 74, CYBER 170 Model 171, 172, 173, 174, 175, or 6000 Series computer, no action is taken unless the indefinite value is used as an operand in an expression. In this case, the program aborts and an error message is printed.

On a CYBER Model 76, CYBER 170 Model 176, 835, 855, or 7600, the indefinite flag is set as soon as the indefinite value is generated. This flag causes the program to abort and an error message to be printed.

Infinite -

An infinite value results from a calculation that generates a value that exceeds the upper or lower range of the computer, such as computing $10^{100} * 10^{100}$. An infinite value is printed as the symbol R.

If the result of a calculation exceeds the upper range of the computer, the outcome depends on the computer being used.

On a CYBER 70 Model 71, 72, 73, 74, CYBER 170 Model 171, 172, 173, 174, 175, or 6000 Series computer, no action is taken unless the value is used as an operand in an expression. In this case, the program aborts and an error message is printed.

On a CYBER 70 Model 76, CYBER 170 Model 176, 835, 855, or 7600, the overflow condition flag is set in the Program Status Designator register as soon as the range is exceeded. This flag causes the program to abort and an overflow message to be listed. This condition also results from the use of an operand that was not generated by an arithmetic operation.

If the result of a calculation is less than the lower range, no further action is taken except on the CYBER 70 Model 76, CYBER 170 Model 176, and 7600 Series computers. On these computers, no further action is taken unless underflow has been selected as a mode error by a MODE control statement. In this case, the underflow condition flag is set in the Program Status Designator register as soon as the condition is generated. This flag causes the program to abort and an underflow error message to be printed.

Internal File -

A character variable, array, or substring on which input/output operations are performed by formatted READ and WRITE statements. Internal files provide a method of transferring and converting data from one area of memory to another.

Labeled Common Block -

A common block into which data can be stored at load time. The first program unit declaring a labeled common block determines the amount of memory allocated. Contrast with Blank Common Block.

Logical File Name -

The name by which a file is identified; consists of one through seven letters or digits, the first a letter. File names used in standard FORTRAN 5 input/output statements can have a maximum of six letters or digits.

Main Overlay -

An overlay that must remain in memory throughout execution of an overlaid program.

Mass Storage Input/Output -

The type of input/output used for random access to files; it involves the subroutines OPENMS, READMS, WRITMS, CLOSMS, and STINDX.

Object Code -

Binary code produced by the compiler. Object code must be processed by the CYBER Loader before it can be executed.

Object Listing -

A compiler-generated listing of the object code produced for a program, represented as COMPASS code.

Offset -

The starting position of the array in the first word of its storage (0 to 9).

Optimizing Mode -

One of the compilation modes in the FORTRAN 5 compiler, indicated by the control statement options OPT=0, 1, 2 or 3.

Overlay -

One or more relocatable programs that were relocated and linked together into a single absolute program. It can be a main, primary, or secondary overlay.

Partition -

CYBER Record Manager defines a partition as a division within a file with sequential organization. Generally, a partition contains several records or sections. Implementation of a partition boundary is affected by file structure and residence. Partition boundaries are shown in table C-1.

Notice that in a file with W type records a short PRU of level 0 terminates both a section and a partition.

TABLE C-1. PARTITION AND SECTION BOUNDARIES

Device	Record Type (RT)	Block Type (BT)	Physical Representation of Partition	Physical Representation of Section
PRU device†	W	I	A short PRU of level 0 containing a one-word deleted record pointing back to the last I block boundary, followed by a control word with a flag indicating a partition boundary.	A deleted one-word record pointing back to the last I block boundary followed by a control word with flags indicating a section boundary. At least the control word is in a short PRU of level 0.
	W	C	A short PRU of level 0 containing a control word with a flag indicating a partition boundary.	A control word with flags indicating a section boundary. The control word is in a short PRU of level 0.
	D,F,R,T,U,Z	C	A short PRU of level 0 followed by a zero-length PRU of level 17 octal.	A short PRU with a level less than 17 octal.
	S	-	A zero-length PRU of level number 17 octal.	Undefined.
S or L format tape	W	I	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with a flag indicating a partition boundary.	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with flags indicating a section boundary.
	W	C	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with a flag indicating a partition boundary.	A separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a control word with flags indicating a section boundary.
	D,F,T,R,U,Z	C,K,E	A tapemark.	Undefined.
	S	-	A tapemark.	Undefined.
Any other tape format	-	-	Undefined.	Undefined.
†NOS and NOS/BE only.				

Pass by Name -

A method of referencing a subprogram in which the addresses of the actual arguments are passed.

Pass by Value -

A method of referencing a subprogram in which only the values of the actual arguments are passed.

Primary Overlay -

A second level overlay that is subordinate to the main overlay. A primary overlay can call its associated secondary overlays and can reference entry points and common blocks in the main overlay.

Procedure -

A FORTRAN function subprogram, subroutine, statement function, or intrinsic function.

Program Unit -

A sequence of FORTRAN statements terminated by an END statement. The FORTRAN program units are main programs, subroutines, functions, and block data subprograms.

PRU -

Under NOS and NOS/BE, the amount of information transmitted by a single physical operation of a specified device. The size of a PRU depends on the device: a PRU which is not full of user data is called a short PRU; a PRU that has a level terminator, but not user data, is called a zero-length PRU. PRU sizes are shown in table C-2.

TABLE C-2. PRU SIZES

Device	Size in Number of 60-Bit Words
Mass storage (NOS and NOS/BE only).	64
Tape in SI format with coded data (NOS/BE only).	128
Tape in SI format with binary data.	512
Tape in I format (NOS only).	512
Tape in any other format.	Undefined.

PRU Device -

A mass storage device or a tape in SI (NOS and NOS/BE), I (NOS and NOS/BE), or X (NOS/BE only) format, so called because records on these devices are written in PRUs.

Record -

CYBER Record Manager defines a record as a group of related characters. A record or a portion thereof is the smallest collection of information passed between CYBER Record Manager and a user program in a single read or write operation. Eight different record types exist, as defined by the RT field of the file information table.

Other parts of the operating systems and their products might have additional or different definition of records.

Record Length -

The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Record Type -

The term record type can have one of several meanings, depending on the context. CYBER Record Manager defines eight record types established by an RT field in the file information table.

Reference Listing -

A part of listing produced by a FORTRAN compilation, which displays some or all of the entities used by the program, and provides other information such as attributes and location of these entities.

Relocation -

Placement of object code into central memory in locations that are not predetermined, and adjusting the addresses accordingly.

SCOPE 2 Record Manager -

The record manager used under the SCOPE 2 operating system. It processes all files read and written as a result of user requests at execution time, as well as all files read and written at compile time by the compiler. The SCOPE 2 Record Manager processes all input/output files.

Secondary Overlay -

The third level of overlays. A secondary overlay is called into memory by its associated primary overlay. A secondary overlay can reference entry points and common blocks in both its associated primary overlay and the main overlay.

Section -

CYBER Record Manager defines a section as a division within a file with sequential organization. Generally, a section contains more than one record and is a division within a partition of a file. A section terminates with a physical representation of a section boundary. Section boundaries are described in table C-1.

The NOS and NOS/BE operating systems equate a section with a system-logical-record of level 0 through 16 octal.

Sequential -

A file organization in which the location of each record is defined only as occurring immediately after the preceding record. A file position is defined at all times, which specifies the next record to be read or written.

Sequential Access Input/Output -

A method of input/output in which records are processed in the order in which they occur on a storage device.

Source Code -

Code written by the programmer in a language such as FORTRAN, and input to a compiler.

Source Listing -

A compiler-produced listing, in a particular format, of the user's original source program.

Specific Function Name -

The name of an intrinsic function that accepts arguments of a particular data type, and returns a result of a particular data type.

System-Logical-Record -

Under NOS/BE, a data grouping that consists of one or more PRUs terminated by a short PRU or zero-length PRU. These records can be transferred between devices without loss of structure.

Unit Specifier -

An integer constant, or an integer variable with a value of either 0 to 999, or an L format logical file name. In input/output statements, it indicates on which unit the operation is to be performed. It is linked with the actual file name by the PROGRAM statement or OPEN statement.

Word Addressable -

A file organization in which the location of each record is defined by the ordinal of the first word in the record, relative to the beginning of the file.

Working Storage Area -

An area within the user's field length, intended for receipt of data from a file or transmission of data to a file. Transmission to or from a buffer intervenes, except for buffer statements.

Zero-Byte Terminator -


12 bits of zero in the low order position of a word that marks the end of the line to be displayed at a terminal or printed on a line printer. The image of cards input through the card reader or terminal also has such a terminator.

The following symbols are used in the descriptions of the FORTRAN 5 statements:

- v variable name, array name, or array element
- sl statement label
- iv integer variable
- name symbolic name
- u input/output unit specifier, which can be an integer expression with a value of 0 through 999, or a Boolean expression containing a display code file name in L format
- fs format specifier
- iolist input/output list
- ios input/output status specifier
- recn record number

Other symbols are defined individually in the statement descriptions.

ASSIGNMENT

	<u>Page</u>
v = arithmetic expression	3-1
Boolean v = Boolean expression	3-4
character v = character expression	3-5
logical v = logical or relational expression	3-5
 	 3-10

TYPE DECLARATION

INTEGER v[,v]...	2-12
REAL v[,v]...	2-12
DOUBLE PRECISION v[,v]...	2-12
COMPLEX v[,v]...	2-12
BOOLEAN v[,v]...	2-13
LOGICAL v[,v]..	2-12
CHARACTER [*length][,]v[*length][,v[*length]]...	2-13
IMPLICIT type(ac[,ac]...)[,type(ac[,ac]...)]...	2-7
ac Is a single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.	

EXTERNAL DECLARATION

EXTERNAL name[,name]... 2-7

INTRINSIC DECLARATION

INTRINSIC name[,name]... 2-8

STORAGE ALLOCATION

type array(d)[,array(d)]... 2-5

DIMENSION array(d)[,array(d)]... 2-5

type Is INTEGER, CHARACTER, **DOUBLE PRECISION**, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

d Is one through seven array bound expressions separated by commas, as described in section 2.

COMMON [/[name]/]nlist[[,]/[name]/nlist]... 2-1

nlist Is a list of variables or arrays, separated by commas, to be included in the common block.

DATA nlist/clist/[[,]nlist/clist/]... 2-3

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

variable

array

element

substring

implied DO list

clist Is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in section 2.

EQUIVALENCE (nlist)[,(nlist)]... 2-6

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

LEVEL n_name[,name]... 2-9

n Is an unsigned integer constant, or symbolic constant, with the value 0, 1, 2, or 3 indicating the storage level.

PARAMETER (name=exp[,name=exp]...) 2-9

exp Is a constant or constant expression.

SAVE [name[,name]...] 2-10

FLOW CONTROL

GO TO s1 4-1

GO TO (s1[,s1]...)[,]expression 4-1

GO TO iv[[,](s1[,s1]...)] 4-2

ASSIGN s1 TO iv 4-2

IF (arithmetic or Boolean expression) s11,s12,s13	4-3
IF (logical expression) statement	4-3
IF (logical expression) THEN	4-3
ELSE IF (logical expression) THEN	4-4
ELSE	4-4
END IF	4-4
DO s1[,]v=e1,e2[,e3]	4-6
e1,e2,e3	Are indexing parameters. They can be integer, real, double precision, or Boolean constants , symbolic constants, variables, or expressions.
PAUSE [n]	4-9
STOP [n]	4-10
n	Is a string of 1 through 5 digits, or a character constant.
END	4-10

MAIN PROGRAM

PROGRAM name(**fpar[,fpar]...**) 6-3

fpar Is a file declaration in one of the following forms:

```

filename
filename=buffer length
filename=/record length
filename=buffer length/record length
alternate name=filename

```

SUBPROGRAM

SUBROUTINE name([argument[,argument]...]) 6-4

[type]FUNCTION name([argument[,argument]]...) 6-5

 type Is **BOOLEAN**, CHARACTER, INTEGER, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

BLOCK DATA[name] 6-13

STATEMENT FUNCTION

name([argument[,argument]]...)=expression 6-8

SUBROUTINE CALL

CALL name([argument[,argument]...]) 6-5

FUNCTION REFERENCE

name([argument[,argument]]...) 6-6

ENTRY POINT

ENTRY name[(argument[,argument]...)]

6-6

RETURN

RETURN [expression]

6-7

FORMATTED INPUT/OUTPUT

READ ({ u,fn
u,FMT=fn
UNIT=u,FMT=fn } [,IOSTAT=ios][,ERR=s1][,END=s1]) [iolist]

5-25

READ fn[,iolist]

5-25

WRITE ({ u,fn
u,FMT=fn
UNIT=u,FMT=fn } [,IOSTAT=ios][,ERR=s1]) [iolist]

5-26

PRINT fn[,iolist]

5-26

████████████████████

5-26

UNFORMATTED INPUT/OUTPUT

READ ([UNIT=]u[,IOSTAT=ios][,ERR=s1][,END=s1]) [iolist]

5-27

WRITE ([UNIT=]u[,IOSTAT=ios][,ERR=s1]) [iolist]

5-27

LIST DIRECTED INPUT/OUTPUT

READ ({ u,*
u,FMT=*
UNIT=u, FMT=* } [,IOSTAT=ios][,ERR=s1][,END=s1]) [iolist]

5-27

READ *[,iolist]

5-27

WRITE ({ u,*
u,FMT=*
UNIT=u,FMT=* } [,IOSTAT=ios][,ERR=s1]) [io-list]

5-29

PRINT *[,iolist]

5-29

████████████████████

5-29

DIRECT ACCESS INPUT/OUTPUT

READ ({ u,fn
u,FMT=fn
UNIT=u,FMT=fn } [,IOSTAT=ios][,ERR=s1][,REC=recn]) [iolist]

5-1

WRITE ({ u,fn
u,FMT=fn
UNIT=u,FMT=fn } [,IOSTAT=ios][,ERR=s1][,REC=recn]) [iolist]

5-1

NAMelist INPUT/OUTPUT

NAMelist /name/v[,v]...[/name/v[,v]...]....

5-30

READ ({ u, name
 u, FMT=name
 UNIT=u, FMT=name } [, IOSTAT=ios][, ERR=s1][, END=s1])

5-31

READ name

5-31

WRITE ({ u, name
 u, FMT=name
 UNIT=u, FMT=name } [, IOSTAT=ios][, ERR=s1])

5-32

PRINT name

5-32

PUNCH name

5-32

name Is a NAMelist group name.

BUFFER INPUT/OUTPUT

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

5-35

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

5-35

p Is an integer constant or variable:

 zero - even parity

 nonzero - odd parity

a Is the first word of the data block to be transferred.

b Is the last word of the data block to be transferred.

INTERNAL DATA TRANSFER

ENCODE (c,fs,v)lolist

5-42

DECODE (c,fs,v)lolist

5-42

v Is the starting location of the record to be transferred.

c Specifies the number of characters to be transferred to or from each record.

FORMAT SPECIFICATION

s1 FORMAT (flist)

5-5

flist Is a list of items, separated by commas, having the following forms:

[r]ed
ned
[r](flist)

ed Is a repeatable edit descriptor.

ned Is a nonrepeatable edit descriptor.

r Is a nonzero unsigned integer constant repeat specification.

EDIT DESCRIPTORS

srEw.d	Single precision floating-point with exponent.	5-11
srEw.dEe	Single precision floating-point with specified exponent length.	5-11
srFw.d	Single precision floating-point without exponent.	5-13
srDw.d	Double precision floating-point with exponent.	5-9
srGw.d	Single precision floating-point with or without exponent.	5-15
srGw.dEe	Single precision floating-point with or without specified exponent length.	5-15
rIw	Decimal integer.	5-15
rIw.m	Decimal integer with specified minimum number of digits.	5-15
rLw	Logical.	5-17
rA	Character with variable length.	5-7
rAw	Character with specified length.	5-7
rRw	Rightmost characters with binary zero fill.	5-18
rOw	Octal.	5-17
rOw.m	Octal with minimum digits and leading zeros.	5-17
rXw	Hexadecimal.	5-21
rXw.m	Hexadecimal with minimum digits and leading zeros.	5-21
BN	Blanks ignored on numeric input.	5-9
BZ	Blanks treated as zeros on numeric input.	5-9
SP	+ characters produced on output.	5-19
SS	+ characters suppressed on output.	5-19
S	+ characters suppressed on output.	5-19
nX	Skip n spaces.	5-21
Tn	Tabulate to n th column.	5-19
TRn	Tabulate forward.	5-19
TLn	Tabulate backward.	5-19
nH	Hollerith or character string output.	5-15
'...'	Hollerith or character string output.	5-8
:	Format control.	5-21
/	End of FORTRAN record.	5-13
s	Is an optional scale factor of the form kP.	
r	Is an optional repetition factor.	
w	Is an integer constant indicating field width.	
d	Is an integer constant indicating digits to right of decimal point.	
e	Is an integer constant indicating digits in exponent field.	
m	is an integer constant indicating minimum number of digits in field.	
n	is a positive nonzero decimal digit.	

FILE POSITIONING

BACKSPACE ([UNIT= <i>u</i>][,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>])	5-44
BACKSPACE <i>u</i>	5-44
REWIND ([UNIT= <i>u</i>][,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>])	5-43
REWIND <i>u</i>	5-43
ENDFILE ([UNIT= <i>u</i>][,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>])	5-44
ENDFILE <i>u</i>	5-44

FILE STATUS

OPEN ([UNIT= <i>u</i>][,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>][,FILE= <i>fin</i>][,STATUS= <i>sta</i>][,ACCESS= <i>acc</i>][,FORM= <i>fm</i>][,RECL= <i>r1</i>][,BLANK= <i>blnk</i>][,BUFL= <i>bl</i>])	5-36
INQUIRE ({ [UNIT= <i>u</i>] } [,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>][,EXIST= <i>ex</i>][,OPENED= <i>od</i>][,NUMBER= <i>num</i>][,NAMED= <i>nmd</i>][,NAME= <i>fn</i>][,ACCESS= <i>acc</i>][,SEQUENTIAL= <i>seq</i>][,DIRECT= <i>dir</i>][,FORM= <i>fm</i>][,FORMATTED= <i>fmt</i>][,UNFORMATTED= <i>unf</i>][,RECL= <i>fcl</i>][,NEXTREC= <i>nr</i>][,BLANK= <i>blnk</i>])	5-39
CLOSE ([UNIT= <i>u</i>][,IOSTAT= <i>ios</i>][,ERR= <i>s1</i>][,STATUS= <i>sta</i>])	5-38

OVERLAYS AND OVCAPS

OVERLAY ([<i>fname</i> , <i>i</i> , <i>j</i>][,orig][,OV= <i>n</i>])	9-4
<i>fname</i> Is the name of the file on which the overlay is to be written.	
<i>i</i> , <i>j</i> Are the overlay level numbers.	
<i>orig</i> Specifies the origin of the overlay.	
<i>n</i> Optional specification of number of higher level overlays.	
CALL OVERLAY ([<i>fname</i> , <i>i</i> , <i>j</i>][,recall][,k])	9-4
<i>recall</i> Is the recall parameter.	(Also see CYBER Loader reference manual)
<i>k</i> Indicates location of <i>fname</i> .	
OVCAP	9-5
CALL OVCAP(<i>name</i>)	9-5
<i>name</i> Is a variable or constant of type character.	
CALL OVCAP(<i>name</i> , <i>opt</i>)	9-6
<i>name</i> Is a variable or constant of type character.	
<i>opt</i> Is the <i>opt</i> parameter. It is a list of parameters passed to the subroutine.	
OVCAP(<i>name</i>)	9-6

A C\$ directive is a special form of comment line that controls compiler processing. A particular C\$ directive affects an aspect of the compiler's interpretation of those lines following the directive and preceding either a subsequent directive modifying the same aspect, if such a directive appears, or the end of the program unit. The effects of a C\$ directive do not carry over to subsequent program units. The aspects of interpretation that can be controlled are:

Listing of the program and associated compiler-produced information, called listing control

Specification of program lines to be processed or ignored, called conditional compilation

Character data comparison collation table, called collation control

Minimum trip count and long trip count for DO loops, called DO loop control

The general form of a C\$ directive is shown in figure E-1.

C\$ keyword[[p=[c],[p=[c]] ...]][[,]lab]	
keyword	Is one of LIST, IF, ELSE, ENDF, COLLATE, or DO. The keyword can begin in any column starting with column 7. In sequenced mode the keyword can begin in any column following the character \$.
p	Is a parameter. Depending upon the keyword that appears, one or more parameters can be specified.
c	Is an integer constant, or symbolic name of an integer constant, with a value of zero or one. Depending upon the parameter p, the constant either is optional or must not appear.
lab	Is a label. Depending upon the keyword that appears, a label may be specified. If a label appears and no parameters are present, a comma must separate the keyword and the label.

Figure E-1. C\$ Directive

The letter C in column 1 together with the character \$ in column 2 identify a line as a C\$ directive line. Such a line will be interpreted as a comment only if the directive suppression (DS) option is specified on the FTN5 control statement. The entire directive must appear on a single line. A C\$ directive interrupts statement continuation.

In sequenced mode the letter C in the column immediately to the right of the sequence number together with the character \$ immediately to the right of the C identify a C\$ directive line. A line with no sequence number in sequence mode cannot be a C\$ directive.

LISTING CONTROL

A listing control directive has the keyword LIST. It must have the form shown in figure E-2.

C\$ LIST(p=[c] [,p=[c]] ...)	
p	Is S, O, R, A, M or ALL.
c	Is a constant or the symbolic name of a constant.

Figure E-2. Listing Control Directive

The constant is optional for all parameters; its absence is equivalent to the appearance of a constant with the value 1.

The listing control directive modifies the state of any initially enabled list option switches. A list option switch is initially enabled when the corresponding list option is requested on the FTN5 control statement. Any attempt to modify a list option switch that was not initially enabled is ignored: p=0 disables switch p; p=1 enables switch p.

ALL=c is equivalent to S=c, O=c, R=c, A=c, M=c.

A listing control directive found by the compiler to be in error results in a warning diagnostic.

The list option switches offer the following control:

- S Source lines are listed when enabled.
- O Generated object code is listed for statements processed when enabled.
- R Symbol references are accumulated for the cross-reference list when enabled. Symbols with no accumulated references will not appear in that list; no accumulation for an entire program unit suppresses cross-reference list.
- A The symbol attribute list is generated if this switch is enabled when the END statement is processed.
- M The symbol map list is generated if this switch is enabled when the END statement is processed.

An example of listing control directives is shown in figure E-3. The complete output listing is shown. All source statements appearing between C\$ LIST (S=0) and C\$ LIST (S=1) are suppressed in the output listing. (Source statement lines with errors are listed on the ERROR file along with diagnostics.) The C\$ LIST (ALL=0) directive, active when the END statement is encountered, suppresses the reference map.

CONDITIONAL COMPILATION

A conditional compilation directive has a keyword which is one of IF, ELSE, or ENDIF. Such a directive controls whether the lines immediately following the directive are to be processed or ignored by the compiler.

The conditional compilation directives are divided into three categories:

An IF directive with the keyword IF

An ELSE directive with keyword ELSE

An ENDIF directive with keyword ENDIF

The IF directive, ELSE directive, and ENDIF directive are shown in figures E-4, E-5, and E-6, respectively.

C\$ IF(e)[[,] lab]

- e Is a logical constant expression. If a symbolic constant appears, it must have been previously defined in a PARAMETER statement in the program containing the IF directive.
- lab Is an optional label. It must be a symbolic name. Its use as a label in a conditional compilation directive does not affect, and is not affected by, its use for any other purpose in the program unit.

Figure E-4. IF Directive

C\$ ELSE[,lab]

- lab Is as for an IF directive.

Figure E-5. ENDIF Directive

C\$ ENDIF[,lab]

- lab Is as for an IF directive.

Figure E-6. ELSE Directive

Program containing listing control directive:

```

FTN5,LO=S/A/R/M.
7/8/9 in column 1
      PROGRAM P
C PROGRAM TO TEST LISTING CONTROL DIRECTIVES
C$ LIST(S=0)
      DIMENSION A(10)
C THE FOLLOWING CARD CONTAINS AN ERROR
      INTEGER B/C
C$ LIST(S=1)
      DO 100 I=1,10
100  A(I) = 0.0
C$ LIST(ALL=0)
      END
6/7/8/9 in column 1

```

Compiler output listing:

```

1          PROGRAM P
2          C PROGRAM TO TEST LISTING CONTROL DIRECTIVES
3          C$ LIST(S=0)
6          INTEGER B/C
FATAL *    EXPECTED COMMA -- FOUND /
7          C$ LIST(S=1)
8          DO 100 I=1,10
9          100  A(I) = 0.0
10         C$ LIST(ALL=0)

1 FATAL   ERROR IN P

```

Figure E-3. Listing Control Directive Example

For each IF directive there must appear exactly one ENDIF directive later in the same program unit, and for each ENDIF directive there must appear exactly one IF directive earlier in the same program unit. Between an IF directive and its corresponding ENDIF directive will appear zero or more lines called a conditional sequence. A conditional sequence can optionally contain one ELSE directive corresponding to the IF directive and ENDIF directive delimiting the conditional sequence. An ELSE directive can appear only within a conditional sequence. A conditional sequence can not contain more than one ELSE directive unless it contains another conditional sequence. If an ELSE directive is contained within more than one conditional sequence, the ELSE directive corresponds to that IF-ENDIF pair which delimits the smallest, that is, innermost, conditional sequence containing the ELSE directive.

If two (or three) corresponding conditional directives have a label, it must be the same label. No other restriction applies to labels on conditional directives. There is no requirement that any conditional directive have a label. The same label can be used on more than one sequence of corresponding conditional directives in a single program unit, including the case of conditional directives whose conditional sequence contains other conditional directives with the same label.

A conditional sequence can contain any number of properly corresponding conditional directives, and therefore other conditional sequences. If two conditional sequences contain the same line, one conditional sequence must lie wholly within the other conditional sequence.

A conditional compilation directive found by the compiler to be in error results in a diagnostic message.

If an IF directive is processed by the compiler and the logical expression is true, following lines are processed as if the IF directive had not appeared, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are ignored by the compiler. If an IF directive is processed by the compiler and the logical expression is false, the following lines are ignored until the corresponding ENDIF directive is encountered, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are processed.

An example of conditional compilation directives is shown in figure E-7. The sample program contains two DO loops. Conditional compilation directives are included to test the value of the symbolic constant M. If M is 0, the first loop is ignored and the second loop is compiled. If M is 1, the first loop is compiled and the second loop is ignored. The program is compiled and executed two times; once with the statement PARAMETER (M=0) and once with the statement PARAMETER (M=1).

COLLATION CONTROL

A collation control directive has the form shown in figure E-8.

Example 1:

```
PROGRAM B
PARAMETER (M=1)
DIMENSION A(10)
DATA A/10*0.0/
C$ IF(M .EQ. 0)
DO 8 I=1,10
8 A(I) = A(I) + 1.0
C$ ELSE
DO 12 I=1,10
12 A(I) = A(I) - 1.0
C$ ENDF
PRINT*, ' A = ', A
STOP
END
```

Output:

```
A = -1. -1. -1. -1. -1. -1. -1. -1.
    -1. -1.
```

Example 2:

```
PROGRAM B
PARAMETER (M=0)
DIMENSION A(10)
DATA A/10*0.0/
C$ IF(M .EQ. 0)
DO 8 I=1,10
8 A(I) = A(I) + 1.0
C$ ELSE
DO 12 I=1,10
12 A(I) = A(I) - 1.0
C$ ENDF
PRINT*, ' A = ', A
STOP
END
```

Output:

```
A = 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
```

Figure E-7. Conditional Compilation Example

```
C$ COLLATE(p)
```

p is FIXED or USER.

Figure E-8. Collation Control Directive

The collation control directive specifies whether collation of character relational expressions is directed by the fixed (display code) weight table or by the user-specified (ASCII6) weight table.

A collation control directive directs the interpretation of character relational expressions in the lines following the directive and preceding either another collation control directive or the END statement of the program unit. In the case of a character relational expression in a statement function statement, the collation that applies is that in effect for the line or lines containing a reference to the statement function. Consider the example in figure E-9.

```

PROGRAM P
LOGICAL LSF
CHARACTER*5, X, Y, S, T
C$ COLLATE(USER)

LSF(X,Y) = X.LT.Y
.
.
.
C$ COLLATE(FIXED)
IF (LSF(S,T)) A=1.0
.
.
.
END

```

Figure E-9. Collation Control Directive Example

The reference LSF(S,T) results in an evaluation of the character relational expression S.LT.T with the collation that of the fixed weight table.

A collation control directive found by the compiler to be in error results in a diagnostic.

DO LOOP CONTROL

A DO loop control directive has the form shown in Figure E-10.

The constant is optional for both parameters; its absence is equivalent to the appearance of a constant with the value 1.

```

C$ DO(p=c1,p=c2)

p is either OT or LONG.
c is a constant or the symbolic name of a constant.

```

Figure E-10. DO Loop Control Directive

The DO loop control directive modifies the state of one or both DO loop switches. Each switch is initially set (or reset) when the corresponding DO option (appendix E) is (or is not) requested at compile time. A DO loop control directive switch selection overrides the corresponding DO option request.

The OT parameter controls the minimum trip count. If OT is set (OT=1), the minimum trip count for DO loops is one. If OT is not set (OT=0), the minimum trip count for DO loops is zero.

The LONG parameter controls the maximum trip count. If LONG is set (LONG=1) the trip count can exceed 131071. If LONG is not set (LONG=0), the trip count cannot exceed 131071.

A DO loop control directive affects the interpretation of only those DO loops whose DO statements follow the directive in the same program unit.

A DO loop control directive found by the compiler to be in error results in a diagnostic.

This section describes the structure of files read and written by FORTRAN 5. All files read and written as a result of user requests at execution time are processed through Record Manager. The files read and written at compile time by the compiler itself (including source input, coded output, and binary output) are processed by operating system routines when compilation is under NOS or NOS/BE, and by SCOPE 2 Record Manager when compilation is under SCOPE 2.

EXECUTION-TIME INPUT/OUTPUT

All input and output between a file referenced in a program and the file storage device is under control of Record Manager. The version of Record Manager used depends on the operating system.

NOS and NOS/BE use CYBER Record Manager Basic Access Methods (BAM), encompassing sequential and word addressable file organizations, for standard input/output statements, and CYBER Record Manager Advanced Access Methods (AAM) for indexed sequential, direct access, and actual key file organizations, and multiple-index capability, through the CYBER Record Manager interface routines.

SCOPE 2 uses the SCOPE 2 Record Manager for all input/output.

CYBER Record Manager can be called directly, as described in section 8, to use the extended file structure and processing available. SCOPE 2 Record Manager cannot be called directly from the FORTRAN 5 compiler. This appendix deals only with Record Manager processing that results from standard language use.

File processing is governed by values compiled into the file information table (FIT) for each file. If a file or its FIT is changed by other than standard FORTRAN input/output statements, subsequent FORTRAN input/output to that file may not function correctly. Thus, it is recommended that the user not try to use both standard FORTRAN and nonstandard input/output on the same file within a program.

FILE AND RECORD DEFINITIONS

A file is a collection of records referenced by its logical file name. It begins at beginning-of-information and ends with end-of-information. A record is data created or processed by:

One execution of an unformatted READ or WRITE

One card image or a print line defined within a formatted, list directed, or namelist READ or WRITE

One call to READMS or WRITMS

One execution of BUFFER IN or BUFFER OUT

On storage, a file can have records in one of eight formats (record types) defined to Record Manager. Only four of these are part of standard processing:

Z Record is terminated by a 12-bit zero byte in the low order byte position of a 60-bit word.

W Record length is contained in a control word prefixed to the record by Record Manager.

U Record length is defined by the user.

S System logical record.

The remaining types can be formatted within a program under user control and written to a device using a WRITE statement if the FILE control statement is used to specify another record type. Similarly, these types can be read by a READ statement.

The user is responsible for supplying record length information appropriate to each type before a write and for determining record end for a read. For example, a D type record requires a field within the record to specify record length, and F type records require that the user READ/WRITE exactly FL characters in each record.

Unformatted READ and WRITE are implemented through the GETP and PUTP macros of Record Manager; consequently, record operations must conform to macro restrictions. Specifically, RT=R and RT=Z cannot be specified for unformatted operations.

Direct access I/O must be done with RT=U. RT=U is the default.

STRUCTURE OF INPUT/OUTPUT FILES

FORTRAN 5 sets certain values in the file information table depending on the nature of the input/output operation and its associated file structure. Table F-1 lists these values for their respective FIT fields; all except those marked with an asterisk (*) can be overridden at execution time by a FILE control statement. (Numbers in parentheses refer to notes listed following the table.)

TABLE F-1. DEFAULTS FOR FIT FIELDS

FIT Fields		Formatted, NAMELIST, and List Directed Sequential READ/WRITE	Unformatted Sequential READ/WRITE	BUFFER IN/ BUFFER OUT	Mass Storage Input/Output	Direct Access I/O Formatted and Unformatted
Meaning	Mnemonic					
CIO buffer size (words)	(1) BFS [†]	(1)	(1)	(1)	(1)	(1)
Buffer Below Highest Address	BBH	0	0	n/a	0	0
Block type	BT	C [†] /(9) ^{††}	I [†] /(9) ^{††}	C [†] /(9) ^{††}	n/a	C*
Close flag (positioning of file after close)	CF	N*	N*	N*	N* [†] /R* ^{††}	N*
Length in characters of record trailer count field (T type records only)	CL	0	0	0	n/a	n/a
Conversion mode	CM	YES [†] /NO ^{††}	NO	(2)	n/a	n/a
Beginning character position of trailer count field, numbered from zero (T type records only)	CP	0	0	0	n/a	n/a
Length field (D type records) or trailer count field (T type records) is binary	C1 [†]	NO	NO	NO	n/a	n/a
Type of information to be listed in dayfile	DFC [†]	3	3	3	3	3
Type of information to be listed on error file	EFC [†]	0	0	0	0	0
Error options	EO	AD	AD	AD	AD	AD
Trivial error limit	ERL	0	0	0	0	0
Fatal Flush	FF [†]	0	0	n/a	0	0
Length in characters of an F or Z type record (same as MRL)	FL [†]	150(5)*	n/a	n/a	n/a	n/a
File organization	FO	SQ *	SQ *	SQ *	WA *	WA *
Character length of fixed header for T type records	HL	0	0	0	n/a	n/a
Length of user's label area (number of characters)	(7) LBL	0 *	0 *	0 *	n/a	n/a
Logical file name	LFN	(3)	(3)	(3)	(3)	(3)
Length in characters of record length field (D type records)	LL	0	0	0	n/a	n/a

TABLE F-1. DEFAULTS FOR FIT FIELDS (Contd)

FIT Fields		Formatted, NAMELIST, and List Directed Sequential READ/WRITE	Unformatted Sequential READ/WRITE	BUFFER IN/ BUFFER OUT	Mass Storage Input/Output	Direct Access I/O Formatted and Unformatted
Meaning	Mnemonic					
Beginning character position of record length, numbered from zero (D type records)	LP	0	0	0	n/a	n/a
Label type	(7) LT	ANY	ANY	ANY	n/a	n/a
Maximum block length in characters	MBL	0	0	0	n/a	n/a
Minimum block length in characters	MNB†	0	0	0	n/a	n/a
Minimum record length in characters	MNR†	0	0	0	n/a	n/a
Maximum record length in characters	(5) MRL	n/a	223-1	(8) *	n/a	n/a
Multiple of characters per K, E type block	MUL†	2	2	2	n/a	n/a
Open flag (positioning of file after open)	(7) OF	N*	N*	N*	N*†/R††*	N*
Padding character for K, E type blocks	PC†	76B	76B	76B	n/a	n/a
Processing direction	PD	IO	IO	IO	IO	IO
Number of records per K type block	RB	1	1	1	n/a	n/a
Record mark character (R records)	RMK	62B	n/a	62B	n/a	n/a
Record type	RT	Z†/W††(10)	W(6)	S†/W††	U	U *
Length field (D type records) or trailer count field (T type records) has sign overpunch	SB†	NO	NO	NO	n/a	n/a
Suppress buffering	SBF†	NO*	NO*	YES(11)	NO*	NO*
Suppress read ahead	SPR	NO	NO	NO	n/a	n/a
Character length of trailer portion of T type records	TL	0	0	0	n/a	n/a
User label processing	(7) ULP	NO	NO	NO	NO	n/a
End of volume flag (positioning of file at volume CLOSEM time)	VF	U	U	U	U	U

TABLE F-1. DEFAULTS FOR FIT FIELDS (Contd)

Notes:	
n/a	FIT field not applicable to this input/output mode.
*	Default cannot be overridden by a FILE control statement.
(1)	Buffer size can be declared on the PROGRAM statement, OPEN statement, or FILE control statement. Otherwise, CRM chooses the buffer size according to device type. Buffer is allocated on the first I/O operation and deallocated when the file is closed.
(2)	Set by parity designator in BUFFER IN or BUFFER OUT statement.
(3)	Set by PROGRAM statement, OPEN statement, or execution control statement.
(4)	Set by CYBER Record Manager.
(5)	Default can be changed on PROGRAM or OPEN statement. For formatted, NAMELIST, and list directed READ/WRITE statements, a FILE control statement can decrease but not increase the maximum record length declared on the PROGRAM statement. This restriction applies to programs run in static mode.
(6)	Default can be overridden by a FILE control statement only if RT≠R and RT≠Z. For RT=F, FL must be a multiple of 10.
(7)	The LABEL subroutine (section 7) sets LBL=80, LT=ST, OF=R, and ULP=F.
(8)	Maximum record length equal to length of record specified in BUFFER IN or BUFFER OUT statement.
(9)	Unblocked if mass storage file; I if tape file.
(10)	Default can be overridden by FILE control statement only if RT≠U.
(11)	On a CYBER 170 Model 176, SBF must be set to NO on a FILE control statement if a level 2 or 3(LCM) variable is used in a buffer statement.
†Applies to NOS and NOS/BE only.	
††Applies to SCOPE 2 only.	

Sequential Files

The following information is valid, unless the FIT field is overridden by a FILE control statement.

With READ and WRITE statements, the record type (RT) depends on whether the access is formatted or unformatted (applies only to NOS and NOS/BE). A formatted WRITE produces RT=Z records, with each record terminated by a system-supplied zero byte in the low order bits of the last word in the record. An unformatted WRITE produces RT=W records, in which each record is prefixed by a system-supplied control word. Blocking is type C for formatted and I for unformatted records. The files named INPUT, OUTPUT, and PUNCH always have record type Z and block type C. These files should only be processed by formatted, list directed, and namelist input/output statements.

On SCOPE 2 only with READ and WRITE statements, the record type is W for all file types; blocking is I for tape files, and unblocked for all other files.

PRINT and PUNCH statements produce Z type records with C type blocks or on SCOPE 2 only, W type records unblocked for processing on unit record equipment.

BUFFER IN and BUFFER OUT assume S type records or, on SCOPE 2 only, W type records. Formatting is determined by the parity designator in each BUFFER statement. An unformatted operation does not convert character codes during tape reading or writing (CM=NO), while a formatted operation does.

The ENDFILE statement writes a boundary condition known as an end-of-partition. When this boundary is encountered during a read, the EOF function returns end-of-file status. An end-of-partition may not necessarily coincide with end-of-information, however, and reading can continue on the same file until end-of-information on the file has been encountered.

End-of-partition is written as the file is closed during program termination. A third boundary for sequential files, a section, is not recognized during reading except for the special case of the file INPUT.

Mass Storage Input/Output

Files created by the random mass storage routines OPENMS, WRITMS, STINDX, and CLOSMS described in section 7 are word addressable files. The master index, which is the last record in the file, is created and maintained by FORTRAN routines rather than Record Manager routines.

One WRITMS call creates one U type record; one READMS call reads one U type record. If the length specified for a READMS is longer than the actual record, the excess locations in the user area are not changed by the read. If the record is longer than the length specified for a READMS, the excess words in the record are skipped.

Direct Access Input/Output

Files created by direct access READ and WRITE statements are word addressable files. There is no index. Except where the format specifies multiple records, one direct access WRITE creates one U type record and one direct access READ reads one U type record.

FILE CONTROL STATEMENT

The FILE control statement provides a means to override FIT field values compiled into a program and consequently a means to change processing normally supplied for standard input/output. In particular, it can be used to read or create a file with a structure that does not conform to the assumptions of default processing.

A FILE control statement can also be used to supplement standard processing. For example, setting DFC can change the type of Record Manager information listed in the dayfile.

At execution time, FILE control statement values are placed in the FIT when the referenced file is opened. These values have no effect if the execution routines do not use the fields referenced. Furthermore, FORTRAN routines may, in some cases, reset FIT fields after the FILE control statement is processed. These fields are noted in table F-1.

The format of the FILE control statement is shown in figure F-1.

FILE(lfn,field=value[,field=value]...)	
lfn	Is the file name as it appears on the execution control statement; if file name does not appear there, then lfn is file name as it appears in the PROGRAM or OPEN statement.
field	Is a FIT field mnemonic.
value	Is a symbolic or integer value.

Figure F-1. FILE Control Statement

The FILE control statement can appear anywhere in the control statements prior to program execution, but it must not interrupt a load sequence.

This deck shown in figure F-2 illustrates the use of the FILE control statement to override default values supplied by the FORTRAN compiler. Assuming the source program is using formatted writes and 100-character records are always written, the file is written on magnetic tape with even parity, at 800 bpi. No labels are recorded, and no information is written except that supplied by the user. The following values are used:

Block type = character count

Record type = fixed length

Record length = 100 characters

Conversion mode = YES

SEQUENTIAL FILE OPERATIONS

The sequential file operations are BACKSPACE/REWIND and ENDFILE.

Backspace/Rewind

Backspacing on FORTRAN files repositions them so that the previous record becomes the next record.

BACKSPACE is permitted only for files with F, S, or W record type or tape files with one record per block.

The user should remember that formatted input/output operations can read/write more than one record; unformatted input/output and BUFFER IN/OUT read/write only one record.

The REWIND operation positions a magnetic tape file so that the next FORTRAN input/output operation references the first record. A mass storage file is positioned to the beginning-of-information.

Table F-2 details the actions performed prior to positioning.

ENDFILE

Tables F-3 and F-4 indicate the action taken when an ENDFILE statement is executed. The action depends on the record and block type, as well as the device on which the file resides.

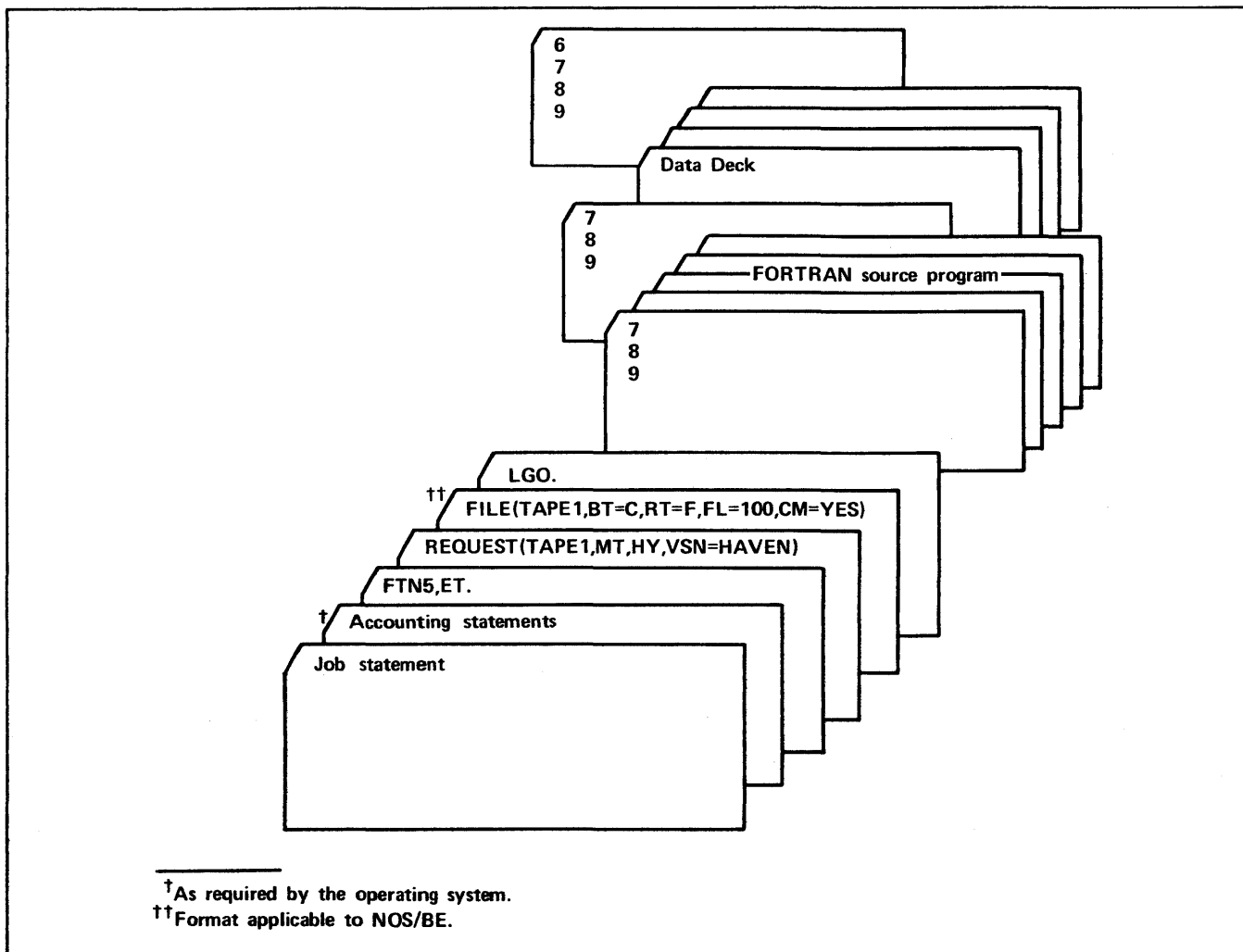


Figure F-2. FILE Control Statement Example

INPUT/OUTPUT RESTRICTIONS

Meaningful results are not guaranteed in the following circumstances:

Mixed formatted and unformatted read or write statements and buffer input/output statements on the same file (without an intervening REWIND, ENDFILE, or without encountering an end-of-file as determined by the EOF Function).

Requesting a LENGTH function or LENGTHX call on a buffer unit before requesting a UNIT function.

Two consecutive buffer input/output statements on the same file without the intervening execution of a UNIT function call.

Writing formatted records on a 7-track S or L tape without specifying CM=NO on a file control statement.

Writing F-type records with namelist or list-directed output.

Attempting to write a noise record on an S or L tape. This can occur with block types K and E (and C for SCOPE 2) using record types F,D,R,T, or U with MNB<noise size.

Sequential I/O operations REWIND, BACKSPACE, and ENDFILE on a direct access file.

RECORD MANAGER ERROR SUPPRESSION

For formatted, namelist, and list directed sequential reads, a default WSA size of 150 characters, the RECL= parameter value of an OPEN statement, or the record length parameter on the PROGRAM statement is passed to the record manager as the maximum record length (MRL).

If the record read from the file exceeds the MRL, record manager raises an RM142 error condition and posts a message to the job's dayfile.

In common FORTRAN usage this error condition is suppressed by runtime input processing routines. However, the informative RM142 dayfile message already generated by record manager cannot be suppressed.

TABLE F-2. ACTION BEFORE POSITIONING FOR BACKSPACE/REWIND

Condition	Device Type	Action
Last operation was WRITE or BUFFER OUT	Mass Storage	Any unwritten blocks for the file are written. An end-of-partition is written. If record format is W, a deleted zero length record is written.
	Unlabeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted zero length record is written. Two file marks are written.
	Labeled Magnetic Tape	Any unwritten blocks for the file are written. If record format is W, a deleted record is written. A file mark is written. A single EOF label is written. Two file marks are written.
Last operation was WRITE or BUFFER OUT ^{††}	Mass Storage	ENDFILE is issued. Any unwritten blocks for the file are written. End-of-information is written.
	Unlabeled Magnetic S or L Tape	ENDFILE is issued. Any unwritten blocks for the file are written. Two file marks are written.
	Labeled Magnetic Tape or Unlabeled System Magnetic Tape	ENDFILE is issued. Any unwritten blocks for the file are written. A tape mark is written. A single EOF label is written. Two tape marks are written.
Last operation was READ, BUFFER IN or BACKSPACE	Mass Storage	None.
	Unlabeled Magnetic Tape	None.
	Labeled Magnetic Tape	None.
No previous operation	All Devices [†]	REWIND request causes the file to be rewound when first referenced. If the file is assigned to on-line magnetic tape, a REWIND request is executed. For SCOPE 2, if the file is staged, the REWIND request has no effect. The file is staged and rewound when it is first referenced.
	Mass Storage ^{††}	
	Magnetic Tape ^{††}	
Previous operation was REWIND		Current REWIND is ignored.

[†]Applies to NOS and NOS/BE only.
^{††}Applied to SCOPE 2 only.

TABLE F-3. ENDFILE ACTION (NOS and NOS/BE)

Record Type	Device Type	
	S or L Tape	Other Device
W	An end-of-partition flag is written. The block is terminated.	An end-of-partition flag is written. The block is terminated with a short PRU of level 0.
Other	The block is terminated. A tape mark is written.	The block is terminated with a short PRU of level 0. A zero length PRU of level 17 is written.

TABLE F-4. ENDFILE ACTION (SCOPE 2)

Record Type	Blocking	
	Blocked	Unblocked
W	An end-of-partition flag is written. The block is terminated.	An end-of-partition flag is written.
Z	If C type blocking, the block is terminated. Otherwise, the block is terminated and a tape mark recovery control word is written.	A level 17 PRU is written.
S	If C type blocking, the block is terminated with a zero length PRU of level 17. Otherwise, the block is terminated and a tape mark recovery control word is written.	Not applicable.
Others on Mass Storage	The block is terminated. A tape mark recovery control word is written.	Ignored.
Others on Magnetic Tape	The block is terminated. A tape mark is written.	Not applicable.

COMPILE-TIME INPUT/OUTPUT

The compiler expects source input files to have certain characteristics and it produces coded and binary files which must be structured in specific ways according to the operating system under which it runs. A program compiled under SCOPE 2 must be executed under control of SCOPE 2; a program compiled under other operating systems cannot be executed under SCOPE 2. Programs compiled under NOS or NOS/BE can be executed under either of these operating systems.

Under SCOPE 2, the compiler uses SCOPE 2 Record Manager for all input/output operations. However,

a FILE control statement should not be used since the compiler overrides file information table settings after this control statement is processed. Under NOS and NOS/BE, the compiler makes direct calls to the operating system for input/output; CRM is not used.

SOURCE INPUT FILE STRUCTURE

A source input file must have a certain structure. Only the first 90 characters of each record are processed or reproduced in the listing output file. The characteristics are described in table F-5.

TABLE F-5. SOURCE INPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a short or zero length PRU	Sequential (FO=SQ) unblocked
Record type	Zero-byte terminated	Control word (RT=W)
Maximum record length	158 characters	158 characters (MRL=158)
Conversion mode	Not applicable	No (CM=NO)
Label type of tape	Under operating system control	Unlabeled (LT=UL)

CODED OUTPUT FILE STRUCTURE

Two coded output files can be produced: the listing file and the errors file. The characteristics are described in table F-6.

BINARY OUTPUT FILE STRUCTURE

The content of the executable object code file differs, depending on the loader supported by the operating system. The characteristics are described in table F-7.

TABLE F-6. CODED OUTPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a short PRU	Sequential (FO=SQ) unblocked
Maximum block length	Not applicable	None
Record type	Zero-byte terminated (equivalent to Record Manager Z type)	Control word (RT=W)
Maximum record length	137 characters	137 characters
Conversion mode	Not applicable	No (CM=NO)
Tape label type	Under operating system control	Unlabeled (LT=UL)

TABLE F-7. BINARY OUTPUT FILE STRUCTURE

File Characteristics	NOS/BE and NOS	SCOPE 2
File organization	Sequential operating system default format with file terminated by a zero length PRU which is then back-spaced over	Sequential (FO=SQ) unblocked
Record type	Operating system logical record (equivalent to Record Manager S type)	Control word (RT=W)
Maximum record length	None	1,310,710 characters
Conversion mode	Not applicable	No (CM=NO)
Tape label type	Under operating system control	Unlabeled (LT=U)

This appendix contains programming practices recommended by CDC for users of the software described in this manual. When possible, application programs based on this software should be designed and coded in conformance with these recommendations.

Two forms of guidelines are given. The general guidelines minimize application program dependence on the specific characteristics of a hardware system. The feature use guidelines ensure the easiest migration of an application program to future hardware or software systems.

GENERAL GUIDELINES

Programmers should observe the following practices to avoid hardware dependency:

Avoid programming hardcoded constants. Manipulation of data should never depend on the occurrence of a type of data in a fixed multiple such as 6, 10, or 60.

Do not manipulate data based on the binary representation of that data. Characters should be manipulated as characters, rather than as octal display-coded values or as 6-bit binary digits. Numbers should be manipulated as numeric data of a known type, rather than as binary patterns within a central memory word.

Do not identify or classify information based on the location of a specific value within a specific set of central memory word bits.

Avoid using COMPASS in application programs. COMPASS and other machine-dependent languages can complicate migration to future hardware or software systems. Migration is restricted by continued use of COMPASS for stand-alone programs, by COMPASS subroutines embedded in programs using higher-level languages, and by COMPASS owncode routines in CDC standard products. COMPASS should only be used to create part or all of an application program when the function cannot be performed in a higher-level language or when execution efficiency is more important than any other consideration.

Avoid using NOS PF subroutines due to anticipated changes to this feature in the future.

FEATURE USE GUIDELINES

The recommendations in the remainder of this appendix ensure the easiest migration of an application program for use on future hardware or software systems. These recommendations are based on known or anticipated changes in the hardware or software system, or comply with proposed new industry standards or proposed changes to existing industry standards.

ADVANCED ACCESS METHODS

The Advanced Access Methods (AAM) offer several features within which choices must be made. The following paragraphs indicate preferred usage.

Access Methods

The recommended access methods are indexed sequential (IS), direct access (DA), and multiple index processor (MIP).

Record Types

The recommended record types are either F for fixed length records, or W for variable length records. Record length for W records is indicated in the control word; the length must be supplied by the user in the RL FIT field on a put operation and is returned to the user in RL on a get operation.

FORTRAN Usage

The following machine-independent coding practices are encouraged for a FORTRAN programmer using AAM:

Initialize the FIT by FILExx calls or by the FILE control statement.

Modify the FIT with STOREF calls.

Use the FORTRAN 5 CHARACTER data type when working with character fields rather than octal values of display code characters; specify lengths of fields, records, and so forth, in characters rather than words.

BASIC ACCESS METHODS

The Basic Access Methods (BAM) offer several features within which choices must be made. The following paragraphs indicate preferred usage.

File Organizations

The recommended file organization is sequential (SQ). For files with word-addressable (WA) organization, use an accessing technique that can easily be modified to byte addresses.

Block Types

The recommended block type is C.

Record Types

The recommended record types are F for fixed length records and W for variable length records. For purely coded files that are to be listed, Z type records can be used.

Block Size

Set the Maximum Block Length (MBL) to 640 characters for mass storage files and 5120 characters for tape files.

Host Language Input/Output

Use of host language input/output statements (for example, a FORTRAN READ statement) to process BAM files is always a safe procedure. Host language statements provide appropriate default values for record type, block type, and block size. Do not use the CYBER Record Manager FORTRAN interface routines to process sequential files.

Collating Sequence

The default collating sequence or the ASCII collating sequence should be used.

FORTRAN 5

FORTRAN 5 offers users several capabilities that are processor-dependent. The use of such capabilities restrict FORTRAN 5 program migration. The following paragraphs indicate preferred usages.

Processor-dependent Values

Coding should not depend on the internal representation of data (floating-point layout, number of characters per word, and so forth). Where coding must depend on these representations, use parameter variables for processor-dependent characteristics such as the number of characters per word.

Boolean Data Types

Do not use Boolean data types and operations (SHIFT, MASK, and so forth) because they can be processor-dependent. Use type CHARACTER instead, if working with character data.

LOCF Function

Do not use the intrinsic function LOCF. For most applications, this function should not be necessary.

ENCODE and DECODE Statements

Do not use ENCODE and DECODE; use the ANSI standard internal files feature instead. ENCODE and DECODE are generally dependent on the number of characters per word.

DATE, TIME, and CLOCK Functions

Do not dismantle values returned by the DATE, TIME, and CLOCK functions; use these functions only for printing out values as a whole.

BUFFER IN and BUFFER OUT Statements

Do not use BUFFER IN and BUFFER OUT, especially when use depends on the number of characters per word.

Common Memory Manager Interface Routines

Avoid use of these routines because they are processor-dependent and inhibit portability.

CYBER Record Manager Interface Routines

Do not use the CYBER Record Manager interface routines for sequential files. Instead, use FORTRAN input/output statements such as READ or WRITE.

Overlays

If possible, use segmented loading instead. If overlays must be used, do not depend on such properties as reinitialization of variables when an overlay is reloaded.

LABEL Subroutine

Avoid use of the LABEL subroutine. Changes to the ANSI standard for tape labels might require changes to the interface used by this subroutine.

STATIC Memory Management and Capsule Loading

Do not use this capability unless absolutely necessary. Use of OVCAPs is preferred.

The user must be thoroughly aware of the capsules needed to perform the types of I/O operations required. It is the user's responsibility to ensure that the capsules are loaded by explicitly specifying the appropriate STLxxx subroutine call. Only default block and record types are supported by the STLxxx subroutines. To force load nondefault block type/record type handling of capsules, the user must use the following control statement sequence:

```
FILE,1fn,...,RT=...,BT=...,USE=...  
LDSET(STAT=1fn)
```

SORT/MERGE VERSIONS 4 AND 1

Sort/Merge offers several features among which choices must be made. The following paragraphs indicate preferred usage.

Key Alignment

Ensure that SORT keys are aligned on character or word boundaries. Do not place SORT keys in arbitrary bit positions within words.

SORT and MERGE Statements

Always perform logically separated SORT and MERGE operations with separate control statements.

INDEX

- A edit descriptor 5-7
- Abort, recovery 7-15
- ABS 7-7
- ACOS 7-7
- Actual arguments 6-10
- Adjustable dimensions 6-11
- ATMAG 7-7
- AINT 7-7
- AL 11-2
- ALOG 7-7
- ALOG10 7-7
- Alternate return 6-8
- AMAX0 7-7
- AMAX1 7-7
- AMINO 7-7
- AMIN1 7-7
- AMOD 7-7
- AND 7-7
- ANINT 7-7
- ANSI diagnostics 11-2.1
- Apostrophe
 - Character constant 1-1, 1-7
 - In FORMAT specification 5-8
- Argument list format 8-29
- Arguments
 - Actual 6-10
 - Dummy or formal 6-10
- Arithmetic
 - Assignment 3-9
 - Expressions 3-1
 - IF statement 4-3
 - Operators 3-2
- Arrays
 - And Substrings 1-12
 - Assumed-size 1-10, 6-12
 - Dimensions 1-9
 - Element location 1-11
 - EQUIVALENCE 2-6
 - In subprogram 6-12
 - NAMELIST 5-30
 - Structure 1-10
 - Subscripts 1-10
 - Transmission 6-12
 - Type statements 1-9, 2-11
- ASIN 7-7
- ASSIGN statement 4-1
- Assigned GO TO 4-2
- Assignment statements
 - Arithmetic 3-9
 - Boolean 3-9
 - Character 3-9
 - Logical 3-9
 - Multiple 3-10
 - Statement label 4-1
- Asterisk
 - Arguments 6-11
 - Comment 1-2
 - In SUBROUTINE statement 6-12
 - Multiplication 3-2
- ATAN 7-7
- ATANH 7-7
- ATAN2 7-8

- BACKSPACE 5-44
- Batch job files 5-2

- Binary
 - I/O, see Unformatted input/output 5-26
 - Program execution 11-1, 11-3, 11-26
- Blank Common 2-1
- Block
 - Common 2-1, 6-12
 - Data subprogram 6-13
- Block IF
 - Nested 4-5
 - Statement 4-3
 - Structures 4-4
- BN edit descriptor 5-9
- BOOL 7-8
- Boolean
 - Constants 1-6
 - Expressions 3-4
 - Type statement 2-13
 - Variables 1-8
- BOOLEAN statement 2-13
- Buffer
 - In OPEN statement 5-36
 - In PROGRAM statement 6-2
 - Input/output 5-32
- BUFFER IN statement 5-34
- BUFFER OUT statement 5-35
- BZ edit descriptor 5-9

- C comment line 1-2
- CABS 7-8
- CALL statement 6-5
- Calling
 - Overlay 9-4
 - Subroutine 6-5, 6-10
- Carriage control 5-9
- CCOS 7-8
- CEXP 7-8
- CHAR 7-8
- Character
 - Arguments 6-10
 - Constants 1-7
 - DATA initialization 2-5
 - Editing 5-7
 - Expressions 3-5
 - String 5-8
 - Substrings 1-11
 - Type statement 2-13
 - Variables 1-8
- Character set
 - CDC 1-1, A-1
 - FORTAN 1-2, A-1
- CHARACTER statement 2-13
- CHEKPTX 7-15
- CLOCK 7-14
- CLOG 7-8
- CLOSE statement 5-38
- CLOSEMS 7-23
- CMPLX 7-8
- Collation control 7-30, 7-31, E-3
- COLSEQ 7-31
- Column usage 1-2
- Comment line 1-2
- Common
 - And equivalence 2-6
 - Overlay communication 9-2

Common (Contd)
 Statement 2-1
 Usage 2-1, 6-12
 Common Memory Manager 8-1, 8-24
 COMMON statement 2-1
 COMPASS
 Calling sequence 8-28
 Program entry points 8-29
 Subprogram 8-28
 Compilation
 Control statement 11-1
 Listings 11-12
 Optimization 11-7
 Compile-time diagnostics B-1
 Compiler
 Call 11-1
 Diagnostics B-1
 Directive E-1
 Output listings B-1, B-28
 Supplied functions 7-1
 COMPL 7-8
 Complex
 Constants 1-6
 Editing 5-7
 Type statement 2-12
 Variables 1-8
 COMPLEX statement 2-12
 Computed GO TO 4-1
 Concatenation 3-5
 Conditional compilation E-1
 CONJG 7-8
 CONNEC 7-20
 Constants
 Boolean 1-6
 Character 1-7
 Complex 1-6
 Double precision 1-5
 Hexadecimal 1-7
 Hollerith 1-7
 Integer 1-4
 Logical 1-6
 Octal 1-7
 Real 1-5
 Symbolic 1-4, 2-11
 Types of 1-4
 Continuation line 1-2
 CONTINUE statement 4-8
 Control
 Carriage 5-9
 Column 5-19
 Listing 5-27
 Control statement
 DEBUG 10-1
 EXECUTION 11-27
 FILE F-5
 FTN5 11-1
 Conversion
 Data on input/output 5-7
 Mixed mode 3-2, 3-9
 Specification for input/output 5-5
 COS 7-8
 COSD 7-8
 COSH 7-8
 CRM utilities 8-16
 Cross-reference map 11-13, 11-22
 CSIN 7-8
 CSOWN 7-32
 CSQRT 7-9
 CYBER Database Control System 8-30
 CYBER Interactive Debug 10-1
 CYBER Record Manager
 Call syntax 8-6
 File handling F-1
 File information table 8-6
 File processing 8-6

CYBER Record Manager (Contd)
 Interface 8-1, 8-5
 Utilities 8-16
 C\$ Directives 1-3, E-1
 D edit descriptor 5-9
 DABS 7-9
 DACOS 7-9
 DASIN 7-9
 Data conversion on input/output 5-7
 DATA statement 2-3, 2-4
 DATAN 7-9
 DATAN2 7-9
 DATE 7-14
 Dayfile messages 7-15
 DBLE 7-9
 DCOS 7-9
 DCOSH 7-9
 DDIM 7-9
 DEBUG control statement 10-1
 Debugging aids
 CYBER Interactive Debug 10-1
 LIMERR 7-28
 NUMERR 7-28
 Post Mortem Dump 10-3
 Reference map 11-21
 Deck structure 12-1
 Declarative statements (see Specification statements)
 DECODE statement 5-42
 DEXP 7-9
 Diagnostics
 Compilation B-1, B-2
 Compiler output listing messages B-1, B-28
 Execution B-1, B-29
 Special compilation B-1, B-26
 DIM 7-9
 DIMENSION
 Adjustable 6-11
 Statement 2-5
 DINT 7-9
 Direct access input/output 5-1, F-5
 DISCON 7-21
 DISPLA 7-14
 Display code A-1
 Division 3-2
 DLOG 7-9
 DLOG10 7-9
 DMAX1 7-9
 DMIN1 7-9
 DMOD 7-9
 DNINT 7-10
 DO loops
 Active and inactive 4-7
 Implied in DATA list 2-4
 Implied in I/O list 5-3
 Nested 4-7
 Range 4-6
 DO statement 4-6
 Double precision
 Constants 1-5
 Editing 5-9, 5-11
 Type declaration 2-12
 Variables 1-8
 DOUBLE PRECISION statement 2-12
 DPROD 7-10
 DSIGN 7-10
 DSIN 7-10
 DSINH 7-10
 DSQRT 7-10
 DTAN 7-10
 DTANH 7-10
 DUMP 7-27

- E edit descriptor 5-11
- ECS (see Extended memory)
- ELSE statement 4-4
- ELSE IF statement 4-4
- ENCODE statement 5-42
- END IF statement 4-4
- END statement 4-10, 6-7
- ENDFILE statement 5-44, F-5
- END= 5-24
- ENTRY statement 6-6
- EOF 7-18
- EQUIVALENCE statement 2-6
- EQV 7-10
- ERF 7-10
- ERFC 7-10
- Error processing
 - SYSTEM or SYSTEMC 7-27
- ERR= 5-24
- Evaluation of expressions 3-8
- Execution control statement 11-27
- Execution time
 - Diagnostics B-1, B-29
 - File name handling F-1
 - FORMAT 5-22
 - Input/output 5-22
- EXIT 7-15
- EXP 7-10
- Exponentiation 3-2
- Expressions
 - Arithmetic 3-1
 - Boolean 3-4
 - Character 3-5
 - Evaluation 3-8
 - General rules for 3-8
 - Logical 3-5
 - Relational 3-6
 - Subscripts 1-9
- Extended memory 2-9
- External function 2-7, 6-4
- EXTERNAL statement 2-7

- F edit descriptor 5-13
- FALSE 1-6
- FILE control statement F-5
- File
 - Definition F-1
 - Name substitution 11-27
 - Name (TAPEk) 5-23, 11-27
 - Positioning 5-43
 - Sequential F-4
 - Status 5-36
 - Structure F-1
 - Usage 5-23
- File information table (FIT)
 - Defaults for standard I/O F-2
 - Defined F-1
 - Direct call by CYBER Record Manager 8-6
- FLOAT 7-10
- FMT= 5-24
- Formal argument (parameter) (see Dummy argument)
- FORMAT statement 5-5
- Format
 - Control, termination of 5-21
 - Execution time 5-22
 - Specification 5-4
- Formatted
 - Input/output 5-25
 - PRINT statement 5-26
 - READ statement 5-25
 - WRITE statement 5-25
- FORTRAN
 - Compiler call 11-1
 - Syntax summary D-1
- FTN5 control statement 11-1
- Function
 - Intrinsic 2-8, 7-1
 - Referencing 6-6
 - Statement 6-5
 - Subprogram 6-5
- Future System migration G-1

- G edit descriptor 5-15
- GETPARM 7-13
- Glossary C-1
- GO TO statements
 - Assigned GO TO 4-1
 - Computed GO TO 4-1
 - Unconditional GO TO 4-1

- H edit descriptor 5-15
- H specification
 - In format specification 5-15
 - Hollerith constant 1-7
- Hexadecimal/octal conversion 5-17
- Hexadecimal constant 1-7
- Hierarchy in expressions 3-2, 3-6
- Hollerith
 - Constant 1-7
 - Format specification 5-15

- I edit descriptor 5-15
- IABS 7-10
- ICHAR 7-10
- IDIM 7-10
- IDINT 7-10
- IDNINT 7-10
- IF statements
 - Arithmetic IF 4-3
 - Block IF 4-3
 - Logical IF 4-3
- IFIX 7-11
- IMPLICIT statement 2-7
- Implicit typing of variables 2-7, 2-11
- Implied DO
 - In DATA list 2-4
 - In I/O list 5-3
- INDEX 7-11
- Index
 - DO loop 4-6
 - Mass storage files 7-23
- Information Management Facility 8-35
- Initial line 1-2
- INPUT file 5-40
- Input/output
 - BUFFER 5-32
 - Compile time 5-4
 - Direct access 5-1
 - Execution time 5-22
 - Formatted 5-25
 - Implementation F-1
 - Keyword = value forms 5-22
 - List directed 5-27
 - Lists 5-3
 - Mass storage 7-22
 - NAMELIST 5-30
 - Restrictions F-6
 - Status checking 7-17
 - Status statements 5-36
 - Unformatted 5-26
- INQUIRE statement 5-38
- INT 7-11
- Integer
 - Constants 1-4
 - Editing 5-6

Integer (Contd)
 Type declaration 2-12
 Variables 1-8
 INTEGER statement 2-12
 Internal files
 Extended 5-41
 Standard 5-40
 Intrinsic functions 2-8, 7-1
 INTRINSIC statement 2-8
 IOCHEC 7-18
 Iolist 5-25
 IOSTAT= 5-24, 5-27
 ISIGN 7-11

 JDATE 7-14
 Job decks, examples 12-1
 Job files, batch 5-2

 L edit descriptor 5-17
 L format Hollerith constant 1-7
 LABEL 7-21
 Labeled
 Common 2-1
 Files 7-20
 Labels
 Statement labels 1-2
 Use in alternate return 6-8
 LCM (see Extended memory)
 LEGVAR 7-27
 LEN 7-11
 LENGTH, LENGTHX 7-19
 LEVEL Statement 2-9
 Levels, overlay 9-1
 LGE 7-11
 LGO 11-3, 11-27
 LGT 7-11
 Library functions 7-1
 LIMERR 7-28
 List directed
 Input 5-27
 PRINT 5-29
 PUNCH 5-29
 Output 5-29
 READ 5-27
 WRITE 5-29
 Listings
 Control of 11-13
 Object 11-20
 Reference map 11-13
 Source 11-13
 L List File 11-6
 LLE 7-11
 LLT 7-11
 LOCF 7-11
 LOG 7-12
 Logical
 Assignment statement 3-9
 Constants 1-6
 Expressions 3-5
 File names 5-1
 IF statement 4-3
 Operators 3-6
 Unit number 5-1
 Variables 1-8
 LOGICAL statement 2-12
 LOG10 7-12
 Loops
 DO 4-6
 Implied in DATA statement 2-4
 Implied in input/output statements 5-3
 Nested 4-7

 Main program 6-1
 Map, reference 11-13
 MASK 7-12
 Mass storage input/output
 CLOSMS 7-23
 OPENMS 7-22.1
 READMS 7-22.1
 STINDX 7-23
 WRITMS 7-22.1
 Mathematical functions 7-1
 MAX 7-12
 MAXO 7-12
 MAXI 7-12
 Messages
 Compilation diagnostics B-1
 Compiler output listing B-1, B-28
 Execution diagnostics B-1, B-29
 Special compilation diagnostics B-1, B-26
 MIN 7-12
 MINO 7-12
 MIN1 7-12
 Mixend mode arithmetic conversion 3-2, 3-3, 3-9
 MOD 7-12
 Mode
 Debug 10-1
 Nonsequenced 1-1
 Optimizing 11-7
 Sequenced 1-3
 MOVLCB 7-19
 MOVLEV 7-19
 Multiple
 Assignment statement 3-10
 Binary value 11-1
 Entry 6-7
 Return 6-7

 Named common 2-1
 Namelist
 PRINT 5-32
 PUNCH 5-32
 READ 5-30
 WRITE 5-32
 NAMELIST statement 5-30
 Names
 Common block 2-1
 File 5-1
 Program unit 1-4, 6-1
 Symbolic 1-4
 Variable 1-8
 NEQV 7-12
 Nesting
 Block IF structures 4-5
 DO loops 4-7
 Parentheses 3-8
 NINT 7-12
 Nonsequenced mode 1-1
 Number
 Formats (see Constants)
 Statement label 1-2
 NUMERR 7-28

 O edit descriptor 5-17
 Object code 11-6, 11-20
 Octal Constants 1-7
 Offset 1-10, 5-1
 OPEN statement 5-36
 OPENMS 7-22.1
 Operands, evaluation of 3-2
 Operating system interface routines 7-14

Operators
 Arithmetic 3-2
 Boolean 3-4
 Character 3-5
 Logical 3-6
 Relational 3-7
 Optimization
 Object code 11-7
 Source code 11-8
 Unsafe 11-8
 Options, FTN5 control statement 11-1
 OR 7-12
 Order, statements in program unit 1-12
 Output (see Input/output)
 File 5-25
 Print limit specification 11-27
 Record length 5-34
 OVCAPS 9-5
 OVERLAY statement 9-4
 Overlays 9-1

 P scale factor 5-17
 Parameter, see Argument
 PARAMETER statement 2-9
 Parameters, FTN5 control statement 11-2
 Pass by reference 8-28
 Pass by value 8-29
 PAUSE statement 4-9
 PD 11-9
 PDUMP 7-27
 Permanent file 8-1
 PF call 8-2
 PL 11-9
 PMD 10-5
 PMDARRAY 10-5
 PMDLOAD 10-6
 PMDSTOP 10-6
 PMDDUMP 10-5
 PN 11-9
 Post Mortem Dump 10-3, 11-28
 Precedence of operators 3-2
 Print
 Control characters 5-9
 Density 11-9
 Limit specification 11-27
 PRINT statement 5-26
 Procedures 6-3
 Program
 Examples 12-8.1
 Maps 11-13
 Units 6-1
 PROGRAM statement 6-1
 PS 11-9
 Punch codes A-1
 PUNCH
 File 5-2
 Statement 5-26
 PW 11-9

 QC 11-10
 Queued Terminal Record Manager 8-36
 Quote
 Character string delimiter 1-7
 Edit descriptor 5-8

 R edit descriptor 5-18
 R format Hollerith constant 1-7
 Random
 Access 7-21
 Number routines 7-14
 RANF 7-12

 Range of DO loops 4-6
 RANGET 7-14
 RANSET 7-14
 READ statements
 Direct access 5-2
 Formatted 5-25
 Internal 5-40
 List directed 5-27
 Namelist 5-30
 Unformatted 5-27
 READMS 7-22.1
 Real
 Constant 1-5
 Variable 1-8
 REAL 7-12
 REAL statement 2-12
 Record
 Definition F-1
 Length 5-1, 6-2
 Types F-1
 Record Manager (see CYBER Record Manager)
 Recovery 7-15
 RECOVER 7-15
 Reference, function 6-9
 Reference map 11-13
 Relational
 Evaluation 3-6
 Expressions 3-6
 Operators 3-7
 REMARK 7-15
 RETURN statement 6-7
 REW 11-10
 REWIND statement 5-43
 ROUND 11-10

 S edit descriptor 5-19
 S system text file 11-10
 Sample
 Coding form 1-1
 Decks 12-1
 FTN5 control statement 11-11
 Programs 12-8.1
 SAVE statement 2-10
 Scale factor 5-17
 Scaling 5-18
 SECOND 7-13
 Sense switch 7-15
 Separator, slash and comma 5-7
 Sequenced mode 1-3
 Sequential access input/output 5-1
 Sequential file structure F-4, F-5
 SHIFT 7-13
 SIGN 7-13
 SIN 7-13
 SIND 7-13
 SINH 7-13
 Slash in FORMAT statement 5-13
 SNGL 7-13
 Sort/Merge 5 8-18
 Subroutines 8-19
 Sort/Merge 4 and 1 8-21
 SP edit descriptor 5-19
 Specification statements 2-1
 SQRT 7-13
 SS edit descriptor 5-19
 SSWTCH 7-15
 Standard, FORTRAN ANSI v
 Statement
 Format 1-1
 FORTRAN (see individual statement name)
 Function name 1-4
 Labels 1-2
 Order in program unit 1-12

- Statement functions 6-8
- STATIC capsule loading 7-32
- STINDEX 7-23
- STOP statement 4-10
- STRACE 7-27
- Structure
 - Block IF 4-4
 - Program unit 6-1
- Subprogram linkage 8-28
- Subprograms
 - Block data 6-13
 - Function 6-5
 - Miscellaneous utility 7-13
 - Subroutine 6-3
- Subroutines, calling 6-3
- SUBROUTINE statement 6-4
- Subscripts 1-9, 11-28
- Substrings 1-11
- Symbolic names 1-4
- Syntax summary D-1
- SYSTEM and SYSTEMC 7-27

- T edit descriptor 5-19
- Tabulation control 5-19
- TAN 7-13
- TAND 7-13
- TANH 7-13
- TAPEk 5-23, 6-2, 11-27
- Terminal interface 7-20
- Texts, system 8-28
- TIME 7-14

- TL edit descriptor 5-19
- Tn edit descriptor 5-19
- TR edit descriptor 5-19
- Traceback 10-4
- Transaction Facility 8-36
- TRUE 1-6
- Type of

- Arithmetic expressions 3-1
- Functions 6-6
- Variables 1-8

- Type statements
 - Dimension information in 1-9, 2-11
 - Explicit 2-11
 - Implicit 2-11

- Unconditional GO TO 4-1
- Unformatted input/output
 - READ 5-27
 - WRITE 5-27

- UNIT 7-18
- UNIT= 5-23

- User parameters 11-27
- Utility subprograms 7-14

- Variable
 - FORMAT statements 5-25
 - Name and type 1-8
- Variables
 - Boolean 1-8
 - Character 1-9
 - Complex 1-8
 - Double precision 1-8
 - Integer 1-8
 - Logical 1-8
 - Real 1-8

- Weight tables 7-31

- WRITE statement
 - Direct access 5-2
 - Formatted 5-25
 - Internal 5-41
 - List directed 5-29
 - Namelist 5-32
 - Unformatted 5-27

- WRITMS 7-22.1

- WTSET 7-32

- X edit descriptor 5-21
- XOR 7-13

- Z edit descriptor 5-21

- 8-bit subroutines 8-30

- .AND. 3-6
- .EQ. 3-6
- .EQV. 3-6
- .FALSE. 1-6
- .GE. 3-7
- .GT. 3-7
- .LE. 3-7
- .LT. 3-7
- .NE. 3-7
- .NEQV. 3-6
- .NOT. 3-6
- .OR. 3-6
- .TRUE. 1-6
- .XOR. 3-6

*

- In column 1 1-2

- In SUBROUTINE statement 6-11

- " or #

- Hollerith constant 1-1, 1-7

- In FORMAT specification 5-8

- / end-of-record indicator 5-13

- or ↑

- Character constant 1-1, 1-7

- In FORMAT specification 5-8

- : in FORMAT specification 5-21

COMMENT SHEET

MANUAL TITLE: FORTRAN Version 5 Reference Manual

PUBLICATION NO.: 60481300

REVISION: H

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments on the back (please include page number references).

_____ Please reply

_____ No reply necessary

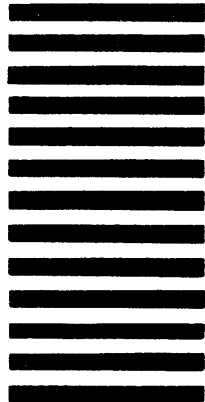
FOLD

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.



POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
P.O. BOX 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD

FOLD

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE

TAPE

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A



CONTROL DATA CORPORATION