CRAY RESEARCH, INC.

CRAY X-MP AND CRAY-1®
COMPUTER SYSTEMS

FORTRAN (CFT)
REFERENCE MANUAL
SR-0009

RECORD OF REVISION PUBLICATION NUMBER SR-0009

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
2520 Pilot Knob Road
Suite 310
Mendota Heights, Minnesota 55120

Revision	Description
	July 1976 - Preliminary distribution, copies
A	February 1977 - First printing. Since changes are very extensive, they are not noted by change bars.
B	November 1977 - Second printing. Since this represents a complete rewrite, changes are not noted by change bars.
C	April 1978 - Updates the manual to be in full agreement with the April 1978 release of the CRAY-1 FORTRAN Compiler (CFT) Version 1.01.
C-01	July 1978 - Included in this change packet, which brings the manual into agreement with the FORTRAN Compiler Version 1.02, is a new description of listable output, changes to the type statements, the addition of several utility procedures, and several new messages.
C-02	October 1978 - This change packet brings the manual into agreement with the FORTRAN compiler, Version 1.03. It includes the FLOWTRACE directive, new CFT messages, DO-loop table list option, and the ABORT subroutine.
D	January 1979 - Reprint. This printing brings the manual into agreement with the FORTRAN compiler, Version 1.04. Major changes include DO-variable usage; addition of ERR and END to the control information list; scheduler directives; the TRUNC parameter on the CFT card; M, R, and W compiler options; vector and code generation information with intrinsic functions and utility procedures; new subroutines ERREXIT, REMARK2, and TRBK; and new CFT messages.
D-01	April 1979 - This change packet brings the manual into agreement with version 1.05 of the FORTRAN compiler. Major changes include the alternate return feature, upper and lower bounds of DIMENSION declarators, and the NAMELIST statement.

E April 1979 - This revision is the same as Revision D with change packet D-01 incorporated.

E-01 July 1979 - This change packet brings the manual into agreement with version 1.06 of the CFT compiler. Major changes include conditional block statements ELSE IF, block IF, ELSE, and END IF; a new CFT directive BOUNDS; a new compiler option, O; a debugging utility, SYMDEBUG; and new CFT messages. Minor changes include clarification of Boolean arithmetic concepts and the introduction of dynamic memory allocation.

E-02 July 1979 - This change packet corrects a technical error appearing on page 6-6 of the E-01 change packet.

It also replaces pages inadvertently deleted by the E-01 change packet.

E-03 December 1979 - This change packet brings the manual into agreement with version 1.07 of the CFT compiler. Major changes include a symbolic debug package, enabled by the CFT control statement option Z; utility procedures that permit or prohibit floating-point interrupts and that determine the current floating-point interrupt mode; an enhancement to the editing process that allows D, E, F, G, and O format specifications to edit both real and double-precision list items; and reprieve processing routines.

F December 1979 - This reprint includes change packets E-01, E-02, and E-03. It contains no other changes.

F-01 April 1980 - This change packet brings the manual into agreement with version 1.08 of the CFT compiler. Major changes include lower-case letters in the CFT character set, character constants, the POINTER statement, full implementation of reprieve processing, new subprograms REMARKF and DUMPJOB, new CFT messages, and unblocked I/O.

G May 1980 - This revision is the same as Revision F with change packet F-01 incorporated.

G-01 October 1980 - This change packet brings the manual into agreement with version 1.09 of the CFT compiler. Major changes include arithmetic constant expressions; the IMPLICIT NONE statement; the implementation of the PAUSE statement; sequential, direct, and random access; file identifiers in input/output statements; the INQUIRE, OPEN, and CLOSE statements; further clarification on vector operations; page header lines on listable output; a new flowtrace routine, FLODUMP; a new CFT parameter, AIDS; and new CFT messages.

H August 1981 - Rewrite. With this printing, the manual has been completely reorganized and updated to agree with version 1.10 of the CFT compiler. Major changes include adherence to ANSI X3.9-1978 (FORTRAN 77), including the character data type and the generic function feature and adding list-directed I/O. Other miscellaneous changes were also added. Changes are not noted by change bars. All previous versions are obsolete.

H-01 August 1982 - This change packet brings the manual into agreement with version 1.10 of the CFT compiler. Major changes include adding to the comment lines description; new intrinsic function names; new internal file restrictions, changing the INQUIRE table, the OPEN table, and the CLOSE table; adding to the CLOSE statement description and the NAMELIST statement description; moving time functions, Boolean functions, and vectorization aids from Appendix C to Appendix B; new CFT messages; and the Hollerith format specification.

I November 1982 - This revision is the same as Revision H with change packet H-01 incorporated.

J April 1983 - This reprint with revision brings the manual into agreement with version 1.11 of the CFT compiler. The formats of the following have changed: character substring, CHARACTER type statement, COMMON statement, FORMAT statement, CALL statement, SUBROUTINE statement, RETURN statement, and INT24 directive. The following are additions: DATA statement restrictions; information to program control statements and input/output statements; user control subroutine; the MAXBLOCK and INT parameters on the CFT control statement; optimization options; the INT64 integer control directive; the multiply/divide directives (FASTMD, SLOWMD); the optimization directives NO SIDE EFFECTS, ALIGN, NOIFCON, and RESUMEIFCON; and vectorization and optimization information to Cray FORTRAN programming. The following items have changes: nonrepeatable edit descriptors and the format specifications. The calling sequence information was moved from Appendix F to the Macros and Opdefs Reference Manual, CRI publication SR-0012. The console attention handler information was removed from Appendix I.

J-01 July 1983 - This change packet brings the manual into agreement with the CFT 1.11 release. Changing the default of IF optimization from OPT=PARTIALIFCON to OPT=NOIFCON on the CFT control statement is the only major change. Miscellaneous technical and editorial changes are also included.

J-02 January 1984 - This change packet brings the manual into agreement with the CFT 1.13 release. The CFT release has been numbered 1.13 in conjunction with the 1.13 COS release. Major changes include the addition of: reentrancy support; new instruction scheduler; gather/scatter; dollar sign editing; the ALLOC, CPU, DEBUG, and SAVEALL control statement parameters; SAFEDOREP, FULLDOREP, NODOREP, INVMOV, NOINVMOV, UNSAFEIF, SAFEIF, BL, NOBL, BTREG, and NOBTREG control statement options; the U compiler option; UNSAFEIF, SAFEIF, BL, and NOBL scheduler directives; RESUMEDOREP and NODOREP optimization directives; implementing the ALIGN directive; DEBUG and NODEBUG directives; dependency information; population parity count Boolean function; and new CFT messages. The M and Y compiler options, the SCHED/NOSCH compiler directives, and several CFT messages have been removed. Miscellaneous technical and editorial changes are also included.

J-03 December 1984 - This change packet brings the manual into agreement with the CFT 1.14 release. Major changes include the addition of: extended memory common blocks; task common blocks; the EDN, UNROLL, and ANSI control statement parameters; the CVL/NOCVL and KEEPTEMP/KILLTEMP control statement options; CPU control statement parameter characteristics; table of parameters encountered; ROLL/UNROLL compiler directives; IVDMO vectorization control directive; CVL/NOCVL optimization directives; conditional vector loops; compressed index references; Bidirectional Memory; new CFT messages; and the FTREF utility. Miscellaneous technical and editorial changes are also included.

K January 1986 - This revision is the same as revision J with change packets J-01, J-02 and J-03 incorporated. No other changes have been made.

L February 1986 - This rewrite brings the manual into agreement with the CFT 1.15 under both COS 1.15 and UNICOS[†] 1.0. This revision obsoletes previous editions of the manual. Features incorporated into this rewrite include: correction of Boolean inconsistencies, code and data separation, CPU targeting, Do-loop table enhancements, generalized loop CIIs (CIVs), extension of local data addressability, initialization of stack variables to undefined values (INDEF), update to FLOWTRACE, vectorization of search loops, the LOOPMARK utility, function and subroutine recursion, and support for Cray C and Cray Pascal external procedures. Miscellaneous technical and editorial changes are also included.

[†] UNICOS is a trademark of Cray Research, Inc. and is derived from the AT&T UNIX system; UNIX is a trademark of AT&T Bell Laboratories.

PREFACE

The Cray FORTRAN (CFT) Compiler translates FORTRAN language statements into Cray Assembly Language (CAL) programs that make effective use of the CRAY X-MP and CRAY-1 Computer Systems. This manual describes the Cray FORTRAN language in its entirety; related characteristics of the Cray operating systems COS and UNICOS are described where applicable.

This manual is a reference manual for CFT programmers. The programmer is assumed to have a working knowledge of the FORTRAN programming language. However, when basic terms and concepts are being defined, they are italicized.

Related publications for the Cray FORTRAN programmer are:

SR-0011	COS Version 1 Reference Manual
SR-0012	Macros and Opdefs Reference Manual
SM-0017	FORTRAN (CFT) Internal Reference Manual
SR-0039	COS Message Manual
SR-0112	Symbolic Debugging Package Reference Manual
SR-0113	Programmer's Library Reference Manual
SN-0222	CRAY X-MP Multitasking Programmer's Manual
SR-2011	UNICOS Commands Reference Manual

CONTENTS

PREFACE vii

1. THE CFT COMPILER 1-1

1.1 THE COS CFT CONTROL STATEMENT 1-1
1.1.1 Allocation of variables to storage 1-10
1.2 THE UNICOS CFT COMMAND 1-13
1.3 COS ERROR MESSAGES DURING PROGRAM EXECUTION 1-20
1.4 INPUT TO CFT 1-20
1.5 OUTPUT FROM CFT 1-20
1.5.1 Page header lines 1-21
1.5.2 Source statement listings 1-21
1.5.3 Block begins messages 1-21
1.5.4 Table of statement numbers 1-22
1.5.5 Table of names encountered 1-22
1.5.5.1 Address field 1-22
1.5.5.2 Name field 1-22
1.5.5.3 Type field 1-22
1.5.5.4 Main usage field 1-24
1.5.5.5 Block field 1-24
1.5.6 Table of parameters encountered 1-25
1.5.7 Table of block names and lengths in octal . . . 1-25
1.5.8 Static Space Table anad Stack Space Table . . . 1-25
1.5.9 Table of external names 1-26
1.5.10 Table of loops encountered 1-26
1.5.11 Cross-reference information 1-27
1.5.12 Messages 1-28
1.5.13 Program Unit Page Table 1-28
1.6 COMPILER DIRECTIVES 1-28
1.6.1 Using compiler directive lines 1-29
1.6.2 Categories of compiler directives 1-29
1.6.2.1 Listable output control directives . . 1-30
1.6.2.2 Vectorization control directives . . . 1-31
1.6.2.3 Integer control directives (INT24, INT64) 1-36
1.6.2.4 Multiply/divide directives (FASTMD, SLOWMD) 1-36
1.6.2.5 Flow trace directives (FLOW/NOFLOW) . 1-37
1.6.2.6 Scheduler directives 1-40
1.6.2.7 Dynamic common block directive (DYNAMIC) 1-40
1.6.2.8 Array bounds checking directive (BOUNDS) 1-41

1.6.2 Categories of compiler directives (continued)
1.6.2.9 Optimization directives 1-42
1.6.2.10 Debugging directives (DEBUG, NODEBUG) 1-45
1.6.2.11 ROLL/UNROLL directives 1-46
1.7 EXTERNAL ROUTINES 1-46

2. PROGRAM STRUCTURE 2-1

2.1 THE EXECUTABLE PROGRAM 2-1
2.2 PROGRAM UNITS 2-3
2.2.1 Program statement 2-3
2.3 FUNCTIONS 2-4
2.3.1 Function reference 2-5
2.3.2 Statement functions 2-5
2.3.2.1 Statement function definition statement 2-6
2.3.3 Intrinsic functions 2-8
2.3.3.1 Referencing intrinsic functions . . . 2-8
2.3.3.2 Restrictions 2-8
2.4 SUBPROGRAMS 2-9
2.4.1 Recursion in subprograms 2-10
2.4.2 External functions and function subprograms . . 2-11
2.4.2.1 External functions 2-11
2.4.2.2 Function subprograms 2-12
2.4.2.3 FUNCTION statement 2-12
2.4.3 Subroutines and subroutine subprograms 2-13
2.4.3.1 Requirements 2-13
2.4.3.2 SUBROUTINE statement 2-14
2.4.3.3 CALL statement (subroutine reference) 2-14
2.4.4 Statements for using subprograms 2-16
2.4.4.1 ENTRY statement 2-16
2.4.4.2 RETURN statement 2-17
2.4.5 SAVE statement 2-18
2.4.6 Block data subprograms 2-19
2.4.6.1 BLOCK DATA statement 2-20
2.5 ARGUMENTS 2-20
2.5.1 Association of arguments 2-21
2.5.2 Actual arguments for external procedures 2-22
2.5.3 Dummy arguments 2-23
2.5.4 Dummy procedures 2-24
2.5.4.1 EXTERNAL statement 2-24
2.5.4.2 INTRINSIC statement 2-25

3. DATA TYPES 3-1

3.1 DATA SPECIFICATION STATEMENTS 3-2
3.1.1 Type statements 3-3
3.1.2 IMPLICIT statement 3-4
3.1.3 IMPLICIT NONE statement (CFT extension) 3-5
3.2 INTEGER TYPE 3-5

3. DATA TYPES (continued)

3.3 REAL TYPE 3-6
3.4 DOUBLE-PRECISION TYPE 3-7
3.5 COMPLEX TYPE 3-7
3.6 BOOLEAN TYPE 3-8
3.7 LOGICAL TYPE 3-10
3.8 CHARACTER TYPE 3-10
3.8.1 Character type statement 3-11
3.8.2 Character substrings 3-12
3.8.3 Arguments of type character 3-12
3.9 POINTERS 3-13
3.9.1 POINTER statement (CFT extension) 3-14
3.9.2 Restrictions 3-16

4. DATA STRUCTURES, STORAGE, AND ASSOCIATION 4-1

4.1 CONSTANTS 4-1
4.1.1 PARAMETER statement 4-1
4.2 VARIABLES 4-2
4.3 ARRAYS 4-3
4.3.1 Dummy and actual arrays 4-4
4.3.2 Constant, adjustable, assumed-size and pointee arrays 4-4
4.3.3 DIMENSION statement 4-5
4.3.4 Array declarators 4-5
4.3.4.1 Kinds of array declarators 4-7
4.3.4.2 Adjustable array declarators 4-7
4.3.5 Array elements and subscripts 4-7
4.3.5.1 Array subscripts and storage sequence 4-9
4.3.6 Array size 4-11
4.3.6.1 Size of an assumed-size array 4-11
4.3.6.2 Maximum array size 4-12
4.3.7 Arrays as arguments 4-12
4.3.8 Use of array names 4-13
4.4 DATA STATEMENT 4-14
4.4.1 Implied-DO list in a DATA statement 4-15
4.4.2 Data types in a DATA statement 4-16
4.5 STORAGE AND ASSOCIATION 4-17
4.5.1 Storage units and sequences 4-17
4.5.2 Static and stack storage 4-18
4.5.3 Definition 4-18
4.5.4 Association of entities 4-19
4.5.5 EQUIVALENCE statement 4-20
4.5.5.1 Array names and array element names . 4-21
4.5.5.2 Restrictions on EQUIVALENCE statements 4-21
4.6 COMMON BLOCKS 4-22
4.6.1 COMMON statement 4-23
4.6.2 Referencing common blocks 4-23

4.6 COMMON BLOCKS (continued)
4.6.3 Common block names 4-24
4.6.4 Common block storage sequence 4-24
4.6.5 Common block size 4-25
4.6.6 Extended memory common blocks 4-25
4.6.7 Task common statement 4-25
4.7 SYMBOLIC NAMES 4-26
4.7.1 Scope of symbolic names 4-27
4.7.1.1 Global entities 4-27
4.7.1.2 Local entities 4-27

5. EXPRESSIONS AND ASSIGNMENT 5-1

5.1 ARTHMETIC EXPRESSIONS 5-2
5.1.1 Arithmetic assignment statement 5-3
5.1.2 Arithmetic operators 5-4
5.1.2.1 Precedence of arithmetic operators . . 5-5
5.1.3 Arithmetic operands 5-6
5.1.3.1 Primaries 5-6
5.1.3.2 Factors 5-7
5.1.3.3 Terms 5-7
5.1.3.4 Arithmetic expressions 5-7
5.1.4 Data type of arithmetic expressions 5-8
5.1.4.1 Type conversion 5-11
5.1.5 Considerations in evaluating arithmetic expressions 5-12
5.2 CHARACTER EXPRESSIONS 5-13
5.2.1 Character assignment statement 5-13
5.2.2 Character expression evaluation 5-14
5.2.3 Hollerith type 5-14
5.3 RELATIONAL EXPRESSIONS 5-14
5.3.1 Arithmetic relational expressions 5-16
5.3.2 Character relational expressions 5-16
5.4 LOGICAL EXPRESSIONS 5-16
5.4.1 Logical assignment statement 5-17
5.4.2 Logical operators 5-17
5.4.3 Form and interpretation of logical expressions . 5-19
5.5 MASKING EXPRESSIONS 5-20

6. PROGRAM CONTROL STATEMENTS 6-1

6.1 GOTO STATEMENTS 6-1
6.1.1 Conditional GOTO statement 6-1
6.1.2 Computed GOTO statement 6-2
6.1.3 Assigned GOTO statement 6-3
6.2 ARITHMETIC IF STATEMENT 6-3
6.3 LOGICAL IF STATEMENT 6-4
6.4 CONDITIONAL BLOCKS 6-5
6.4.1 Block IF statement 6-6

6.4 CONDITIONAL BLOCKS (continued)
6.4.2 ENDIF statement 6-6
6.4.3 ELSEIF statement 6-6
6.4.4 ELSE statement 6-7
6.5 DO LOOPS 6-7
6.5.1 DO variable 6-8
6.5.2 Executing a DO statement 6-8
6.5.3 Loop control processing 6-9
6.5.4 DO statement 6-11
6.5.5 Terminal statement 6-11
6.5.6 CONTINUE statement 6-11
6.6 STOP STATEMENT 6-12
6.7 PAUSE STATEMENT 6-12
6.8 END STATEMENT 6-13

7. INPUT/OUTPUT STATEMENTS 7-1

7.1 INPUT/OUTPUT RECORDS 7-2
7.1.1 Formatted records 7-2
7.1.2 Unformatted records 7-2
7.1.3 End-of-file (endfile) records 7-2
7.1.4 End-of-data records 7-4
7.2 INPUT/OUTPUT FILES 7-4
7.3 COS DATASETS 7-4
7.4 INTERNAL RECORDS AND FILES 7-5
7.5 EXTERNAL FILE ACCESS METHODS 7-6
7.5.1 File position after data transfer 7-7
7.5.1.1 Sequential access 7-7
7.5.1.2 Direct access 7-7
7.6 UNITS 7-7
7.7 IDENTIFIERS 7-8
7.7.1 Unit identifiers 7-8
7.7.2 File identifiers 7-8
7.7.3 Format identifiers 7-9
7.8 DATA TRANSFER STATEMENTS 7-9
7.8.1 Control information lists 7-10
7.8.2 I/O lists 7-12
7.8.2.1 Input list items 7-12
7.8.2.2 Output list items 7-12
7.8.2.3 Implied DO lists 7-13
7.9 DATA TRANSFER OPERATIONS 7-14
7.9.1 Identifying a unit 7-14
7.9.2 Establishing a format 7-14
7.9.3 Transferring data 7-14
7.9.3.1 Unformatted data transfer 7-15
7.9.3.2 Formatted data transfer 7-15
7.9.4 Output to a printer 7-16
7.9.5 Error and end-of-file conditions 7-16
7.10 SEQUENTIAL FILE STATEMENTS 7-17
7.10.1 BACKSPACE statement 7-18

7.10 SEQUENTIAL FILE STATEMENTS (continued)
7.10.2 ENDFILE statement 7-18
7.10.3 REWIND statement 7-19
7.11 INQUIRE STATEMENT 7-19
7.11.1 Inquiry by file name 7-19
7.11.2 Inquiry by unit 7-20
7.11.3 INQUIRE statement restrictions 7-23
7.12 OPEN STATEMENT 7-23
7.13 CLOSE STATEMENT 7-24
7.14 NAMELIST STATEMENT (CFT EXTENSION) 7-27
7.14.1 NAMELIST input 7-28
7.14.1.1 NAMELIST input variables 7-29
7.14.1.2 NAMELIST input processing 7-30
7.14.1.3 User control subroutines 7-30
7.14.2 NAMELIST output 7-32
7.14.2.1 User control subroutines 7-33
7.15 BUFFER IN AND BUFFER OUT STATEMENTS (CFT EXTENSIONS) . . 7-35
7.15.1 The unit function (CFT extension) 7-37
7.15.2 The length function (CFT extension) 7-38
7.16 RANDOM INPUT/OUTPUT OPERATIONS 7-38
7.16.1 Creating a dataset for random access 7-38
7.16.2 Dataset connection 7-39
7.16.2.1 Positioning while connected for random access (GETPOS/SETPOS) 7-39
7.16.3 Modifying a record under random access 7-40
7.17 RESTRICTIONS ON INPUT/OUTPUT STATEMENTS 7-40
7.18 I/O ERROR RECOVERY 7-40
7.19 CHANGING MAXIMUM LENGTH FOR I/O LISTS AND FORMAT SPECIFICATIONS 7-41
7.19.1 Changing I/O buffer lengths using SEGLDR directives 7-41
7.19.2 Changing I/O buffer lengths by regenerating $IOLIB 7-41

8. INPUT/OUTPUT FORMATTING 8-1

8.1 UNFORMATTED I/O 8-1
8.2 LIST-DIRECTED I/O 8-2
8.2.1 List-directed input 8-3
8.2.2 List-directed output 8-4
8.3 FORMAT STATEMENTS 8-5
8.4 EDIT DESCRIPTORS 8-6
8.4.1 Interaction betwwen I/O lists and format specifications 8-10
8.4.2 Positioning by format control 8-11
8.4.3 Apostrophe and quotation mark editing 8-12
8.4.4 H editing 8-12
8.4.5 Positional editing (T, TL, TR, and X) 8-13
8.4.5.1 T, TL, and TR editing 8-13
8.4.5.2 X editing 8-14

8.4 EDIT DESCRIPTORS (continued)
8.4.6 Slash editing 8-14
8.4.7 Colon editing 8-15
8.4.8 Dollar sign editing (CFT extension) 8-15
8.4.9 P editing 8-16
8.4.10 Numeric editing (BN, BZ, S, SP, SS, I, F, E, D, and G) 8-17
8.4.10.1 BN and BZ editing 8-18
8.4.10.2 S, SP, and SS editing 8-18
8.4.10.3 Integer editing 8-18
8.4.10.4 F editing 8-19
8.4.10.5 E editing 8-21
8.4.10.6 D (double-precision) editing 8-23
8.4.10.7 G editing 8-23
8.4.11 Complex editing 8-25
8.4.12 O (octal) editing (CFT extension) 8-25
8.4.13 Z (hexadecimal) editing (CFT extension) 8-25
8.4.14 L (logical) editing 8-26
8.4.15 A (alphanumeric) editing 8-27
8.4.16 R (right-justified) editing (CFT extension) . . 8-28

9. CRAY FORTRAN PROGRAMMING 9-1
9.1 VECTORIZABLE DO LOOPS 9-1
9.1.1 Qualifications for vectorization 9-1
9.1.2 Entity categories 9-2
9.1.3 Dependencies 9-5
9.1.4 Conditional vector loops 9-11
9.1.5 Vectorization with arrays 9-11
9.1.6 Using optimized routines 9-13
9.1.7 Use of optimized routines by CFT 9-13
9.1.7.1 Conditional statements 9-15
9.1.8 Compressed index reference 9-18
9.1.9 General guidelines for vectorization 9-18
9.2 BIDIRECTIONAL MEMORY 9-19

APPENDIX SECTION

A. CHARACTER SET A-1

B. CRAY FORTRAN INTRINSIC FUNCTIONS B-1

B.1 CSMG (CRAY SCALAR MERGE) FUNCTION B-12

C. CRAY FORTRAN UTILITY PROCEDURES C-1

D. CFT MESSAGES D-1

D.1 COMPILE-TIME MESSAGES D-2
D.2 LOGFILE MESSAGES D-27
D.3 NOVECTOR MESSAGES D-29
D.4 INFORMATIVE DEPENDENCY MESSAGES D-38

E. OUTMODED FEATURES E-1

E.1 HOLLERITH DATA E-2
E.1.1 Hollerith expressions E-4
E.1.1.1 Hollerith relational expressions . . . E-5
E.1.2 Hollerith format specification E-6
E.2 TWO-BRANCH ARITHMETIC IF STATEMENTS E-6
E.3 INDIRECT LOGICAL IF STATEMENTS E-7
E.4 FORMATTED DATA ASSIGNMENT E-7
E.4.1 ENCODE and DECODE statements E-7
E.4.1.1 The ENCODE statement E-8
E.4.1.2 The DECODE statement E-9
E.5 EDIT DESCRIPTORS E-9
E.6 DOUBLE DECLARATION STATEMENTS E-10
E.7 DATA STATEMENT FEATURES E-11
E.8 PUNCH STATEMENT E-11
E.9 TYPE STATEMENT DATA LENGTH E-11
E.10 EXTENDED RANGE OF A DO LOOP E-13
E.11 NONCHARACTER ARRAYS FOR FORMAT SPECIFICATION E-13
E.11.1 EOF, IEOF, and IOSTAT functions E-13

F. CREATING NON-FORTRAN PROCEDURES F-1

F.1 CAL F-1
F.2 CRAY PASCAL F-1
F.3 CRAY C F-2

G. SYMBOLIC DEBUG PACKAGE G-1

H. UNBLOCKED DATASETS H-1

I. REPRIEVE PROCESSING I-1

I.1 REPRIEVE INITIATION I-1
I.2 REPRIEVE TERMINATION I-2

J. FTREF UTILITY J-1

FIGURES

2-1 Subcategories of FORTRAN Terms (Program Units Boxed) 2-2
4-1 Array Specification and Size 4-8
4-2 Storage Sequence for a Three-dimensional Array 4-10

TABLES

1-1 Compiler Options Under COS 1-12
1-2 Compiler Options Under UNICOS 1-18
1-3 External Routines 1-46
3-1 Values Represented in Different Data Types 3-16
4-1 Subscript Evaluation 4-11
5-1 Allowed Assignment Statements: y=x 5-4
5-2 Arithmetic Operators and Their Use in Expressions 5-5
5-3 Use of Data Types with Arithmetic Operations: +, -, *, / . . 5-9
5-4 Data Types in Exponentiation: ** 5-10
5-5 Data Types in Relational Operations: .EQ.,.NE.,.GT.,.GE., .LT.,LE. 5-15
5-6 Logical Operators 5-18
5-7 Meanings of Logical Operators 5-18
5-8 Allowed Logical and Masking Operations and Result Types . . . 5-21
7-1 CFT Input/Output Statements 7-1
7-2 Print Control Characters 7-16
7-3 INQUIRE Specifiers and Their Meanings 7-20
7-4 OPEN Specifiers and Their Meanings 7-25
7-5 CLOSE Specifiers and Their Meanings 7-27
8-1 Repeatable Edit Descriptors 8-7
8-2 Nonrepeatable Edit Descriptors 8-8
8-3 Edit Descriptors with Data Types 8-9
8-4 Edit Descriptors and Data Types When SEGLDR and the EQUIV Directive Are Used 8-10
9-1 Array A elements in vector and scalar modes 9-5
9-2 Dependency information combinations 9-7
A-1 Character Set A-2
B-1 Generic Arithmetic Functions B-2
B-2 Trigonometric Functions (Angles in Radians) B-4
B-3 Exponential Functions B-5
B-4 Logarithmic Functions B-6
B-5 Maximum/Minimum Functions B-6
B-6 Character Functions B-7
B-7 Type Conversion Functions B-7
B-8 Boolean Functions B-9
B-9 Time and Date Functions B-11
B-10 Miscellaneous Functions B-11
B-11 Vectorization Functions B-12
C-1 CFT Utility Procedures C-2
E-1 Data Length E-12

INDEX

1. THE CFT COMPILER

The Cray FORTRAN Compiler (CFT) transforms a Cray FORTRAN language program into a relocatable binary program that can be loaded and executed on the CRAY-1 or CRAY X-MP Computer System.

CFT can be used with the Cray operating system COS or the Cray operating system UNICOS. The COS and UNICOS compiler call formats are described separately in this section.

1.1 THE COS CFT CONTROL STATEMENT

Under COS, the CFT compiler is loaded and executed when a CFT control statement is encountered in the control statement stream.

Format:

CFT,AIDS=*aids*,ALLOC=*alloc*,ANSI,B=*bdn*,C=*cdn*,CPU=*cputype*:
  cpuchar,DEBUG,E=*eml*,EDN=*edn*,I=*idn*,INDEF,INT=*il*,L=*ldn*,
  LOOPMARK=*lmmsgs*,MAXBLOCK=*mb*,OFF=*opts*,ON=*opts*,OPT=*optim*,
  SAVEALL,TRUNC=*tr*,UNROLL=*r*.

Options can be in any order. If a keyword and option are omitted from the statement, the compiler uses a default value. The following control statement shows the default values for all options.

CFT,AIDS=LOOPPART,ALLOC=STATIC,B=$BLD,CPU=*(characteristics of machine executing CFT statement)*,E=3,I=$IN,INT=64,L=$OUT,
MAXBLOCK=2310,OFF=ABDFGHIJMNOWXZ,ON=CELPQRSTUV,OPT=BL:NOBTREG:
CVL:INVMOV:SLOWMD:KEEPTEMP:NOIFCON:SAFEIF:SAFEDOREP:NOZEROINC,
TRUNC=0,UNROLL=3.

A left parenthesis can be used in place of the first comma. A right parenthesis can be used in place of the period. If all options are omitted, a period can be used in place of empty parentheses. Dataset names are limited to 7 characters.

The compiler does not reposition datasets before or after compilation.

AIDS=*aids* Controls number of vectorization inhibition messages to be listed. *aids* can be one of the following.

aids	Description
LOOPNONE	No messages issued
LOOPPART	Maximum of 3 messages per inner DO loop up to a total of 100 messages per compilation (default option)
LOOPALL	All messages issued

ALLOC=*alloc*

Specifies memory allocation scheme for entities in memory. *alloc* can be one of the following.

alloc	Description
STATIC	All memory is statically allocated; a stack is not used (default option). (See the SAVEALL parameter, BTREG control statement option, and subsection 1.1.1, Allocation of Variables to Storage.)
STACK	Read-only constants and entities in a DATA statement, SAVE statement, or a common block are statically allocated. All other entities are allocated on the stack.

ANSI Enables non-ANSI messages to be printed at compile time. Some of these messages have a NOTE, CAUTION, or WARNING severity type when ANSI is not selected as an option. Specifying ANSI on the CFT control statement causes CFT to further analyze the compiled code and detect more occurrences of nonstandard FORTRAN. When ANSI is specified, messages indicating nonstandard code are issued with the prefix NON-ANSI instead of NOTE, CAUTION, or WARNING. A count of the non-ANSI messages is placed in the logfile. When ANSI is used, non-ANSI messages are issued regardless of the severity type of CFT messages selected with the E parameter. ANSI is disabled by default.

B=*bdn* Name of dataset on which compiler writes binary load modules; default is $BLD. If B=0, no binary load files are written. An end of file is not written.

C=*cdn* Name of pseudo Cray Assembly Language (CAL) dataset; default is no dataset. This option provides for the generation of a text file that contains acceptable input to the CAL assembler with minor manual corrections. DATA statements are not supported with this option. It is intended to be used for hand coding of inner loops for enhanced efficiency.

CPU=*cputype*:*cpuchar*

Specifies mainframe type and optional mainframe characteristics running the generated code; default is obtained from the operating system. Separate characteristics must be separated by commas.

cputype	Description
CRAY-1A	Generates code for CRAY-1 A Computer Systems
CRAY-1B	Generates code for CRAY-1 B Computer Systems
CRAY-1M	Generates code for CRAY-1 M Computer Systems
CRAY-1S	Generates code for CRAY-1 S Computer Systems
CRAY-XMP	Generates code for any CRAY X-MP Computer System
CRAY-X1	Generates code for single-processor CRAY X-MP Computer Systems
CRAY-X2	Generates code for dual-processor CRAY X-MP Computer Systems
CRAY-X4	Generates code for four-processor CRAY X-MP Computer Systems

characteristics	Description
AVL|NOAVL	Target machine does/does not have two vector logical functional units
BDM|NOBDM	Target machine does/does not have bidirectional memory
CI|NOCI	Target machine does/does not have compressed index hardware
CIGS|NOCIGS	Target machine does/does not have compressed index gather/scatter hardware (CIGS implies CI and GS; NOCIGS implies NOCI and NOGS)
EMA|NOEMA	Target machine does/does not have extended memory addressing
GS|NOGS	Target machine does/does not have gather/scatter hardware
IBUFSIZE=*words*	Instruction buffer size in words (16 or 32)
MEMSPEED=*cps*	Memory speed in clock periods
VPOP|NOVPOP	Target machine does/does not have a vector population count functional unit
VRECUR|NOVRECUR	Target machine does/does not have vector recursion

Both *cputype* and *cpuchar* are optional. If *cputype* is not specified, a default mainframe type and a set of default characteristics are obtained from the operating system.

If a mainframe type is specified, unspecified characteristics are assumed to be the minimum traits for that mainframe. For example, if CPU=CRAY-X2 is specified, the following characteristics are assumed.

```
NOEMA:NOCI:NOGS:NOCIGS:VPOP:NOAVL:
BDM:NOVRECUR:IBUFSIZE=32:MEMSPEED=11
```

Characteristics specified on the CPU parameter override the values obtained from the operating system.

DEBUG

Writes sequence number labels at each executable FORTRAN statement to the Debug Symbol Table, allowing breakpoints to be set with SID at statement sequence numbers. DEBUG forces ON=IZ and sets MAXBLOCK=1. DEBUG on the control statement enables recognition of CDIR$ DEBUG and CDIR$ NODEBUG.

If DEBUG is not specified on the control statement (default), CDIR$ DEBUG and CDIR$ NODEBUG are ignored and debugging is turned off for the compilation.

E=*eml*

Highest level of severity of CFT-produced messages to be listed. For example, E=2 allows CAUTION, WARNING, and ERROR messages to appear. Fatal errors are never suppressed. Default is E=3. If E=0 is specified, no suppression takes place. The following levels are available. (Also see the ANSI parameter.)

eml	Severity Type	Description
1	COMMENT	Comments on programming inefficiencies (vectorization messages are controlled by the AIDS parameter)
2	NOTE	May cause problems with other compilers (Example: non-ANSI 66)
3	CAUTION	Possible user error (Example: no path to a statement)
4	WARNING	Probable user error (Example: using an array with too few subscripts)
5	ERROR	Fatal error

EDN=*edn* Name of dataset receiving an alternate error listing; default is no dataset. Error messages with a higher severity type than E=*n* type are printed on dataset *edn*. Error messages printed to *ldn* are not affected by EDN.

I=*idn* Name of dataset containing source input; default is $IN.

INDEF Enables the stack frame to be initialized to the octal value 0605054000000070000000 upon entering a routine. INDEF causes a floating-point error to occur when floating-point arithmetic is performed on an uninitialized real variable, and causes an operand range error to occur when an uninitialized integer variable is used as an array subscript. INDEF is ignored if ALLOC=STACK is not specified.

INT=*il* Length of integers. *il* values are as follows.

il	Description
64	Full 64-bit integers (default option)
24	Short 24-bit integers

L=*ldn* Name of dataset to receive list output; default is $OUT. L=0 suppresses all list output except for error messages written on $OUT. If L=0 is specified, individual list options (see table 1-1) specified by the ON= specification are overridden.

LOOPMARK[=*lmmsgs*]
Activates the LOOPMARK utility. LOOPMARK draws brackets delimiting DO loops in the source code listing. Each loop is marked with a symbol indicating the type of loop (see subsection 1.5.9, Table of Loops Encountered). If LOOPMARK or LOOPMARK=MSGS is specified, a message is printed on the line following each inner DO loop that did not vectorize, indicating the reason that the loop did not vectorize.

The *lmmsgs* parameter indicates whether messages are to be printed by the LOOPMARK utility.

lmmsgs	Description
MSGS	Messages are printed for all inner DO loops that did not vectorize (default if LOOPMARK is specified alone)
NOMSGS	No messages are printed

LOOPMARK implies ON=D. LOOPMARK is disabled by default.

MAXBLOCK=*mb*

Allows CFT to optimize or vectorize a block of code with a length up to *mb* words. Default is 2310 words of internal intermediate text. Values larger than 2310 may increase optimization but there may also be internal compiler errors (the errors may be undetected by CFT). MAXBLOCK=1 eliminates optimization and vectorization.

ON=*string* Enables compile options (see table 1-1, list of up to 15 characters representing options to be enabled)

OFF=*string*

Disables list or compile options (see table 1-1, list of up to 15 characters representing options to be disabled)

OPT=*option*

Specifies optimization options. When selecting multiple options, separate values by colons. Option values are:

<u>*option*</u> <u>Description</u>

NOZEROINC Assumes that constant increment variables (CIVs) are not incremented by variables with the value 0 (default option)

ZEROINC Assumes constant increment variables (CIV) can be incremented by variables with the value 0. This option inhibits the vectorization of any DO loop in which there are CIVs of the form CIV=CIV+VARIABLE.

NOIFCON Disables optimization of conditional replacement statements of the form IF(*logical exp*)*var*=*expression* except where CFT replaces these statements with MAX/MIN intrinsic functions (default option)

PARTIALIFCON Allows CFT to optimize conditional replacement statements of the form IF(*logical exp*)*var*=*expression* if *var* is of type integer, real, or logical, and *expression* does not involve division or an external function reference. The optimization causes CFT to generate code similar to *var*=CVMGx(*expression*,*var*,*condition*). If the optimization is performed, the IF statement will not inhibit vectorization or break an optimization block. See CDIR$ NOIFCON and CDIR$ RESUMEIFCON.

option Description

FULLIFCON Allows CFT to optimize conditional replacement statements as described for PARTIALIFCON; unlike PARTIALIFCON, conditional replacement statements involving division and external functions are also optimized.

FASTMD Causes CFT to use the fast integer multiply and divide algorithms. Operands and results are limited to 46 bits; there is no overflow protection.

SLOWMD Causes CFT to generate the full 64-bit integer multiply and divide (default option)

SAFEDOREP Enables replacement of one-line DO loops with a call to a $SCILIB routine performing the same operation more efficiently (default option). Replacement does not occur when a one-line DO loop contains potential dependencies or equivalenced variables. DO-loop replacement can be disabled and reenabled within a program unit by specifying CDIR$ NODOREP and RESUMEDOREP, respectively.

FULLDOREP Enables replacement of one-line DO loops with a call to a $SCILIB routine performing the same operation more efficiently. Potential dependencies and equivalences are ignored. DO-loop replacement can be disabled and reenabled within a program unit by specifying CDIR$ NODOREP and RESUMEDOREP, respectively.

NODOREP Disables replacement of one-line DO loops with a call to a $SCILIB routine. NODOREP has no effect on vectorization of loops in the program. When OPT=NODOREP is specified, CDIR$ RESUMEDOREP is ignored.

INVMOV Enables movement of invariant code from a DO-loop body over an IF statement into the loop preamble (default option)

NOINVMOV Disables movement of any invariant code from the DO-loop body over an IF statement into the loop preamble

option Description

UNSAFEIF Enables the instruction scheduler to move instructions over a branch instruction

SAFEIF Disables instructions moving over a branch instruction. Prevents movement of a floating-point operation or subscripted reference before the branch of an IF statement put in to protect the operation (default option).

BL Enables scalar loops to be bottom loaded; operand prefetched over the branch of the loop (default option).

NOBL Disables bottom loading for scalar loops; intended to obtain correct code where the subscript for a load would be out of range if executed.

BTREG Causes CFT to allocate certain scalar variables in a program unit to T registers during the program unit existence. Some variables, such as dummy arguments, arrays, and variables named in SAVE, DATA, COMMON, or NAMELIST statements and variables named in I/O control information lists are allocated to memory.

The maximum number of T registers available for variable allocation is 25. If there are fewer than 25 local integer (including INT24), real, logical, and compiler-generated variables, the remaining T registers are used as scratch registers during expression evaluation. If there are more than 25 variables in a program unit, the first 25 variables in the source code are allocated to the T registers and the remaining variables are allocated to memory. Specific variables can be forced into T registers by declaring them part of the first 25 variables at the beginning of a program unit. Variables can be excluded from T registers by specifying their names in a SAVE statement.

option Description

BTREG (continued)

Variables allocated to T registers are not initialized upon routine entry and become undefined when a RETURN or END is executed. Subprograms depending on local variables retaining their values across calls, which violates the ANSI FORTRAN standard, do not work properly unless the SAVE statement is used. (See the SAVEALL control statement option and subsection 1.1.1, Allocation of Variables to Storage.) The INDEF option does not affect variables assigned to T registers.

Multitasked programs can use the BTREG option; however, all variables passed as arguments to a task, through TSKSTART, must be excluded from T registers (for example, named in a COMMON or SAVE statement).

NOBTREG

Causes CFT to allocate all user variables to memory. NOBTREG does not affect the allocation of compiler-generated variables to B or T registers or the use of B or T registers temporarily holding values during expression evaluation. Default is NOBTREG.

CVL

Allows CFT to generate both scalar and vector code for loops with specific ambiguous dependencies. A run-time test determines which version is used. Default is OPT=CVL.

NOCVL

Prevents CFT from generating both scalar and vector code for loops with ambiguous dependencies. Only scalar code is generated for these loops.

KEEPTEMP

Variables used as scalar temporaries will have the correct updated values when the vector DO loops execute (default option)

KILLTEMP

Variables used as scalar temporaries in vector DO loops do not have their values updated when the DO loops execute. The values of the scalar temporaries will be undefined when the DO loops terminate.

SAVEALL Compilation occurs as if a SAVE statement with an empty list was in each program unit. All user variables in a program unit are allocated to static storage. Compiler-generated variables are allocated to B or T registers.

SAVEALL overrides OPT=BTREG. SAVEALL can be specified with ALLOC=STACK, that is, CFT uses the stack only for compiler-generated variables, argument lists, etc. (See the BTREG option, the ALLOC parameter, and subsection 1.1.1, Allocation of Variables to Storage)

TRUNC=*tr* Number of bits to be truncated. Range is $0 \le tr \le 47$. Default is 0. Specifies truncation for all floating-point results. Does not truncate double-precision results, function results, or constants. Truncated bits are set to 0.

UNROLL=*r* Specifies that inner DO loops with constant limits iterating *r* times or less may use DO-loop unrolling. The maximum value of *r* is 9, and the default value is 3. DO-loop unrolling makes *n* copies of the DO loop body, where *n* is the trip count, and replaces all occurrences of the DO control variable with constants. The DO control variable is set to the same value it would have had if the DO loop did not unroll. A DO loop is not unrolled if it has labels, references to labels, external calls, or modifications to the DO control variable. A DO loop must also be small enough to make unrolling practical. UNROLL=0 turns DO-loop unrolling off.

1.1.1 ALLOCATION OF VARIABLES TO STORAGE

FORTRAN variables are allocated to storage according to the following criteria.

1. All user variables are allocated to static storage if the SAVEALL option is specified on the CFT control statement.

2. User variables appearing in a SAVE, DATA, or COMMON statement are always allocated to static storage.

3. Other user variables are allocated to a T register if the BTREG option is specified on the CFT control statement, subject to the following limitations:

   - A maximum of 24 variables in each program unit can be allocated to T registers.

   - Array variables and variables declared in CHARACTER, COMPLEX, DOUBLE, EQUIVALENCE, and NAMELIST statements are not allocated to T registers.

4. Other user variables are assigned to the stack if ALLOC=STACK is specified on the CFT control statement, or to static storage if ALLOC=STACK is not specified.

5. Compiler-generated variables are assigned to a register or to memory (to the stack if ALLOC=STACK is specified, to static memory otherwise) depending on how the variable is used. Compiler-generated variables include DO-loop trip counts, dummy argument addresses, temporaries used in expression evaluation, argument lists, and variables storing adjustable dimension bounds at entries.

Table 1-1. Compiler Options Under COS

Option	Description	Default
A	Aborts job after compilation if any program unit contains a fatal error	OFF
B	Lists beginning sequence number of each code generation block (G implies B)	OFF
C	Lists common block names and lengths listed on *ldn* after each program unit	ON
D	Lists DO-Loop Table	OFF
E	Enables recognition of compiler directive lines	ON
F	Enables FLOWTRACE option. (Also see FLOW/NOFLOW directives.)	OFF
G	Lists generated code for each program unit.[†] (See CODE/NOCODE directives)	OFF
H	Causes listing of the first statement of each program unit and error messages. All other list options are ignored or disabled.	OFF
I	Enters compiler-generated statement labels in the Symbol Table	OFF

[†] The G option lists the skeleton for the code generated for ENTRY and RETURN sequences. The actual number of B and T registers saved and the address where they are saved are not indicated. If no T registers are to be saved, the instruction to save T registers is replaced by a pass instruction.

Table 1-1. Compiler Options Under COS (continued)

Option	Description	Default
J	Causes all DO loops to be executed at least once	OFF
L	Enables recognition of output listing control directives	ON
M	Lists current machine characteristics table	OFF
N	Enters null symbols in the Symbol Table (defined but not referenced)	OFF
O	Prints a message identifying any array references with out-of-bounds subscripts found during execution[†]. Enables the BOUNDS compiler directives.	OFF
P	Allows double precision. Setting OFF=P causes at compile time: 1. All double-precision declaratives to be treated as real; 2. Double-precision functions to be changed to the corresponding single-precision functions; 3. Double-precision constants to be converted as double-precision and truncated to real; 4. D's in FORMAT statement to be changed to E's.	ON
Q	Aborts compilation when 100 fatal error messages counted	ON
R	Rounds result on multiply operations	ON
S	Lists FORTRAN source code	ON
T	Lists the Symbol Table after each program unit	ON
U	Enables recognition of INTEGER*2 declaration. OFF=U processes variables declared INTEGER*2 as 64-bit integers.	ON

[†] Bounds checking inhibits many optimizations CFT normally performs.

Table 1-1. Compiler Options Under COS (continued)

Option	Description	Default
V	Vectorizes inner DO loops	ON
W	Compiles all floating-point operations as return jumps to user-supplied external routines.† (See table 1-4.)	OFF
X	Lists the Symbol Table with cross references after each program unit (X overrides T)	OFF
Z	Writes the Debug Symbol Table on $BLD	OFF

† The W option has no effect on complex or double-precision arithmetic, intrinsic functions, or expressions in a DATA or PARAMETER statement.

1.2 THE UNICOS CFT COMMAND

Under UNICOS, the CFT compiler is loaded and executed when a CFT command is encountered.

Format:

```
cft [-a alloc] [-b binfile] [-c calfile] [-d oplist]
[-e oplist] [-i intlen] [-m mlev] [-o options] [-t trunc]
[-u unroll] [-v vmsgs] [-A aids] [-C type,characteristics]
[-E errfile] [-M maxblock] [--] filename
```

Options can be in any order, but must precede the *filename* argument.

-a *alloc* Specifies memory allocation scheme for entities in memory. *alloc* can be one of the following.

alloc	Description
static	All memory is statically allocated; a stack is not used (default).
stack	Read-only constants and entities in a DATA or SAVE statement, or a common block are statically allocated. All other entities are allocated on the stack.

+b *binfile* Binary output file name; default is *filename*.o.

-c *calfile* CAL listing file name; default is no file. If -e C is specified, *filename*.s is used.

-d *oplist* List of options to be disabled (see table 1-2)

-e *oplist* List of options to be enabled (see table 1-2)

-i *intlen* Specifies length of integers, 64 or 24 bits; default is 64 bits.

-l *listfile* Source listing file name; default is no file. If -e L is specified, *filename*.l is used.

-m *mlev* Highest message level to be suppressed. Fatal errors are not suppressed.

mlev	Severity
1	COMMENT
2	NOTE
3	CAUTION (default)
4	WARNING
5	ERROR

-o *options* List of optimization options to be enabled or disabled. Options must be separated by commas; only one option in each group can be selected.

bl | nobl
bl enables scalar loops to be bottom loaded (default)
nobl prevents bottom loading of scalar loops

btreg | nobtreg
btreg causes CFT to allocate specific scalar variables in a program unit to T registers
nobtreg causes CFT to allocate all user variables to memory. Does not affect the allocation of compiler-generated variables to B or T registers or the use of B or T registers temporarily holding values during expression evaluation. (default)

cvl | nocvl
cvl enables compilation of conditional vector loops (default)
nocvl disables compilation of conditional vector loops

fastmd | slowmd

fastmd enables fast 46-bit integer multiply and divide
slowmd enables full 64-bit integer multiply and divide (default)

invmov | noinvmov

invmov enables movement of invariant code from DO loops (default)
noinvmov disables movement of invariant code from DO loops

keeptemp | killtemp

keeptemp updates scalar temporary variables in DO loops (default)
killtemp does not update scalar temporary variables in DO loops. The variable values will be undefined when the DO loops terminate.

nodorep | safedorep | fulldorep

nodorep disables replacement of one-line DO loops with a call to a $SCILIB routine
safedorep enables replacement of one-line DO loops (without potential dependencies or equivalenced variables) with a call to a $SCILIB routine performing the same operation more efficiently (default)
fulldorep enables replacement of one-line DO loops (potential dependencies and equivalencies are ignored) with a call to a $SCILIB routine performing the same operation more efficiently

noifcon | partialifcon | fullifcon

noifcon disables optimization of conditional replacement statements (default) partialifcon enables optimization of conditional replacement statements not involving division and external functions
fullifcon enables optimization of conditional replacement statements involving division and external functions

safeif | unsafeif

safeif prevents instructions from moving over a branch instruction (default)
unsafeif enables instructions to move over a branch instruction

zeroinc | nozeroinc

zeroinc allows constant increment variables to be incremented by zero variables
nozeroinc prevents constant increment variables from being incremented by zero variables (default)

-t *trunc* Number of bits truncated for floating-point results. Truncated bits are zeroed. Range is 0≤*nn*≤47; default is 0.

-u *unroll* Specifies that inner DO loops with constant limits iterating *unroll* times or less may use DO-loop unrolling. The maximum value of *unroll* is 9; default is 3. -u 0 turns off DO-loop unrolling.

-v *lmmsgs* Enables the LOOPMARK utility. LOOPMARK draws brackets delimiting DO loops in the source code listing. Each loop is marked with a symbol indicating the type of loop (see subsection 1.5.9, Table of Loops Encountered). If -v msgs is specified, a message is printed on the line following each inner DO loop that did not vectorize, indicating the reason that the loop did not vectorize.

The *lmmsgs* parameter indicates whether messages are to be printed by the LOOPMARK utility. This parameter is required.

lmmsgs	Description
msgs	Messages are printed for all inner DO loops that did not vectorize
nomsgs	No messages are printed

-v *lmmsgs* implies -e d. LOOPMARK is disabled by default.

-A *aids* Number of vectorization inhibition messages.

aids	Number of Messages
LOOPNONE	None
LOOPPART	Three per compiler block; 100 messages per compilation (default).
LOOPALL	All

-C *type,characteristics*
Specifies mainframe type and optional mainframe characteristics running the generated code; default is the machine running CFT and its characteristics.

type	Mainframe Type
cray-1	Generates code for CRAY-1/A Computer Systems
cray-1a	Generates code for CRAY-1/A Computer Systems
cray-1b	Generates code for CRAY-1/B Computer Systems
cray-1m	Generates code for CRAY-1 M Computer Systems

type	Mainframe Type
cray-1s	Generates code for CRAY-1 S Computer Systems
cray-xmp	Generates code for CRAY X-MP Computer Systems
cray-x1	Generates code for single-processor CRAY X-MP Computer Systems
cray-x2	Generates code for dual-processor CRAY X-MP Computer Systems
cray-x4	Generates code for four-processor CRAY X-MP Computer Systems

characteristics	Description
[no]avl	Target machine does/does not have two vector logical functional units
[no]bdm	Target machine does/does not have bidirectional memory
[no]ci	Target machine does/does not have compressed index hardware
[no]ema	Target machine does/does not have extended memory addressing
[no]gs	Target machine does/does not have gather/scatter hardware
[no]cigs	Target machine does/does not have compressed index gather/scatter hardware (cigs implies ci and cs, nocigs implies noci and nogs)
ibufsize=*words*	Instruction buffer size in words (16 or 32)
memspeed=*cps*	Memory speed in clock periods
[no]vpop	Target machine does/does not have a vector population count functional unit
[no]vrecur	Target machine does/does not have vector recursion

-E *errfile* Alternate error listing file; default is stderr.

-M *maxblock* Length of code block being optimized or vectorized; default is 2310. Values greater than 2310 may increase optimization and internal compiler errors. -M 1 eliminates optimization or vectorization.

Table 1-2. Compiler Options Under UNICOS

Option	Description	Default
A	Prints non-ANSI messages at compile time	-d A
B	Creates a binary object file	-e B
C	Creates a pseudo CAL file	-d C
D	Writes sequence number labels for executable FORTRAN statements to the Debug Symbol Table; forces -e iz and -M 1.	-d D
E	Creates an error listing file; default file name is stderr.	-d E
I	Sets stack variables to an undefined value.	-d I
L	Creates a listing file	-d L
S	Compiles as if a SAVE statement with an empty list were in each program unit; overrides -o btreg.	-d S
a	Aborts job after compilation if any program unit contains compilation errors	-d a
b	Lists beginning sequence number of each code generation block	-d b
c	Lists common block names and lengths after each program unit	-e c
d	Lists the DO-loop Table	-d d
e	Enables recognition of compiler directive lines	-e e
f	Enables the FLOWTRACE utility	-d f
g	Lists generated code for each program unit	-d g
h	Lists the first statement of each program unit and error messages; disables or ignores all other *oplist* options.	-d h

† The g option lists the skeleton for the code generated for ENTRY and RETURN sequences. The actual number of B and T registers saved and the address where they are saved are not indicated. If no T registers are to be saved, the instruction to save T registers is replaced by a pass instruction.

Table 1-2. Compiler Options Under UNICOS (continued)

Option	Description	Default
i	Enters compiler-generated statement labels in the symbol table	-d i
j	Causes all DO loops to execute at least once	-d j
l	Enables recognition of output control directives	-d l
n	Enters null symbols in symbol table (defined but not referenced)	-d n
o	Prints messages identifying array references with out-of-bounds subscripts found during execution.† Enables BOUNDS compiler directives.	-d o
p	Allows use of double precision	-e p
q	Aborts compilation when 100 fatal error messages have been issued	-e q
r	Rounds result on multiply operations	-e r
s	Lists FORTRAN source code	-e s
t	Lists symbol table after each program unit	-e t
u	Allows recognition of INTEGER*2 declaration	-e u
v	Vectorizes inner DO loops	-e v
w	Compiles all floating-point operations as return jumps to user-supplied external routines.†† (See table 1-4.)	-d w
x	Lists symbol table with cross reference after each program unit (overrides t)	-d x
z	Writes the Debug Symbol Table	-d z

† Bounds checking inhibits many optimizations CFT normally performs.

†† The w option has no effect on complex or double-precision arithmetic, intrinsic functions, or expressions in a DATA or PARAMETER statement.

1.3 COS ERROR MESSAGES DURING PROGRAM EXECUTION

The executing program calls on operating system routines to accomplish its mathematical, input/output, and utility operations. These routines are not executed during compilation of the program. They are loaded from the system or user libraries and linked to the program by a loader (SEGLDR or LDR). When used, the routines respond to programming and/or equipment discrepancies by placing messages in the jobfile and in the user output file or dataset. Some of these discrepancies also cause the job to abort. The COS error messages are described in the COS Message Manual, publication SR-0039; the UNICOS error messages are described in the UNICOS Kernel Error Message Manual, publication SR-2015.

1.4 INPUT TO CFT

CFT, when initiated, seeks two types of information: the program to be compiled and instructions on controlling the compilation.

A FORTRAN program to be compiled by CFT must be specified in a form using the ASCII character codes listed in appendix A. The result is *source code*.

Other information required by CFT to complete its operations is provided by the operating system and compiler directives specified in the program being compiled. (See a description of compiler directives later in this section.) This information includes identification of the input dataset containing the source and identification of datasets receiving binary and listable output from CFT during compilation. The CFT options to use are specified in the CFT control statement or command.

1.5 OUTPUT FROM CFT

Relocatable binary output is written on the dataset specified on the B parameter under COS, or the file specified in the -b parameter under UNICOS, in a format suitable for input by SEGLDR or LDR, one record per program unit. When requested, the loader loads and links this file plus routines required from the system or user libraries. CFT optionally produces a file or dataset containing the following.

- A source statement listing
- Error messages and their severity
- Tables of statement numbers, names encountered, parameters encountered, block names and their octal lengths, external names, and loops encountered

The CFT control statement or command, and the CFT compiler directives allow the user to control this output and specify the receiving file or dataset. Listable output is divided into pages.

1.5.1 PAGE HEADER LINES

Each page of listable output contains a header line with the following information.

- The name of the program unit (except for the first page for each program unit)
- The current page number within the program unit
- The truncation count, if nonzero (see the TRUNC parameter on the CFT control statement, earlier in this section)
- A list of compiler options currently turned on (see tables 1-1 and 1-2)
- The date and time compilation began
- The CFT revision level and assembly date
- The global page number

1.5.2 SOURCE STATEMENT LISTINGS

The source statement listing is generated when the S list option is selected. The listing is a record of all FORTRAN statements comprising the program as they are sequentially read and interpreted from the source input dataset. A sequence number is listed for each statement identifying its position in the program. A line number for each line is listed to the left of the sequence number. Continuation lines and comments are separate lines but not separate statements. Errors encountered during a statement compilation are flagged by lines subsequent to that statement or recorded at the end of the source statement listing.

1.5.3 BLOCK BEGINS MESSAGES

CFT divides program units into smaller units called blocks, which are the basic units optimized by CFT. Specifying ON=B or ON=G in the CFT control statement produces a BLOCK BEGINS message for each block, listing the sequence number and relative program address of the beginning of each block. If ON=B or ON=S is selected, the message VECTOR LOOP BEGINS is listed for blocks with a vector loop.

A vector block can begin several lines before a vectorized DO loop. (Only an innermost loop is a candidate for vectorization.) Results calculated in this loop preamble are used by the optimizer in the loop. Debugging instructions should be inserted between blocks to avoid altering the generated code of the block being tested.

1.5.4 TABLE OF STATEMENT NUMBERS

The table of statement numbers can be in a short form, excluding cross reference information (T option) or in a long form, including cross reference information (X option). In either case, the table lists all statement numbers used in the program unit, followed by a suffix indicating whether the number is inactive (SN), a FORMAT statement, or undefined (UNDEF*). For active statements, the relative address of the beginning of the statement is given.

Statement numbers are internally generated for logical IF statements, implied-DO statements, and ENTRY statements. A 5-digit number in sequence starting with 00001 is generated with leading zeros present and significant.

For DO loops, two internally generated statement labels are created, one at the top of the loop (the reloop point) and one after loop termination control (the zero trip point). These labels are generated by suffixing the loop terminal number with letters A, B, etc., taken in pairs.

By default, internally generated numbers are not listed in this table. Specifying ON=I lists them.

1.5.5 TABLE OF NAMES ENCOUNTERED

This table has a short form, excluding cross reference information (T option) and a long form, including cross reference information (X option). In either case, the following fields of information are presented.

- Address
- Name
- Type
- Main usage
- Block

1.5.5.1 Address field

A value in this field is octal and is either an address relative to the beginning of the program, the local stack area, or a named common block; or it is a B or T register number.

1.5.5.2 Name field

The name field contains an alphabetized list of all symbolic names specified in the program unit. If the PROGRAM statement is omitted from the executable program, CFT identifies the main program with the name $MAIN.

1.5.5.3 Type field

The type field gives the type of array, variable, or program unit and can contain the following values.

Type	Significance
C	Complex
D	Double precision (prefix to other types, if they are double)
I	Integer (64 bits)
I'	Integer (24 bits)
L	Logical
R	Real
CH	Character
none	Typeless function or subroutine

If the item is defined or declared but not used, the type code is preceded by *.

1.5.5.4 Main usage field

An entry in this field describes the use of the corresponding symbolic name and can contain the following.

Use	Significance
*n*D. EQ. ARRAY	*n*-dimensional array in EQUIVALENCE
*n*DIM ARRAY	Array with *n* dimensions ($1 \leq n \leq 7$)
ENTRY	Entry
EQUIVALENCE	Variable or array in EQUIVALENCE
EXTERNAL	External function or subroutine
INTRINSIC	Intrinsic function
PARAMETER	Symbol appears in PARAMETER statement
ST. FUNCTION	Arithmetic statement function
UNDEF EQUIV	Variable or array appears in EQUIVALENCE statement but does not appear on the left side of an assignment operator or in a DATA statement
UNDEFINED ***	Variable or array never defined
VARIABLE	Simple variable
#T-REG	Simple variable assigned to a T register instead of memory

1.5.5.5 Block field

The block field identifies the common block containing a variable or array. If no common block name appears, the variable or array is local to the program unit. If the name #ST appears, the variable or array is assigned to stack storage. If the name #T-REG or #B-REG appears, the variable is permanently assigned to a register.

If the common block name is preceded by a %, the common block is declared a task common block. All variables declared in a task common block are assigned to the task common block heap. For more information, see task common blocks in section 4.

If the symbol is a dummy argument to the subroutine or function, the field contains the characters DUM.ARG. and the address field contains the dummy argument number.

If the symbol is a pointee, the field contains the characters POINTEE and the address field contains the pointee number.

1.5.6 TABLE OF PARAMETERS ENCOUNTERED

This table contains the names and values of the symbolic constants and is generated only when cross-reference information (X or T option) is requested.

1.5.7 TABLE OF BLOCK NAMES AND LENGTHS IN OCTAL

This table lists the name of each block referenced in the program unit preceded by its word length in octal. The C list option controls this table.

The program block is the first block listed, and it has the same name as the program unit being compiled. Pound blocks #TB, #CL, and #ST follow the program block, in that order. The program block and pound blocks are created by the compiler and contain code and static data to execute the compiled program unit. The compiler uses the pound blocks to store the following entities:

#TB Temporary variables local to a code block; space in #TB is reused from block to block. #TB is used in static mode only.

#CL In static mode, #CL holds argument lists and the space into which passed-in argument lists are copied for multiple-entry routines. In stack mode, #CL holds argument list headers only; the headers are built as compile-time constants.

#ST All stacked entities except the B/T save area and variables initialized in DATA statements. #ST is not used in static mode. #ST contains all entities that are in #TB in static mode, and most entities that are in #CL in static mode. In generated code listings, the offset field gives the offset from the run-time address in B03 to the first word of the stacked entity.

1.5.8 STATIC SPACE TABLE AND STACK SPACE TABLE

The Static Space Table and Stack Space Table follow the Block Name and Length Table and describe space usage of the compiled program units. The C list option controls printing of the Static and Stack Space Tables.

The Static Space Table describes how space is used in program and pound blocks. The Static Space Table has the following entries.

B SAVE Number of words reserved to hold values in B registers; one greater than the number of B registers used by the generated code.

T SAVE Number of words reserved to hold values in T registers; equal to the number of T registers used by the generated code.

CONSTANTS Number of words reserved to hold read-only constants.

VARIABLES Number of words reserved for local variables, including variables declared by the user and variables created by the compiler.

TEMPORARIES
Number of words reserved to hold temporary variables. Temporary variables are compiler-generated variables and are usually reused from code block to code block.

CODE Number of words occupied by generated code.

TOTAL Number of static words required to execute the compiled program unit; equal to the sum of the preceding items and also the sum of the lengths of program and pound blocks.

If stack mode is requested (see the subsection on the CFT control statement for a description of the ALLOC parameter), the table describing stack space usage in the program units is printed. The Stack Space Table has the same format as the Static Space Table and contains the number of words of stack space required for the B-register save area, T-register save area, stack-based variables, and stack-based temporaries. The amount of stack space needed by the program unit is also printed.

1.5.9 TABLE OF EXTERNAL NAMES

This table is generated only when cross reference information (X option) is requested and contains external names and source program references.

1.5.10 TABLE OF LOOPS ENCOUNTERED

This table presents the following fields of information relevant for program loops when the D list option is selected.

Label Statement number ending the loop

Index DO-loop index

From Beginning source line number

To Ending source line number

Address Parcel address of loop start (blank if no loop is generated)

Length Octal number of words of code generated for the loop body. When a loop is not generated for a DO loop, such as a short vector loop, the word INLINE appears in place of the length.

Properties

Properties of the DO loop, indicating whether the DO loop was vectorized, unrolled, or replaced with a call to a $SCILIB routine. This portion of the table contains messages for each DO loop that did not vectorize, indicating the reason the DO loop did not vectorize (see appendix D for descriptions of these messages).

Rp - REPLACED
This DO loop was replaced by a call to an optimized $SCILIB routine.

Ur - UNROLLED
This DO loop was unrolled.

V - VECTOR
This DO loop was vectorized.

V2 - TWO SUBSEQUENT AMBIGUOUS VECTORS
Two vector versions of this DO loop were generated. A run-time test determines which version is executed.

Vc - CONDITIONAL VECTOR
Scalar and vector versions of this loop were generated. A run-time test determines which version is executed.

Vs - SHORT VECTOR
A vector loop with a known iteration count of at most 64 was generated. This loop has less overhead than other vector loops, and will execute slightly faster.

1.5.11 CROSS-REFERENCE INFORMATION

Cross-reference information is optionally included in the list output with the selection of the X list option. When requested, the table of statement numbers, the table of names encountered, and the table of external names include the source program references. These references are keyed to the source listing line numbers. The following codes are used in these references.

Code	Significance
A	Used in FORTRAN ASSIGN statement
D	Defined in declarative statement
E	Statement number ending a DO loop
I	Index of a DO or implied DO loop
J	Statement number used in transfer
L	Source line of a statement number
N	Name used as a DO loop parameter
P	Used in CALL/FUNC call or array reference
R	Format used in READ statement
S	Stored so contents can be changed
U	Name used in executable statement
W	Format used in WRITE statement
?	Ten or more references to symbol

1.5.12 MESSAGES

Up to six levels of messages are produced by CFT, depending on the E and ANSI parameters on the CFT control statement. (See appendix D for details of messages.)

1.5.13 PROGRAM UNIT PAGE TABLE

If more than one program unit is compiled, CFT prints a sorted table of the names of the units compiled, listing the beginning global page number of each program unit. This table appears at the end of CFT's output.

1.6 COMPILER DIRECTIVES

Compiler options selected by the user in the CFT control statement or command (see descriptions earlier in this section) establish particular methods for application throughout the compilation of all related FORTRAN program units. *Compiler directives* encountered in the program units being compiled can change or reinstate this set of methods. Certain other compiler actions are enabled and disabled only by compiler directives.

The COS CFT compiler option ON=E or the UNICOS CFT compiler option -e e (enable compiler directives) option must be in effect in order for compiler options to be recognized by CFT (this option is enabled by default in both operating systems). Otherwise, the lines containing compiler options are treated as comment lines.

1.6.1 USING COMPILER DIRECTIVE LINES

A *compiler directive line* contains the characters CDIR$ in columns 1 through 5. Column 6 of the initial line must be blank or contain the character 0. Columns 7 through 72 of the initial line contain zero or more compiler directives separated by commas. If the compiler directive has a list associated with it, no other compiler directive can appear on the same line. Spaces can precede, follow, or be embedded within a compiler directive. Columns 73 through 96 can be used for any purpose. Continuation of compiler directive information beyond a single line can be accomplished by one of the following methods.

- Enter any character except a blank or zero in column 6 of up to 19 subsequent lines.

- Enter the characters CDIR$ in columns 1 through 5 of all lines in the sequence.

Comment or blank lines cannot occur within a continued CDIR$ sequence. The first non-CDIR$ line terminates the CDIR$ continuation sequence.

The character C in column 1 identifies lines as comment lines to all but the Cray FORTRAN Compiler. This feature maintains the transportability of programs using compiler directives.

Compiler directive lines are listed in the source statement listing.

1.6.2 CATEGORIES OF COMPILER DIRECTIVES

CFT provides the following categories of compiler directives.

- Listable output control
- Vectorization control
- Integer control
- Multiply/divide control
- Flow trace
- Scheduler
- Dynamic common block
- Array bounds checking
- Optimization
- Debugging
- Roll/unroll

1.6.2.1 Listable output control directives

Following are the listable output control directives.

- EJECT
- LIST
- NOLIST
- CODE
- NOCODE

The listable output control directive (compiler option L under COS; compiler option -e l under UNICOS) must be on in order to cause recognition of this set of compiler directives.

EJECT directive - A compiler directive line containing an EJECT directive is printed as the last line of the current page of source statement listing. If the EJECT directive is contained in a continuation set of compiler directive lines, the last of these becomes the last line of the page. In either case, a new page begins. The EJECT directive has no effect if production of the source statement listing has been suppressed.

Format:

```
 _________
|         |
|  EJECT  |
|_________|
```

LIST directive - The LIST directive causes the production of a source statement listing or is ignored if one is already being produced. The LIST directive also restores the other list options specified on the CFT control statement.

Format:

```
 ________
|        |
|  LIST  |
|________|
```

NOLIST directive - The NOLIST directive suppresses the production of all listable output. If no listable output is being produced, the NOLIST directive is ignored.

Format:

```
 __________
|          |
|  NOLIST  |
|__________|
```

CODE directive - The CODE directive produces CFT-generated code listings if previously suppressed by a listing directive or by the CFT compiler option OFF=G or OFF=L under COS, or by the CFT compiler option -d C or -d 1 under UNICOS. Code is listed for the optimization block where the CODE directive occurs. The listing continues until a NOCODE directive is encountered or until superseded by another LIST directive.

Format:

```
 ________
|        |
| CODE   |
|________|
```

NOCODE directive - The NOCODE directive suppresses the production of a CFT-generated code listing. The NOCODE directive takes effect at the beginning of the next optimization block, and no generated code is produced until a CODE directive is encountered. If no CFT-generated code listings are being produced, the NOCODE directive is ignored.

Format:

```
 __________
|          |
| NOCODE   |
|__________|
```

NOTE

The CODE and NOCODE directives apply on an optimization block basis instead of a program unit basis.

1.6.2.2 Vectorization control directives

The vectorization control directives require the COS ON=V or the UNICOS -e v CFT compiler option. Following are the vectorization control directives.

- VECTOR
- NOVECTOR
- NORECURRENCE
- IVDEP
- IVDMO
- VFUNCTION
- NEXTSCALAR
- SHORTLOOP

VECTOR directive - The VECTOR directive causes the compiler to resume its attempts to vectorize inner DO loops if such attempts were suppressed or modified by another vectorization directive. After a VECTOR directive is specified, DO loops with a known iteration count of one are executed in scalar mode; those with an iteration count of two or more or with an unknown iteration count are executed in vector mode.

DO loops containing recurrences are affected only by the NORECURRENCE directive. (See NOVECTOR and NORECURRENCE directives.)

The VECTOR directive takes effect at the next DO loop and applies to the rest of the compilation unless it is superseded by another vectorization directive.

Format:

```
 _________
|         |
| VECTOR  |
|_________|
```

NOVECTOR directive - The NOVECTOR directive suppresses the compiler's attempts to vectorize inner DO loops. The NOVECTOR directive takes effect at the next DO loop and applies to the rest of the compilation unit unless it is superseded by another vectorization directive.

Format:

```
 ________________
|                |
| NOVECTOR[=n]   |
|________________|
```

n An integer constant or a previously defined integer parameter in the range 0 to 64.

Generally, vector loops are faster than scalar loops, but because more preparation time is needed for vector registers than for scalar registers, DO loops executed a few times may be executed faster in scalar mode than in vector mode.

If the NOVECTOR directive is not in effect, the compiler causes vectorizable loops to execute in scalar mode if the DO-loop iteration count is less than 2.

If the NOVECTOR directive is in effect and *n* is not specified, DO loops are executed in scalar mode. If *n* is specified, DO loops with an iteration count greater than *n* are executed in vector mode, if possible. Those with an iteration count of *n* or less are executed in scalar mode.

The determination of scalar versus vector mode is made during compilation. If the value of any of the DO parameters cannot be determined during compilation (that is, if an expression contains anything other than constants or parameters), the loop is executed in vector mode unless vectorization is inhibited for some other reason.

If attempted vectorization of inner DO loops is not specified by CFT control statement option, the NOVECTOR directive is ignored.

NOTE

Both VECTOR and NOVECTOR directives can be specified in a single program unit.

NORECURRENCE directive - The NORECURRENCE directive causes DO loops containing recurrences to be executed in scalar or vector mode. The NORECURRENCE directive takes effect at the next DO loop and applies to the rest of the compilation unit unless it is superseded by another vectorization directive.

Format:

```
 ___________________
|                   |
| NORECURRENCE[=n]  |
|___________________|
```

n Integer constant or a previously defined integer parameter in the range 0 to 64 (default is 2).

An assignment statement is a recurrence relation if the right side involves a variable just computed. The CFT compiler can vectorize DO loops containing most recurrence relations of scalar variables. The following recurrence relations can be vectorized.

S=S+*e*

S=S**e*

S A scalar variable

e Any expression not inhibiting vectorization (see section 5 for a description of FORTRAN expressions)

Because more preparation time is needed for vector registers than for scalar registers, DO loops executed only a few times are executed faster in scalar mode than in vector mode.

If *n* is not specified, DO loops containing recurrences are executed in scalar mode. If *n* is specified, DO loops with a known iteration count greater than *n* are executed in vector mode; those with a known iteration count of *n* or less are executed in scalar mode.

The determination of scalar versus vector mode is made during compilation. If the value of any of the DO parameters cannot be determined during compilation (that is, if an expression contains anything other than constants or parameters), the loop is executed in vector mode unless vectorization is inhibited for some other reason.

If the NORECURRENCE directive is omitted, the CFT compiler executes vectorizable loops with recurrences in vector mode if the iteration count is known to be 3 or greater. Generally, vector mode is faster than scalar mode for DO loops with recurrences. If attempted vectorization of inner DO loops is not specified by a CFT control statement option, the NORECURRENCE directive is ignored.

<u>IVDEP directive</u> - The IVDEP directive is specified before a DO statement causing the compiler's attempts to vectorize the corresponding DO loop to ignore any vector dependencies, but any dependencies must be processed in source text order. The IVDEP directive affects only the single innermost DO loop it directly precedes. Conditions other than vector dependencies can inhibit vectorization even if an IVDEP directive is specified.

Format:

```
 ________
|        |
| IVDEP  |
|________|
```

<u>IVDMO directive</u> - The IVDMO directive is specified before a DO statement causing the compiler's attempts to vectorize the corresponding DO loop to ignore any vector dependencies and memory overlaps. Conditions other than vector dependencies and Bidirectional Memory hazards can inhibit vectorization whether or not an IVDMO directive is specified.

Format:

```
 ________
|        |
| IVDMO  |
|________|
```

<u>VFUNCTION directive</u> - The VFUNCTION directive declares that a vector version of an external function exists.

Format:

```
 _____________________
|                     |
| VFUNCTION f[,f]...  |
|_____________________|
```

f   The symbolic name of a vector external function

The function *f* must be written in CAL and must use the call-by-value sequence. Because CFT prefixes and suffixes the name with % as part of the calling sequence, *f* must be limited to six characters. (See the Macros and Opdefs Reference Manual, CRI publication SR-0012, for details on CFT linkage macros.) *f* must not be the name of a dummy procedure.

VFUNCTION arguments must be either vectorizable expressions or scalar expressions. If the argument list contains both scalar and vector arguments in a vector loop, the scalar arguments are broadcast into the appropriate vector registers. If all arguments are scalar or the reference is not in a vector loop, the function *f*% is called with all arguments passed in S registers. Functions named in a VFUNCTION list must not have side effects. (CDIR$ VFUNCTION implies NO SIDE EFFECTS; the names of functions appearing in the VFUNCTION directive need not appear in a CDIR$ NO SIDE EFFECTS list.) Registers are used for argument transmission and, therefore, no more than seven single-word items or three double-word items can be passed by a call. One register passes each single-word argument and two registers pass each double-word argument; these can be mixed in any order with a maximum of seven required registers.

The VFUNCTION directive must precede any statement function definitions or executable statements in a program. If the names of functions in a VFUNCTION directive also appear in an EXTERNAL declaration, the EXTERNAL declaration must precede the VFUNCTION directive.

A VFUNCTION function should receive inputs from its argument list. The VFUNCTION function should not change the value of its arguments or variables in common blocks and should not reference variables in common blocks which are also used by a program unit in the calling chain.

NEXTSCALAR directive - The NEXTSCALAR directive, specified in advance of a DO statement, causes only that DO loop to be executed in scalar mode. Vectorization is inhibited.

Format:

```
 ______________
|              |
| NEXTSCALAR   |
|______________|
```

SHORTLOOP directive - The SHORTLOOP directive, specified in advance of a DO statement, states that the succeeding DO loop will be executed at least once and at most 64 times, allowing CFT to generate special code for the succeeding DO loop. This directive may decrease execution time because it eliminates the run time tests that determine if a vectorized DO loop has been completed. Using this directive before a zero-iteration DO loop or a DO loop that should be executed more than 64 times produces indeterminate results.

Format:

```
SHORTLOOP
```

1.6.2.3 Integer control directives (INT24, INT64)

The specification of INT24 or INT64 in a program unit causes all variables and arrays named in its argument list to be identified as entities of type integer. When INT24 is specified, the integers provide 24-bit (instead of the usual 64-bit) values when referenced. The INT24 directive is not a Cray FORTRAN language statement. It must, however, be specified in a program unit according to the rules for specifying type statements.

Formats:

```
INT24 v[,v...]
INT64 v[,v...]
```

INT24 Specifies a 24-bit integer data type,

INT64 Specifies a 64-bit integer data type, and

v Is the symbolic name of a variable or array. If *v* is omitted, the INT24 or INT64 directive implicitly types all variables beginning with the letters I-N as short or long integers.

Use caution with INT24 variables. The INT24 directive is intended to allow the programmer to force CFT to use the fast 24-bit registers for performing some arithmetic operations. When a 24-bit variable is used as an argument to a function or subroutine, the 24-bit variable is sign extended and treated as a 64-bit variable. Overflow on values greater than $2^{23}-1$ is never detected. The INT64 directive overrides a default specification of INT24.

1.6.2.4 Multiply/divide directives (FASTMD, SLOWMD)

The two multiply/divide directives are FASTMD and SLOWMD. When the FASTMD directive is specified, the fast 46-bit integer multiply and divide algorithms are used in the current block. When the SLOWMD directive is specified, the normal 64-bit integer arithmetic is used in the current block. When the 46-bit integer arithmetic is used, the integer multiply or divide result has only 46 bits of accuracy and there is no overflow protection for operands or results greater than 46 bits.

1.6.2.5 Flow trace directives (FLOW/NOFLOW)

Flow trace directives print a summary listing the following information about each subroutine in a program. This summary is written to dataset $OUT under COS, or to file *filename*.l under UNICOS, where *filename* is the name of the CFT source file.

- The time spent in the subroutine
- The percent of the total time spent in the subroutine
- The number of times the subroutine was called
- The average time per call spent in the subroutine
- A list of the first 14 routines called by the subroutine
- A list of the first 14 routines that call the subroutine
- Subroutine linkage overhead, which consists of the following information

  Total number of subroutine calls

  Total amount of B and T register usage and number of arguments passed for the entire job

  Minimum, maximum, and average number of B and T registers used and arguments passed for each routine traced. (Averages are weighted by calling frequency.)

  Time spent saving and restoring the B and T registers

  Time spent in the calling sequence and the approximate time spent in the flow trace routine. (The time is listed in number of clock cycles, number of seconds, and percent of total job time.)

Flow trace enable/disable - Flow trace is enabled under COS by using ON=F on the CFT statement or under UNICOS by specifying -e f on the CFT command, or by using a CDIR$ FLOW directive in the source program. A matching CDIR$ NOFLOW disables flow trace. To be useful, the CDIR$ FLOW or NOFLOW directives must come after an END statement and before the next PROGRAM, SUBROUTINE, or FUNCTION statement. It is often wise to disable flow tracing for small, frequently called routines because the flow trace overhead time can be much greater than the actual subroutine execution time. As currently implemented, the main program where flow trace is enabled must contain a PROGRAM statement.

When flowtrace is enabled, a flow trace summary is listed either after the END statement in the main program is executed or after a STOP statement in the routine being traced is executed. Programs that terminate with CALL EXIT, CALL ABORT, etc., must be modified to use flow trace.

Time spent in a lower level called routine for which flow trace is enabled is not counted as time spent in the calling routine. Time spent in library routines (SIN, PRINT, CFFT, etc.) or in any routine for which flow trace is not enabled is counted as time spent in the calling routine. However, such routines are not listed in the summary.

FLODUMP utility - FLODUMP provides, upon request, a dump of the flow trace tables when a program aborts with flow trace active. FLODUMP dumps the tables in flow trace format. FLODUMP is invoked by specifying ON=F in the COS CFT control statement, or -e f in the UNICOS CFT command, and by including the FLODUMP control statement or command.

The following example shows the use of the FLODUMP control statement in conjunction with the CFT, EXIT and DUMPJOB control statements under COS.

Example:

```
    JOB, ... .
    CFT,ON=F.
    SEGLDR,GO.
    EXIT.
    DUMPJOB.
    FLODUMP.
        .
        .
        .
```

See the COS Version 1 Reference Manual, publication SR-0011, for details of the FLODUMP control statement.

The user can select one or more of the following options.

- SETPLIMQ
- ARGPLIMQ
- FLOWLIM

SETPLIMQ - This option enables the flow trace routine to print a line on the output listing file for every CALL or RETURN statement executed, listing the following information.

- Routine name
- Calling routine name
- Job time
- Time the routine is entered
- Time spent in the routine
- Time the routine returns

Because this option can generate a large volume of output, it must be explicitly requested at run time as follows.

```
    CALL SETPLIMQ(KOUNT)
```

The value of KOUNT specifies the number of trace lines printed. Since one line is produced for each CALL and each RETURN, KOUNT should be set to twice the number of CALL statements for which flow trace is desired.

In effect, each CALL and each RETURN statement is given a sequence number at run time. Each subsequent CALL or RETURN statement whose sequence number is less than ABS(KOUNT) causes a printout. CALL or RETURN statements executed before the CALL SETPLIMQ(KOUNT) count toward the line limit but do not generate any output. In general, CALL SETPLIMQ(KOUNT) is one of the first executable statements in a program.

ARGPLIMQ - ARGPLIMQ enables the flow trace routine to list the subroutine arguments for the next ABS(KOUNT) calls. This option must be explicitly requested at run time as follows.

```
    CALL ARGPLIMQ(KOUNT)
```

This option can be called only once in a program.

FLOWLIM - FLOWLIM enables the flow trace routine to limit the number of traced subroutines to the next ABS(KOUNT) subroutines. After this limit is reached, the flow trace summary is printed. Further calls to FLOWENTR and FLOWEXIT result in a return to the user's calling subroutine, thus reducing overhead time. In effect, the call to FLOWLIM turns off the flow trace option after the limit is reached.

The FLOWLIM option must appear before any subroutine calls and it must be explicitly requested at run time as follows.

```
    CALL FLOWLIM(KOUNT)
```

KOUNT=0 traces all subroutines.

1.6.2.6 Scheduler directives

The list of scheduler directives follows.

- UNSAFEIF
- SAFEIF
- BL
- NOBL

UNSAFEIF/SAFEIF directives - The UNSAFEIF and SAFEIF directives enable or disable movement of code past the branch of an IF statement for a block of code, respectively. If UNSAFEIF is enabled, the code scheduler attempts to move any operation except a store or divide over a branch instruction. A branch instruction may have been inserted to protect the operation. UNSAFEIF allows code movement for a block of code. SAFEIF prevents code movement over an IF statement for a block of code. UNSAFEIF and SAFEIF apply to one block at a time and the last directive appearing in a block is the directive used. The CDIR$ directives override the default or CFT compiler option for one block of code.

BL/NOBL directives - The BL and NOBL directives enable or disable the prefetch of operands over a loop branch, respectively. The code scheduler usually attempts to prefetch operations in eligible short scalar loops. Subscripts for the iteration after the last one may be out of range and cause an operand range error. BL allows prefetch of code. NOBL prevents prefetch of an operand for the next block of code. The CDIR$ directives override the default or CFT compiler options. The CDIR$ directives apply to one block at a time and the last directive appearing in a block is the directive used.

1.6.2.7 Dynamic common block directive (DYNAMIC)

The DYNAMIC directive declares dynamic common blocks for users with dynamic common block capability. The COS loaders do not support the dynamic common block capability.

Format:

```
 ____________________
|                    |
| DYNAMIC b[,b]...   |
|____________________|
```

b Name of a previously encountered common block

1.6.2.8 Array bounds checking directive (BOUNDS)

The BOUNDS directive checks most array references for out-of-bounds subscripts. The BOUNDS directive is enabled when the ON=O compiler option is specified under COS or the -e o compiler option is specified under UNICOS and can be controlled by a CDIR$ BOUNDS directive. If ON=O or -e o is not specified, all CDIR$ BOUNDS directives are ignored.

The ON=O or -e o option is global to all program units in the compilation. The BOUNDS directives are local to the program unit where they appear.

Bounds checking typically increases program run time by a factor of 10 and inhibits vectorization of any DO loop that references a checked array.

Bounds checking is not applied to arrays of type character or array references that appear in argument lists or in input/output statements. If an array has a last dimension of * or 1, bounds checking is not performed on the last dimension. Dependency messages issued with bounds checking turned on may not appear when bounds checking is turned off, because bounds checking is performed by passing an array argument to a nonvectorizable procedure or function. If a DO loop contains an array being checked, a dependency message may be issued.

BOUNDS options - The BOUNDS directive can be specified with three different argument options.

- The BOUNDS directive with no arguments

  ```
  BOUNDS
  ```

  This option enables bounds checking for all arrays. It remains in effect until another BOUNDS directive or the end of the compilation unit is encountered.

- The BOUNDS directive with an empty argument list

  ```
  BOUNDS( )
  ```

  This option disables bounds checking for all arrays. It remains in effect until another BOUNDS directive or the end of the compilation unit is encountered.

- The BOUNDS directive with an argument list

  BOUNDS(*a*,*b*,*c*)

  This option enables bounds checking only for the arrays named in the argument list and remains in effect only for the current routine, or until another BOUNDS directive is encountered. Bounds checking can be enabled and disabled many times in a specific compilation unit. Bounds checking for all arrays is performed in subsequent program units until another BOUNDS directive is encountered.

1.6.2.9 Optimization directives

The following directives are optimization directives.

- BLOCK
- NO SIDE EFFECTS
- ALIGN
- NOIFCON
- RESUMEIFCON
- RESUMEDOREP
- NODOREP
- CVL
- NOCVL

BLOCK directive - The CFT compiler divides source code into sections called blocks. The BLOCK directive, specified in advance of a FORTRAN statement, causes a block to begin with the succeeding FORTRAN statement.

Blocks are used as the basis for optimization and vectorization by the compiler. This directive is useful for machine-timing tests and for certain unusual program debugging applications.

NO SIDE EFFECTS directive - The NO SIDE EFFECTS directive declares that an external subprogram has no side effects. A NO SIDE EFFECTS external subprogram does not redefine the value of a variable local to the calling program, passed as an argument to the subprogram, or declared in a common block. Using the NO SIDE EFFECTS directive allows CFT to keep information in registers across subprogram invocations without reloading the information from memory after returning from the subprogram. Intrinsic functions are assumed to have no side effects.

Format:

```
 _____________________________
|                             |
| NO SIDE EFFECTS f[,f]...    |
|_____________________________|
```

f Symbolic name of an external subprogram the user guarantees to have no side effects. *f* must not be the name of a dummy procedure.

A NO SIDE EFFECTS subprogram should receive inputs from its arguments. The subprogram should not reference or define variables in a common block shared by a program unit in the calling chain, or redefine the value of its arguments. If these conditions are not met, results can be unpredictable.

The NO SIDE EFFECTS directive must precede arithmetic statement functions or executable statements in a program. If the name of a subprogram appears in a NO SIDE EFFECTS directive and an EXTERNAL declaration, the EXTERNAL declaration must precede the NO SIDE EFFECTS directive.

CFT may move invocations of a NO SIDE EFFECTS subprogram from the body of a DO loop to the loop introduction if the arguments to that function are invariant in the loop. This may affect the results of the program, particularly if the NO SIDE EFFECTS subprogram calls functions like the random number generator or the real-time clock.

ALIGN directive - The ALIGN directive causes the next referenced statement label, the first instruction of the next DO loop body, or the next ENTRY point to align on an instruction buffer boundary. The beginning of a DO loop, a referenced statement label, or an ENTRY point will be aligned. The ALIGN directive must appear immediately before the aligned statement.

If the ALIGN directive does not immediately precede a SUBROUTINE statement, PROGRAM statement, FUNCTION statement, ENTRY statement, DO statement, or a statement with a referenced statement label, a warning message is issued and the directive is ignored.

CFT does not generate a loop construct for short vector loops; therefore, these loops are not aligned. If an ALIGN directive appears before a short vector loop, a warning message is issued and the directive is ignored.

The ALIGN directive is useful for fitting loops and short subprograms into instruction buffers, so the buffer will not need frequent reloading.

An ALIGN directive preceding a DO statement with a referenced label on the statement causes the body of the DO loop, not the preamble, to be aligned.

Format:

```
 ________
|        |
| ALIGN  |
|________|
```

NOIFCON directive - The NOIFCON directive disables optimization of conditional replacement statements of the form IF(*logical exp*)*var=expression,* except when the statement can be converted to a MAX/MIN function. Conditional replacement statements appearing before a NOIFCON directive can be optimized at the level specified on the CFT control statement or command. Optimization is disabled only for conditional replacement statements appearing after the NOIFCON directive. The NOIFCON directive is ignored if the optimization level is NOIFCON.

Format:

```
 __________
|          |
| NOIFCON  |
|__________|
```

RESUMEIFCON directive - The RESUMEIFCON directive enables optimization of conditional replacement statements at the level specified in CFT compiler call. When this optimization is enabled, CFT attempts to optimize statements of the form IF(*logical exp*)*var=expression* by producing code similar to that for *var*=CVMG*x(expression,var,condition*). If NOIFCON is specified on the compiler call, either by default or with an OPT=NOIFCON parameter, or if the optimization has not been disabled by a CDIR$ NOIFCON directive, the RESUMEIFCON directive is ignored.

Format:

```
 ______________
|              |
| RESUMEIFCON  |
|______________|
```

RESUMEDOREP directive - The RESUMEDOREP directive, specified before a DO statement, enables replacement of successive one-line DO loops by calling a $SCILIB routine at the level specified by the OPT parameter on the CFT control statement or the -o parameter on the UNICOS CFT command. If OPT=NODOREP or -o nodorep , CDIR$ RESUMEDOREP is ignored. OPT=SAFEDOREP or -o safedorep is the default option.

Format:

```
 _____________
|             |
| RESUMEDOREP |
|_____________|
```

NODOREP directive - The NODOREP directive disables replacement of one-line DO loops with a call to a $SCILIB routine until a RESUMEDOREP directive is used. Specifying NODOREP has no effect on vectorization of successive DO loops.

Format:

```
 _________
|         |
| NODOREP |
|_________|
```

CVL directive - CVL compiles the next DO loop containing potential unvectorizable dependencies into a vector and a scalar version of the loop. The version which will be used is determined by a run-time test. CVL overrides the OPT=NOCVL compiler option.

NOCVL directive - NOCVL prevents CFT from compiling the next DO loop in the conditional vector and scalar version of the loop. NOVCL overrides the OPT=CVL or -o cvl compiler option and can be used to save space for loops with known dependencies.

1.6.2.10 Debugging directives (DEBUG, NODEBUG)

DEBUG and NODEBUG are debugging directives enabling or disabling the generation of sequence number labels, respectively. The DEBUG and NODEBUG directives are recognized only when the DEBUG parameter is specified on the CFT control statement or command. See the Symbolic Interactive Debugger (SID) User's Guide, CRI publication SG-0056 for a detailed description on the debugging directives.

DEBUG writes sequence number labels for executable FORTRAN statements to the Debug Symbol Table, allowing breakpoints to be set with the Symbolic Interactive Debugger (SID) at statement sequence numbers. DEBUG disables vectorization and scheduling.

NODEBUG disables sequence number label generation and restores vectorization and scheduling.

1.6.2.11 ROLL/UNROLL directives

The ROLL and UNROLL directives control DO-loop unrolling. ROLL and UNROLL have no effect if UNROLL=0 is specified on the CFT control statement.

ROLL specifies that all DO loops remain rolled until an UNROLL directive is encountered.

UNROLL specifies that inner DO loops with constant limits are candidates for DO-loop unrolling.

1.7 EXTERNAL ROUTINES

Enabling the W compiler option under COS or the w compiler option under UNICOS causes all floating-point arithmetic operations to be replaced by return jumps to the external routines shown in table 1-3. These routines shown are called with the call-by-value sequence. Normally the first operand is in S1 or V1, the second is in S2 or V2, and results are returned in S1 or V1. An exception is the divide routines, where the reciprocal approximation to S2 or V2 is returned in S1 or V1.

Table 1-3. External Routines

Operation	External routine
s + s	RASS%
s - s	RSSS%
s * s	RMSS%
1 / s	RDSS%
s + v	RASV%
s - v	RSSV%
s * v	RMSV%
1 / v	RDSV%
v + v	RAVV%
v - v	RSVV%
v * v	RMVV%
v - s	RSVS%

v = vector
s = scalar

NOTE

The following operations are changed as indicated before the external routine is called.

```
   v + s → s + v                  s / v → (1/v) * s
   v * s → s * v                   -s   → 0 - s
   v / s → (1/s) * v               -v   → 0 - v
 v1 / v2 → (1/v2) * v1
```

2. PROGRAM STRUCTURE

This section describes the structure of a FORTRAN program and the statements that define the structure. It begins with an overview of the divisions of a FORTRAN program. The remainder of the chapter covers these same subjects in detail.

2.1 THE EXECUTABLE PROGRAM

An *executable program* consists of a group of one or more program units and procedures. A *procedure* is executable code, not necessarily FORTRAN, that can be invoked from a program unit or another procedure; procedures are discussed below. A *program unit* is an ordered set of FORTRAN statements, which can be a main program or a subprogram:

- The *main program* is the first program unit to receive control and cannot be invoked by another program unit; there must be exactly one main program in an executable program.

- A *subprogram* is a program unit that is not the main program. It can be a *procedure subprogram,* which specifies a procedure, or a *specification* (*block data*) *subprogram,* which initializes variables in blocks of memory and is not executable.

A *procedure* is an executable entity invoked by a *procedure call.* A procedure call can pass values to a procedure, and procedures can return values. A call can appear at different points in either a program unit or another procedure. Code for specifying a procedure can be any of the following: a subprogram (function or subroutine), a statement within a program unit (statement function), code provided by the compiler (intrinsic function), or user-supplied non-FORTRAN code (external function or subroutine). A procedure can be a subroutine or a function:

- A *subroutine* is invoked only by the CALL statement.

- A *function* is invoked by its name in an expression; it must return at least one value, called the *function value.*

A function can be one of the following:

- A *statement function* is specified by a single statement within the program unit that uses the function.

- An *intrinsic function* is included with a given compiler and is always available with that compiler.

- An *external function* is specified by user-supplied code outside the calling program unit.

If a procedure is specified by user-supplied code outside the calling program unit, it is an *external procedure,* which can be either an external function or a subroutine. An external procedure can be specified either by a FORTRAN subprogram or by non-FORTRAN code. A FORTRAN subprogram that specifies a procedure is a procedure subprogram, which can be either a *subroutine subprogram* to specify a subroutine or a *function subprogram* to specify an external function.

Figure 2-1 illustrates the breakdown of FORTRAN terms; all instances of a given term appear below it connected to it with dashed lines. All instances of program units are enclosed in boxes. In the interest of legibility, the five instances of the term *function* and the two instances of the term *subroutine* are not tied to unifying labels.

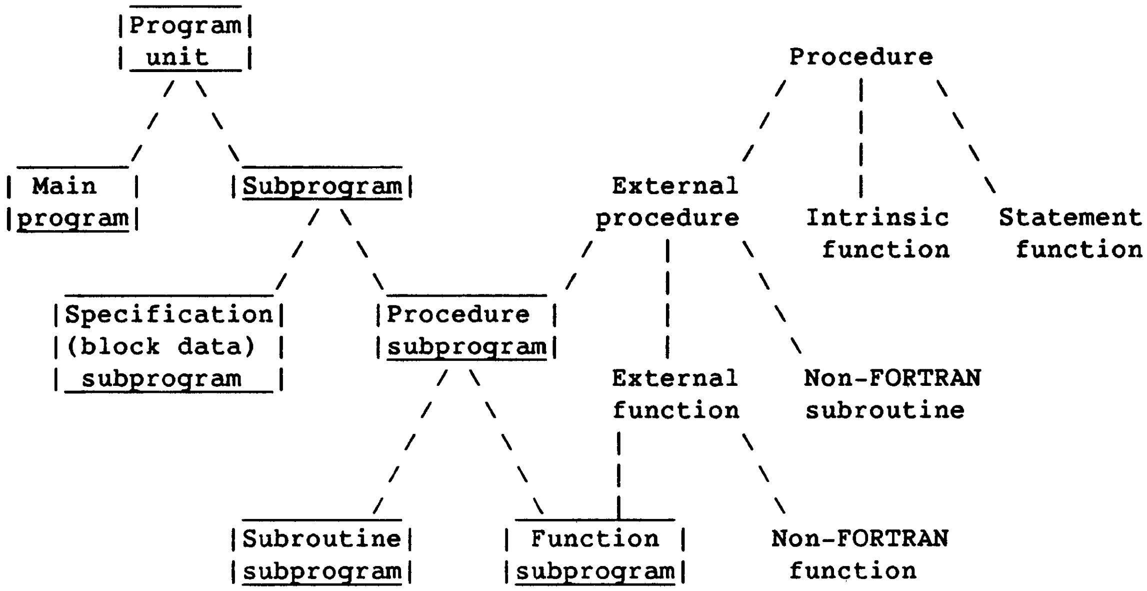


Figure 2-1. Subcategories of FORTRAN Terms (Program Units Boxed)

A *common block* is an area of memory that can be referenced by any program unit or procedure in a program and serves as a means to pass values. A *named* common block has a name specified in a COMMON statement, along with the names of variables, arrays, or constants stored in the block; its size is the same for use by all program units. Common blocks are discussed in section 4.

An *actual argument* is an entity whose value is passed to a procedure; a *dummy argument* is an entity within a procedure which takes the value of an actual argument. Before the actual argument is passed to a procedure, symbolic names are replaced by values, and expressions are evàluated.

2.2 PROGRAM UNITS

A program unit contains a sequence of FORTRAN statements and optional comment lines. An executable program must include one main program and can include one or more subprograms. Each program unit must end with an END statement.

The main program's first statement can be a PROGRAM statement, which is required for several compiler options, and cannot be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

A subprogram begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement. A subprogram must not reference a main program. The main program can reference one or more subprograms during its execution.

When a subprogram entry is referenced, execution begins with the first executable statement following the ENTRY statement named in that reference. If stack mode is specified on the CFT control statement or command, recursive use of procedures is allowed; otherwise CFT runs in static mode and execution of a procedure subprogram must be terminated by a RETURN or END statement before the subprogram can be referenced again.

Neither the main program nor a subprogram can reference a main program. The main program can reference one, more than one, or no subprogram during its execution, provided each is defined before main program execution.

2.2.1 PROGRAM STATEMENT

Although the PROGRAM statement is optional, it is recommended because several compiler options (for example, F) require it. When used, it is the first statement of the main program.

Format:

PROGRAM *pgm* [(*h*)]

pgm Symbolic name of the main program in which the PROGRAM statement appears; from one to eight alphanumeric characters. The name is global.

h Sequence of any CFT characters except ! ; has no effect on the executable program, but is allowed so that programs for other implementations of FORTRAN will run.

The ANSI FORTRAN Standard does not provide for the *h* field in the PROGRAM statement.

Example:

```
PROGRAM A1B2C3D4

PROGRAM X (INPUT,OUTPUT)
```

2.3 FUNCTIONS

A *function* is an executable entity that is invoked by its name and used as an element of an expression. It must return at least one value, called the *function value,* and can return other values.

The value returned by a function has the same data type as the function name or entry name used. Types are determined differently for different kinds of functions, which are discussed under individual headings in this section. Function names obey the same type conventions as variable names.

A function is one of the following:

- A *statement function* is specified by a single statement within the program unit that uses the function.

- An *intrinsic function* is provided by the CFT compiler and is always available unless an external function of the same name is substituted.

- An *external function* is specified by user-supplied code outside the calling program unit; if it is written in FORTRAN, it is a *function subprogram,* discussed later under the heading External Functions and Function Subprograms. (Routines in other languages are not called subprograms in ANSI terminology.)

2.3.1 FUNCTION REFERENCE

All functions use the same form of reference.

Format:

```
 ___________________
|                   |
| fun([a[,a]...])   |
|___________________|
```

fun	Symbolic name of a function or dummy procedure
a	Actual argument; the parentheses are required even with no argument.

The type of each argument and the number of actual arguments specified in a function reference must agree with the (dummy) arguments defined in the specification of the function being referenced.

The length of a character function must be declared by the CHARACTER statement. In addition, the length of a character function subprogram can be declared in the subprogram's FUNCTION statement.

Execution of a function reference results in the following actions:

- Expressions that are actual arguments are evaluated.
- Actual arguments are associated with the corresponding dummy arguments.
- The function is executed.
- Control is returned to the calling program unit. The function value is then available to the expression to be evaluated, which contains the function reference.

2.3.2 STATEMENT FUNCTIONS

A statement function is a function specified by a statement appearing in the same program unit that invokes the function; this specifying statement is called the *statement function definition statement.* It appears after the specification statements, before the first executable statement, and before any other statement that references the function; it is non-executable and is not part of the normal execution sequence.

An actual argument in a statement function reference can be any expression except an array expression. The exception to this is a character expression involving concatenation of an operand whose length is specified by an asterisk in parentheses, unless the operand is the symbolic name of a constant.

A statement function definition cannot reference the function subprogram in which it is defined. For example, a statement function defined within function CAT cannot reference CAT.

2.3.2.1 Statement function definition statement

The statement function definition statement specifies a function for use within the same program unit.

Format:

```
 ________________________
|                        |
| fun ([d[,d]...]) = e   |
|________________________|
```

fun Symbolic name of the statement function, local to the program unit that contains the function. The name must not appear in an EXTERNAL statement and must not be any of the following: a symbolic name in any specification statement other than a type statement (to specify the type of the function); a common block name in the same program unit; an actual argument.

d Dummy argument, local to this statement. The same name cannot appear twice in the list.

e Expression. The relationship between *fun* and *e* must conform to the assignment rules in table 5-1. The type of expression *e* can differ from the type of the statement function name *fun*.

Statement function dummy arguments serve only to indicate the order, number, and type of arguments for a single statement function. A dummy argument name used outside the statement function statement does not refer to the dummy argument, but can be used in the following ways:

- As a dummy argument of the same type in another statement function definition statement

- As a variable of the same type appearing elsewhere in the program unit

- As a common block name

- As a dummy argument in a FUNCTION or SUBROUTINE statement in the same subprogram that contains the statement function statement

Each primary of the expression *e* must be one of the following.

- A constant

- The symbolic name of a constant

- A statement function dummy argument referenced as a variable

- A reference to a variable used elsewhere in the same program unit

- An array element reference

- An intrinsic function reference

- A reference to a statement function defined previously in the same program unit

- An external function reference; this must not cause a dummy argument of the statement function to become undefined or redefined.

- A dummy procedure reference

- An expression enclosed in parentheses

Examples:

```
DISCRIM(X,Y,Z)=Y**2-4.*X*Z

ROOT(A,B,C,SIGN)=(-B+SIGN*SQRT(4*A*C))/(2.*A)

CIRCUM(R)=6.2831852*R

VOL( )=4.1887901*R**3    (where R appears elsewhere in the same
                          program unit)
```

In the following example, function F is evaluated for X=2, even though X has a value of 1 outside the definition statement; Y therefore equals 3 rather than 2; the value of X outside the function statement is unaffected by the use of the function.

```
INTEGER X
F(X)=X+1
X=1
Y=F(2)
```

The type of a statement function is determined in the same way as a variable's type; that is, it can be implicit in the name or can be declared in a type statement preceding the statement function definition statement.

Example:

```
    LOGICAL EVEN
    EVEN(N)=MOD(N,2).EQ.0
```

2.3.3 INTRINSIC FUNCTIONS

An *intrinsic function* is a prespecified function for performing common operations and is always available unless you replace it with an external function of the same name. Some intrinsic functions are called from libraries included with the CFT compiler, while others cause CFT to generate in-line code. Intrinsic functions contain optimized code and frequently run faster than user-supplied code, but certain applications might require functions using different units, number systems, etc. To replace an intrinsic function with a user-supplied function with the same name, use the EXTERNAL statement to declare the name of the external function; this statement is described in the later subsection, Arguments.

CFT also provides a set of non-ANSI utility procedures, referenced like external functions, which extend the use of operating system features and input/output operations. They are described in appendix C.

2.3.3.1 Referencing intrinsic functions

The actual arguments for an intrinsic function must agree in type, number, and order with those shown in appendix B. An actual argument can be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant.

Generic function names are used for families of intrinsic functions that perform similar operations but differ in the data types required for arguments and returned as results. These names simplify referencing because the same function name can be used with more than one type of argument. A generic name, however, cannot be used for an intrinsic function used as an actual argument. Appendix B lists the intrinsic functions in groups, with the first name in each group serving as a generic for the whole group.

2.3.3.2 Restrictions

Intrinsic functions are undefined for some values such as LOG(-1). Out-of-range arguments cause run-time messages to be issued. Appendix B shows allowable argument ranges.

Example:

```
    T = TAN(THETA)
```

The above function reference is undefined if the value of THETA is pi/2 radians (90°).

2.4 SUBPROGRAMS

A *subprogram* is a program unit that is not the main program. A subprogram can be a *procedure subprogram,* which specifies a procedure (function or subroutine) and is invoked by another program unit or a *specification (block data) subprogram,* which initializes variables in blocks of memory. A procedure subprogram can be one of the following:

- A *subroutine subprogram* specifies a subroutine, which is a procedure called by a CALL statement.
- A *function subprogram* specifies a function. The use of functions is described under the previous heading, Functions; function subprograms are described within this subsection.

Some subroutines and external functions are not specified in FORTRAN (and are therefore not considered "subprograms" in ANSI terminology.) They can be written in Cray Assembly Language (CAL) or another high-level language and are separately assembled or compiled. The Macros and Opdefs Reference Manual, CRI publication SR-0012, describes the creation of non-FORTRAN procedures. Appendix F describes the creation of external procedures in CAL, Pascal, and C.

A subprogram begins with a FUNCTION, SUBROUTINE, or BLOCK DATA statement. Execution can begin with the next statement or with the statement following an ENTRY statement in a function or subroutine subprogram, described below.

An ENTRY statement allows entering a procedure subprogram at any executable statement not within a DO loop or block IF range. A procedure subprogram can contain one or more ENTRY statements following its FUNCTION or SUBROUTINE statement.

A subprogram ends with an END statement. A RETURN statement allows returning control to the calling program unit before the end of the subprogram. Execution of a RETURN or END statement terminates a reference to a procedure subprogram and returns control to the calling program unit.

The EXTERNAL statement declares a name to be the name of an external function, subroutine, or dummy procedure; it must be used in the following cases:

- When an external or dummy procedure name is to be passed as an actual argument, even if the use of the name is not ambiguous. See the later subsection, Dummy Procedures.
- When you want to replace an intrinsic function with a function subprogram of the same name, such as to use a different algorithm.

Execution of a RETURN or END statement within a subprogram causes all entities within the subprogram to become undefined, except for the following.

- Entities in a common block
- Initially defined entities
- Entities specified by SAVE statements

The ANSI FORTRAN Standard specifies that, on execution of a RETURN or END statement in a subprogram, entities within the subprogram that are in a named common block become undefined unless the common block name appears in a program unit that is referencing the subprogram.

2.4.1 RECURSION IN SUBPROGRAMS

CFT function and subroutine subprograms may normally not reference themselves, either directly or indirectly. However, in stack mode, a CFT extension, function and subroutine subprograms may call themselves. (Refer to section 1 for a description of the compiler parameter associated with stack mode.)

In stack mode, a subroutine can call itself through a CALL statement, or its entry point address can be passed to another function or subroutine as an argument. For example, the following statements are allowed within subroutine subprogram XYZ:

```
CALL XYZ

CALL ABC(XYZ)

X = DEF(XYZ)
```

A function can call itself by referencing its name. For example, the following statement is allowed within function subprogram FUN:

```
    F = FUN(X)
```

A function name can be used to pass the entry point address of the function as an argument. The function whose name is passed in this manner must be declared in an EXTERNAL or INTRINSIC statement, as in the following example.

```
    EXTERNAL FUN
        ...
    CALL ABC(FUN)
```

Without the prior declaration statement, CFT would not pass the entry point address of FUN to ABC but the current value of function FUN.

The ANSI FORTRAN Standard does not allow a subprogram to call itself.

2.4.2 EXTERNAL FUNCTIONS AND FUNCTION SUBPROGRAMS

An *external function* is a function specified by user-supplied code that is external to the calling program unit; the code may or may not be a FORTRAN subprogram. A *function subprogram* is a FORTRAN subprogram that defines a function, and is one kind of external function.

Functions are called as described in the previous subsection, Functions; function arguments are discussed later under Arguments. Non-FORTRAN functions are discussed in the introduction to this subsection, Subprograms.

2.4.2.1 External functions

An external function name is global to the executable program and must not be the same as any other global name. It cannot be the same as any local name in a program unit where it is referenced or in the program unit that defines it, except that it can be used as a variable in the defining program unit. A function name is followed by an argument list in parentheses, except in a type statement, in an EXTERNAL statement, or when used as an actual argument.

An external function can define one or more of its dummy arguments to return values in addition to the function value. However, this redefinition must not affect any entities referenced in the statement that references the function.

2.4.2.2 Function subprograms

A function subprogram begins with a FUNCTION statement and can contain any statement other than a BLOCK DATA, PROGRAM, SUBROUTINE, or second FUNCTION statement. Execution can begin with the statement following the FUNCTION statement or with the statement following an ENTRY statement. A subprogram ends with a RETURN statement or an END statement.

The symbolic name of a function subprogram or an associated entry name of the same type must appear as a variable name in the function subprogram and must become defined during execution of the procedure. The ***function value*** is the value of this variable when a RETURN or END statement is executed in the subprogram. The symbolic name of a function specified by a FUNCTION or ENTRY statement must not appear in any other non-executable statement except a type statement and must appear only as a variable in executable statements.

The function value's type is implicit to the function name but can be overridden by the specification of another type. If the type is specified in the FUNCTION statement, the function name must not also appear in a type statement; redundant type specifications are not allowed. If the function name is a character variable with a length specification of (*), it must not appear as an operand for concatenation except in a character assignment statement.

In stack mode (specified as an option on the CFT control statement or command), the function value is handled separately for each recursive repetition of the function. See the introduction to this subsection, Subprograms, concerning a subprogram referencing itself.

In a function subprogram, the symbolic name of a dummy argument is local and must not appear in an EQUIVALENCE, PARAMETER, SAVE, or DATA statement; as a pointer; or in a COMMON statement except as a common block name.

2.4.2.3 FUNCTION statement

A function subprogram begins with a FUNCTION statement. It identifies a subprogram as a function subprogram and establishes the function's symbolic name and data type (if the type is not implied by the name).

[*type*] FUNCTION *fun*([*d*[,*d*]...])

type Declares the function's data type, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER[**len*]. The type is declared if it is not implicit in the name, specified within the subprogram, or specified in a type statement preceding the function call.

len Length of the result of a character function; can be an unsigned, nonzero, positive integer constant or positive integer constant expression enclosed in parentheses (expression cannot include the symbolic name of a constant), or an asterisk enclosed in parentheses, (*).

fun Symbolic name of the function subprogram; global name of 1-8 alphanumeric characters

d Dummy argument representing a variable, array, or dummy procedure name. Parentheses are required even with no arguments listed.

2.4.3 SUBROUTINES AND SUBROUTINE SUBPROGRAMS

A ***subroutine*** is a procedure called by a CALL statement within a program unit or procedure; a subroutine is not necessarily a specified by a FORTRAN subprogram. A ***subroutine subprogram*** is a FORTRAN subprogram that specifies a subroutine.

Values can be passed to a subroutine as arguments in the CALL statement or by the use of entities in common blocks. Dummy arguments and entities specified in a COMMON statement in the subroutine can become defined and returned to the calling program unit. Arguments are discussed later in this section; common blocks are described in section 4.

See the introduction to this subsection, Subprograms, concerning a subprogram referencing itself.

2.4.3.1 Requirements

A subroutine subprogram begins with a SUBROUTINE statement containing the subroutine's name and any arguments or alternate return specifiers, and ends with an END statement, which returns control to the calling program unit. A subroutine subprogram cannot contain a BLOCK DATA, FUNCTION, or PROGRAM statement, or a second SUBROUTINE statement.

The RETURN statement normally returns control to the statement that follows the CALL statement, but can return control to a different statement. The RETURN statement and alternate returns are discussed later in this subsection.

A subroutine subprogram can contain one or more ENTRY statements, described later in this subsection, which provide alternative entry points; the name shown in the ENTRY statement is specified in the subroutine call if that entry point is needed.

The name of a subroutine or subroutine entry is global, and within the calling program unit cannot be used as a local name, function name, or function entry name.

2.4.3.2 SUBROUTINE statement

The SUBROUTINE statement identifies a subprogram as a subroutine subprogram. It contains the subroutine name and an optional list of dummy arguments or asterisks corresponding to alternate return specifiers.

Format:

SUBROUTINE *sub* [([*d*[,*d*]...])]

sub Symbolic name of the subroutine; the name is global.

d Dummy argument or asterisk. A dummy argument represents an entity used in the subprogram, corresponding to an actual argument in the CALL statement that calls the subprogram. An asterisk corresponds to an alternate return specifier in the CALL statement; see the RETURN statement in the next subsection.

Examples:

```
SUBROUTINE SAM

SUBROUTINE GEORGE(A,B)

SUBROUTINE TOM(*,X,*,Y)
```

The first example is the first statement of subroutine SAM, which has no dummy arguments or alternate returns. The second example is for a subroutine with dummy arguments A and B. The third is for a subroutine with dummy arguments X and Y, and two asterisks corresponding to alternate return specifiers in the CALL statement. The examples shown for the CALL statement (following) correspond to these examples.

2.4.3.3 CALL statement (subroutine reference)

The CALL statement causes execution of the subroutine specified in the statement; it can include actual arguments to be passed to the subroutine and can specify other statements to which control can be returned.

Format:

```
 ____________________________
|                            |
| CALL sub [([a[,a]...])]    |
|____________________________|
```

sub Symbolic name of a subroutine, subroutine entry (see ENTRY statement later in this subsection), or dummy procedure. A dummy procedure is used within a subprogram; the subroutine to be called is specified as an actual argument in the call to the subprogram that contains this CALL statement.

a Actual argument or alternate return specifier of the form **s*, where *s* is a statement label in the calling program unit. A RETURN statement, described later in this subsection, in the called subroutine can return control to a statement so specified.

Examples:

```
CALL SAM

CALL GEORGE(X,-1)

CALL TOM(*10,X,*20,Y)
```

Corresponding to the examples shown for the SUBROUTINE statement (preceding paragraph), the first example above calls subroutine SAM; the second calls subroutine GEORGE with actual arguments X and -1; the third calls subroutine TOM with actual arguments X and Y and alternate return specifiers for statements 10 and 20.

<u>Execution of a CALL statement</u> - Execution of a CALL statement results in the following:

- Evaluation of actual arguments that are expressions
- Association of actual arguments with the corresponding dummy arguments
- The actions specified by the referenced subroutine

Control can be returned to the first executable statement following the CALL statement or to a statement indicated by an alternate return specifier in the CALL statement. Return of control to the referencing program unit completes execution of the CALL statement.

2.4.4 STATEMENTS FOR USING SUBPROGRAMS

The following statements determine the transfer of control to and from function subprograms and subroutine subprograms.

2.4.4.1 ENTRY statement

The ENTRY statement is used in a procedure subprogram to allow execution to begin at any executable statement not within a DO loop or IF-block range; execution begins with the first executable statement following the ENTRY statement. A procedure subprogram can contain one or more ENTRY statements following its FUNCTION or SUBROUTINE statement.

Format:

ENTRY *en*[([*d*[,*d*]...])]

en Function or subroutine name that is an entry in the procedure subprogram. This name cannot be a dummy argument in the same subprogram.

d Dummy argument representing a variable name, array name, procedure name, or an asterisk associated with an alternate return specifier. An alternate return asterisk can be used only in a subroutine subprogram.

An entry name in a function or subroutine subprogram is referenced in the same way as the subprogram name. The order, number, and types of dummy arguments in an ENTRY statement must agree with the actual arguments in any reference to that ENTRY statement. These names need not agree with those specified in a FUNCTION, SUBROUTINE, or other ENTRY statement in the same subprogram. Type agreement is not required for an actual argument specifying a subroutine name, which has no type.

The function name *en* specified in an ENTRY statement is considered an external function name and is associated with all variables that are associated with the name appearing in the FUNCTION statement. When one of these variables becomes defined, all associated variables and function names of the same type also become defined; those not of the same type become undefined. Names appearing in the ENTRY statement can differ in type from the name in the FUNCTION statement.

Entry statement restrictions - A function or subroutine name specified in an ENTRY statement cannot be the same as any name specified in PROGRAM, BLOCK DATA, FUNCTION, SUBROUTINE, or ENTRY statements in the same executable program.

The function name specified in an ENTRY statement must not appear as a variable in any statement preceding that ENTRY statement except a type statement.

A name appearing as a dummy argument in an ENTRY statement cannot appear in an executable statement preceding that ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement preceding the executable statement.

If a dummy argument name is referenced in an executable statement, it must also be specified in a FUNCTION, SUBROUTINE, or ENTRY statement referenced before execution of the executable statement. A dummy argument specified in an ENTRY statement cannot be referenced in a statement function definition statement unless it also appears as a dummy argument in the same definition statement or in a preceding FUNCTION, SUBROUTINE, or ENTRY statement.

If a function subprogram name is of type character, each entry name in the function subprogram must be of type character; the function name and all entry names must have the same declared length, whether it is an integer or (*) to denote an adjustable length.

2.4.4.2 RETURN statement

A RETURN statement returns control from a procedure subprogram to the referencing program unit. The statement is used only in function subprograms and subroutine subprograms. A subprogram can contain more than one RETURN statement, but a subprogram need not contain a RETURN statement since execution of an END statement in a function or subroutine subprogram has the same effect as executing a RETURN statement.

In a function subprogram, the RETURN statement has no arguments. In a subroutine subprogram, the RETURN statement uses the following format.

Format:

```
 ______________
|              |
| RETURN [i]   |
|______________|
```

i Integer expression specifying an alternate return. *i* indicates the *i*th return specifier in the list of dummy arguments in a subroutine subprogram; see the next paragraph.

<u>Alternate return</u> - The alternate return option allows a subroutine to return control to a statement identified in the CALL statement that called the subroutine. *i* in the RETURN statement references the *i*th asterisk in the dummy argument list of a SUBROUTINE statement; this asterisk in turn specifies a statement label preceded by the *i*th asterisk in the CALL statement. If *i* is less than one or greater than the number of asterisks specified, RETURN *i* is treated as RETURN.

Example:

```
      ...
      CALL SUB (*5,A,B,*6)
      ...
   5  statement
      ...
   6  statement
      ...
      SUBROUTINE SUB (*,A,B,*)
      ...
   10 RETURN 1
   11 RETURN 2
      ...
```

Execution of statement 10 above returns control to statement 5 because RETURN 1 refers to the first asterisk in the SUBROUTINE statement and therefore the first asterisk in the CALL statement. Execution of statement 11 returns control to statement 6.

2.4.5 SAVE STATEMENT

In a subprogram, a SAVE statement retains the definition status of an entity after the execution of a RETURN or END statement. The entity remains defined in the current program unit only. The SAVE statement must appear before any executable statement in a program unit. All entities specified in a SAVE statement are assigned to static storage.

The ANSI FORTRAN Standard does not specify storage allocation methods.

Format:

```
 ___________________
|                   |
| SAVE [a[,a]...]   |
|___________________|
```

a Named common block name preceded and followed by a slash; a variable name; or an array name.

The names of dummy arguments or procedures must not be specified in a SAVE statement. Variables and arrays within a common block must not be specified except by specifying the entire block. A common block specified in a SAVE statement must also be specified in every subprogram where the common block appears.

If *a* is omitted, all common blocks, variables, and arrays are assumed specified.

A name must not appear more than once in the SAVE statements of a program unit.

A SAVE statement can appear in the main program; this use of the SAVE statement can be used for programs running in STACK mode when it is necessary to assign values to static storage. For example, values used in a multitasked program must be allocated to static storage prior to the execution of a TASK START statement; this can be accomplished by using a SAVE statement in the main program prior to the TASK START statement.

2.4.6 BLOCK DATA SUBPROGRAMS

A *block data* subprogram provides initial values for variables and array elements in named common blocks, which are discussed at the end of this section. A block data subprogram contains no executable statements and is not called by another program unit. A block data subprogram can initialize more than one common block, and one common block can be initialized in more than one block data subprogram.

A block data subprogram begins with a BLOCK DATA statement and ends with an END statement. The BLOCK DATA statement can contain the subprogram's name; unnamed block data subprograms are described below. The only other statements that can appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, SAVE, DATA, and type statements.

During one invocation of CFT, up to 26 unnamed block data subprograms can be encountered. CFT assigns the name BLCKDATA to the first unnamed block data subprogram, BLCKDATB to the second, BLCKDATC to the third, etc. Separate compilations can give the same name to two different block data subprograms; this prevents proper loading of the routines, so use care to prevent such duplication. Any number of differently named block data subprograms can be specified in an executable program.

The ANSI FORTRAN Standard does not allow a common block to be initialized in more than one block data subprogram, and allows only one unnamed block data subprogram in an executable program.

2.4.6.1 BLOCK DATA statement

The BLOCK DATA statement identifies a subprogram as a block data subprogram and can contain a subprogram name. A block data subprogram provides initial values for variables and array elements in named common blocks.

Format:

```
BLOCK DATA [sub]
```

sub Symbolic name of the block data subprogram in which the BLOCK DATA statement appears

The optional name *sub* is a global name and must not be the same as the name of an external procedure, main program, or other block data subprogram in the same executable program. The name *sub* must not be the same as any local name in the subprogram.

The ANSI FORTRAN Standard does not allow the name of a block data subprogram to be the same as a common block name.

Example:

```
    BLOCK DATA BD1
    COMMON/NAME1/TABLEA,TABLEB,TEST1,TEST2
    DIMENSION TABLEA(10,10),TABLEB(6,2,2)
    DATA TABLEA/100*123./,TABLEB/12*0.,12*1./
    DATA TEST1/72.35E-20/
    END
```

2.5 ARGUMENTS

An *actual argument* is an entity whose value is passed to a procedure; a *dummy argument* is an entity within a procedure which takes the value of an actual argument. Before the actual argument is passed to a procedure, symbolic names are replaced by values, and expressions are evaluated.

Arguments of type character are discussed in subsection 3.8, Character Type. Arrays as arguments are discussed in subsection 4.3, Arrays.

Data can be communicated in the following ways:

- To a statement function or intrinsic function by an argument list
- To a statement function by local variables
- To and from an external procedure by an argument list or by common blocks, described in section 4

Procedure names can be communicated to an external procedure only by an argument list.

2.5.1 ASSOCIATION OF ARGUMENTS

Execution of a procedure reference causes actual arguments to be associated with dummy arguments: the first actual argument with the first dummy argument, etc. A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument; a subroutine name has no type and must be associated with a dummy procedure. Argument association can be carried through more than one level of procedure reference; a valid association exists at the last level only if a valid association exists at all intermediate levels.

All appearances of a dummy argument within a procedure become associated with the corresponding actual argument when a procedure reference is executed. A dummy argument is undefined if it is not currently associated with an actual argument.

If the actual argument is a variable name, array element name, or substring name, the associated dummy argument can be defined or redefined within the subprogram. A dummy argument must not be redefined within the subprogram if the associated actual argument is a constant, the name of a constant, a function reference, or an expression involving operators or enclosed in parentheses.

If a subprogram reference causes two dummy arguments in the referenced subprogram to become associated with each other, neither dummy argument can become defined in the subprogram. For example, if two dummy arguments are passed the same actual argument, they become associated with each other and cannot legally become defined.

If a subprogram reference associates a dummy argument with an entity in a common block, neither the dummy argument nor the entity in the common block can become defined within the subprogram if the common block is declared in the subprogram or is referenced during execution of the subprogram.

Example:

```
    In calling program unit:            In called subprogram:

    COMMON B                            SUBROUTINE XYZ (A)
    CALL XYZ (B)                        COMMON C
```

The above subroutine call associates A with B; B and C are associated in a common block. Neither A nor C can become defined during the execution of subroutine XYZ or by any procedures it references.

2.5.2 ACTUAL ARGUMENTS FOR EXTERNAL PROCEDURES

Actual arguments specify the entities that are to be associated with the dummy arguments of a referenced subroutine or function.

The actual arguments in a reference to an external procedure must agree in order, number, and type with the dummy arguments in the procedure or procedure entry. An actual argument can be a subroutine name or an alternate return specifier (see subsection 2.4.4.2, RETURN Statement); these are not associated with data types and are exceptions to the requirement for type agreement.

An actual argument in an external procedure reference must be one of the following. (Actual arguments for intrinsic functions and statement functions are discussed in subsection 2.3, Functions.)

- An expression. The expression is evaluated, and the resulting value is passed to the procedure; that is, it is evaluated just before the association of arguments takes place. Expressions include the following, among other possibilities:

  - A variable or array element name. The subscript is evaluated just before the arguments are associated. The subscript value remains constant as long as the arguments are associated, even if the subscript contains variables that are redefined during the association.

  - A function reference with its own argument list (empty where appropriate). The function is evaluated, and the resulting function value is passed to the procedure that is being referenced.

  - A character substring name. A character expression involving concatenation of an operand with a length specification of (*) cannot be used unless the operand is the name of a constant. See subsection 3.8.3, Arguments of Type Character.

- The name of an intrinsic function, external function, or subroutine. This allows a procedure reference to specify a second procedure to be called by the referenced procedure. See the later subsection, Dummy Procedures.

- An array name. The referenced actual array must be at least as large as the dummy array in the called procedure; see the subsection Arrays in section 4.

- *For subroutines only*: An alternate return specifier of the form **n*, where *n* is a statement label in the calling program unit (see the previous subsection, Alternate Return).

- A dummy argument within the subprogram containing the procedure reference. Example:

  ```
  ...                 SUBROUTINE SUBA(Y)     SUBROUTINE SUBB(Z)
  CALL SUBA(X)        CALL SUBB(Y)           ...
  ...                 ...
  ```

  Variable Y above serves as a dummy argument within subroutine SUBA, but is used as an actual argument in the call to subroutine SUBB. This allows a value to be passed through several levels of procedures. (Alternate return specifiers cannot be used this way.)

2.5.3 DUMMY ARGUMENTS

External procedures use dummy arguments to indicate the types of actual arguments and whether each is a single value, an array, or a procedure. A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined. A dummy argument name is local to a subprogram. Dummy arguments for statement functions are limited to scalar values.

Each dummy argument is classified as a variable, array, or procedure. A dummy argument name can appear wherever an actual name of the same class and type can appear, except where explicitly prohibited. A dummy argument that is a variable can be associated with an actual argument that is a variable, array element, substring, or expression. Dummy argument names of type integer can appear as adjustable dimension declarators in dummy array declarators.

A dummy argument name cannot appear in an EQUIVALENCE, DATA, SAVE, INTRINSIC, or PARAMETER statement, as a pointee in a POINTER statement, or in a COMMON statement, except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, or statement function statement in the same program unit.

2.5.4 DUMMY PROCEDURES

A *dummy procedure* is a dummy argument used as a procedure name in a call to an external procedure. This allows a procedure call to specify a second procedure to be referenced by the called procedure. In the following example, dummy argument S is associated with subroutine SUBB, which causes the CALL statement to call subroutine SUBB with argument A.

```
...                              FUNCTION HIGHEST(S,Z)
TOP=HIGHEST(SUBB,A)              ...
...                              CALL S(Z)
                                 ...
                                 END
```

Before being passed as actual arguments, procedure names are declared in EXTERNAL or INTRINSIC statements, to distinguish them from other kinds of arguments. Statement functions cannot be used as dummy procedures. The use of the EXTERNAL statement allows replacing an intrinsic function with another function that is better suited to a given application; when this is done, the intrinsic function of the same name is unavailable.

A dummy procedure name must be associated with an actual argument that is a procedure; the procedure must be available at the time a reference to it is executed. If a dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine and must not appear in a type statement or be referenced as a function. If a dummy argument appears in a type statement and an EXTERNAL statement, the actual argument must be the name of a function.

When an intrinsic function name is passed to a procedure to be used in a function reference within the procedure, the arguments in the function reference must agree in number and type with those specified for the intrinsic function. As with dummy arguments, the type of a dummy procedure must match the type of its actual argument when it is used as a function. (See the intrinsic functions in appendix B.)

A dummy procedure name must appear in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement. It cannot be the name of an array or character variable and is immediately followed by a left parenthesis except: as an argument; in a type, EXTERNAL, or CALL statement; or as a common block name in a COMMON or SAVE statement. A dummy function subprogram name must be declared in an EXTERNAL or INTRINSIC statement.

2.5.4.1 EXTERNAL statement

An EXTERNAL statement identifies a symbolic name as representing an external procedure and permits its use as an actual argument.

Format:

```
 ____________________________
|                            |
| EXTERNAL proc [,proc]...   |
|____________________________|
```

proc Name of an external procedure, dummy procedure, or block data subprogram

The appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is to be an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. A statement function name must not appear in an EXTERNAL statement.

If an intrinsic function or utility procedure name appears in an EXTERNAL statement, that name becomes the name of some external procedure. The intrinsic function or utility procedure of the same name is not available for reference in that program unit.

A given symbolic name can appear only once in all of the EXTERNAL statements of a program unit.

2.5.4.2 INTRINSIC statement

An INTRINSIC statement identifies a symbolic name as an intrinsic function. It permits use of a specific intrinsic function name as an actual argument.

Format:

```
 _________________________
|                         |
| INTRINSIC fun[,fun]...  |
|_________________________|
```

fun Intrinsic function name

The appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name. If an intrinsic function name is an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit.

The following intrinsic function names must not appear as actual arguments.

AMAX0
AMAX1
AMIN0
AMIN1
CHAR
CMPLX
DBLE
DMAX1
DMIN1
FLOAT
ICHAR
IDINT
IFIX
INT
LGE
LGT
LLE
LLT
MAX
MAX0
MAX1
MIN
MIN0
MIN1
REAL
SNGL

The appearance of a generic function name in an INTRINSIC statement does not cause loss of the name's generic property.

A given symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement. In addition, it can appear only once in all of the INTRINSIC statements of a program unit. Appendix B lists the intrinsic functions.

3. DATA TYPES

Data can be specified or input in a FORTRAN program as a constant, a variable, an array, an array element, or a function reference. A *constant* is an invariant value, which cannot be modified. A *variable* is a name that can assume different values during program execution. An *array* is an ordered set of data items of the same type identified by a single name. An *array element* is one item in an array and, like a variable, can assume different values during program execution; it is identified by the array name and one or more numbers to specify its position within the array. Functions are discussed in section 2; the other terms above are discussed in section 4.

Each data item has a *data type*, which specifies how the item is represented, stored, and manipulated. Data types can be any of the following.

- Integer - integral, signed values
- Real - values approximating real numbers, consisting of a mantissa and an exponent
- Double-precision - the same as real data, but extended to about twice the precision
- Complex - values approximating complex numbers as pairs of signed, real data items. The first item in the pair represents the real portion and the second, the imaginary portion of the data.
- Logical - the logical values true and false
- Character - sequences of characters
- Pointer - values representing storage addresses; only a variable can have a pointer value.
- Boolean - octal values representing the binary contents of Cray words; only constants, intrinsic functions, or expressions can be Boolean.

An *arithmetic* value is a number that can be used in an arithmetic operation, and can be of type integer, real, double-precision, complex. Table 3-1 shows some examples of values represented in these data types. Pointer variables, Boolean constants, and Hollerith constants can also be used in limited ways in arithmetic expressions.

The ANSI FORTRAN Standard does not provide for the Boolean or pointer data types.

After a symbolic name is identified with a type, that type applies to all uses of that name. Exception: a common block can have the same name as a variable or array, but the common block name has no type.

A data type (for a symbolic constant, variable, array, external function or statement function) can be specified explicitly in a type statement or implicitly by the first letter of its symbolic name. If no type is specified, a first letter of I, J, K, L, M, or N implies type integer; any other first letter implies type real. The default for implied typing can be changed or confirmed by an IMPLICIT statement. Type statements and the IMPLICIT statement are discussed in subsections 3.1.1 and 3.1.2.

The data type of an array element is the same as the data type of the array that contains it.

The data type of a function establishes the type of data returned when the function is referenced in an expression. The data type of a user-specified function can be implied by its name, specified in a type statement, or specified within a function subprogram. The names of intrinsic functions can be specific or generic, so that the result type agrees with the type of the data passed to the function. See section 2 and appendix B.

3.1 DATA SPECIFICATION STATEMENTS

Data specification statements specify characteristics and values of data to be used by a program. Data specification statements are not executable; therefore, statement labels associated with them cannot be referenced to control the execution sequence. Data specification statements usually appear (and most must appear) before any executable statements in a program.

The kinds of data specification statements are as follows.

- Declaration and initialization: PARAMETER, DIMENSION, and DATA
- Association: EQUIVALENCE and COMMON
- Type statements, including IMPLICIT and IMPLICIT NONE

The first and second categories are discussed in section 4.

3.1.1 TYPE STATEMENTS

A type statement declares the type of an entity and either overrides or confirms implicit typing and can specify array dimensions by including array declarators with array names.

The appearance of the symbolic name of a constant, variable, array, or function in a type statement specifies the data type for all appearances of that name in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

Subroutine names, main program names, and block data subprogram names must not appear in a type statement.

If a specific intrinsic function name appears in a type statement that conflicts with that function's type as specified in appendix B, the function name must first be declared in an EXTERNAL statement. Otherwise, the conflicting type statement is ignored and a warning message is issued.

The form of type statements other than CHARACTER is as follows. (See subsection 3.8, Character Type for additional information.)

```
 _________________
|                 |
| type v[,v]...   |
|_________________|
```

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

v Symbolic name of a constant, variable name, array name, function name, dummy procedure name, or array declarator. An array declarator includes dimension declarators as shown in subsection 4.3, Arrays.

Examples:

```
INTEGER NPAK(60,230),RTEST,XREF(20,2),ARRAY

DOUBLE PRECISION ANG(1014,8),KLIM,PTEST(10)

COMPLEX IMAG,COMARR(30,3),ZREF,KITEMS(64)

LOGICAL KEY2,BOOLSET(64,64),TTABLEB(2,20,15)
```

In the above examples, numbers in parentheses are dimension declarators, and the names preceding the parentheses are array names. See appendix E for extensions of the type declaration statements.

3.1.2 IMPLICIT STATEMENT

An IMPLICIT statement changes or confirms the data typing of constants, variables, arrays, and functions according to the first letter of their symbolic names.

IMPLICIT *type* (a_1[-a_n],a_1[-a_n]]...)[,*type*(a_1[-a_n][,a_1[-a_n]]...)]...

type Data type: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER[**len*], or LOGICAL

a Single letter or a range of single letters denoted by the first and last letter of the range separated by a hyphen. A range of letters within the alphabet can be written in the form a_{first}-a_{last}.

len Length of the character entities. *len* can be an unsigned, nonzero, positive integer constant or expression with a value is less than 2^{24}-1.

An IMPLICIT statement specifies a type for names of constants, variables, arrays, and functions (except intrinsic functions), beginning with any letter appearing singly or within a range in the specification. IMPLICIT statements do not change the types of intrinsic functions. An IMPLICIT statement applies only to the program unit containing it.

The appearance of a constant, variable, array, or function name in a type statement overrides or confirms type specification by an IMPLICIT statement. An explicit type specification in a FUNCTION statement overrides IMPLICIT statement typing for the name of that function subprogram.

Within the specification statements of a program unit, IMPLICIT statements must precede all specification statements other than PARAMETER statements. A PARAMETER statement must follow an IMPLICIT statement to affect the typing of constants named in the PARAMETER statement.

A letter can be specified (or implied within a range of letters) only once in all of the IMPLICIT statements in a program unit.

Examples:

```
    IMPLICIT LOGICAL(L)

    IMPLICIT DOUBLE PRECISION(X,Y),COMPLEX(C)

    IMPLICIT INTEGER(A,B,F-K),REAL(M-W,Z)
```

The last example declares variables starting with A, B, F, G, H, I, J, and K to be integer type.

3.1.3 IMPLICIT NONE STATEMENT (CFT EXTENSION)

The IMPLICIT NONE statement prevents the use of implicit typing by requiring all constant, variable, array, dummy argument, statement function, and function (except intrinsic function) names to appear in an explicit type statement. It also requires all nonintrinsic subroutine and function names to appear in an EXTERNAL statement. The statement consists of the words IMPLICIT NONE with no other parameters.

The IMPLICIT NONE statement applies only to the program unit containing it and must be the first specification statement.

Failure to provide type or EXTERNAL declarations is a fatal error when IMPLICIT NONE is specified, except in the following cases.

- Intrinsic subroutine and function names need not appear in explicit type statements and must not be declared EXTERNAL.

- Pointers appearing in a POINTER statement are always assumed to be of type INTEGER and need not be explicitly typed.

3.2 INTEGER TYPE

Integer data represents positive, negative, or zero values with no fractional part. The form of an *integer constant* is an optional sign followed by a nonempty sequence of digits specifying a decimal integer value. An integer value occupies one storage unit in a storage sequence, as discussed later in this section under the heading Storage and Association.

Integer values are represented in the Cray Computer System by integer binary values (I) in the following range:

$-2^{46} \leq I < 2^{46}$ or $-10^{14} \leq I < 10^{14}$

When the INTEGER=64 option is specified on the CFT control statement (see part 1), integer values can be represented in the following range:

$-2^{63} \leq I < 2^{63}$ or $-10^{19} \leq I < 10^{19}$

The ANSI FORTRAN Standard does not specify a range of values for integer values.

3.3 REAL TYPE

A *real* value approximates the value of a real number, including a mantissa and exponent. A real value occupies one storage unit in a storage sequence described in subsection 4.5, Storage and Association.

A real constant is written as one of the following.

- Basic real constant
- Basic real constant followed by a real exponent
- Integer constant followed by a real exponent

A *basic real constant* consists of an optional sign, an integer portion, a decimal point, and a fractional portion, in that order. The integer and fractional portions are sequences of digits representing integer and fractional decimal constants. Either, but not both, of these portions can be omitted. A basic real constant can be written with more digits than can be used to approximate its constant; the excess digits are lost by CFT in roundoff.

A *real exponent* is a power of 10 specified by an optionally signed integer constant following the letter E. The value preceding the exponent is multiplied by 10 to the specified power.

Nonzero real values are represented in the Cray computer by normalized floating-point binary values (R) in the following range.

$$2^{-8193} \leq R < 2^{8189}$$

Nonzero real values have a maximum of 48 significant binary digits of precision. Rounding and truncation during computation can cause fewer than 48 reliable bits to be generated. This approximates the following decimal range with approximately 14 decimal digits of precision.

$$10^{-2466} < R < 10^{2465}$$

The ANSI FORTRAN Standard does not specify a range of values for real values.

3.4 DOUBLE-PRECISION TYPE

Double-precision values are signed approximations of real numbers, extended to approximately twice the precision of real values. Double-precision values can be positive, negative, or zero, and occupy two consecutive units in a storage sequence, as described in subsection 4.5, heading Storage and Association.

A double-precision constant is written as one of the following:

- Basic real constant followed by a double-precision exponent
- Integer constant followed by a double-precision exponent

Basic real and integer constants are defined in previous subsections.

A double-precision exponent has the same range as a real exponent. The form of a double-precision exponent is the letter D followed by an optionally signed integer value; the value preceding the exponent is multiplied by 10 to the specified integer value. This notation is shown in table 3-1.

Nonzero double-precision values are represented in the Cray computers by normalized floating-point binary values (D) in the following range.

$$2^{-8193} \leq D \leq 2^{8189}$$

Nonzero double-precision values have a maximum of 96 significant binary digits of precision. Rounding and truncation during computation can cause fewer than 96 reliable bits to be generated. This approximates the following decimal range, with approximately 29 decimal digits of precision.

$$10^{-2466} < D < 10^{2465}$$

The ANSI FORTRAN Standard does not specify a range of values for double-precision values.

3.5 COMPLEX TYPE

A *complex* value approximates a complex number as a pair of signed real values. The first item in the pair represents the real portion and the second, the imaginary portion of the value. A complex constant is written as a pair of integer or real constants, within parentheses and separated by a comma. A complex value occupies two consecutive storage units in a storage sequence: the first for the real portion and the second for the imaginary portion. See subsection 4.5, Storage and Association.

As discussed in subsection 4.5, Storage and Association, when a complex entity becomes defined, all partially associated real entities become defined. When both parts of a complex entity become defined as a result of partially associated real or complex entities becoming defined, the complex entity becomes defined.

The real and imaginary components of nonzero complex values are represented in the Cray computer by two real values (C_{real},C_{imag}) in the following range.

$$2^{-8193} \leq C_{real} \text{ or } C_{imag} \leq 2^{8189}$$

Each component contains a maximum of 48 significant binary digits of precision, approximating to the following decimal range with approximately 14 decimal digits of accuracy.

$$10^{-2466} < C_{real} \text{ or } C_{imag} < 10^{2465}$$

The ANSI FORTRAN Standard does not specify a range of values for complex value components.

3.6 BOOLEAN TYPE

A *Boolean* constant consists of zeros and ones representing the contents of bits in a single storage unit (64-bit Cray computer word). There are no Boolean variables, arrays, or array elements. No user-specified function can return a Boolean result, but some intrinsic functions (Cray extensions) can return Boolean results. Masking expressions and some arithmetic expressions involving Boolean operands have Boolean results. Arithmetic operations involving Boolean values are performed with integer arithmetic, and no type conversion is performed.

A Boolean constant is written in one of two forms, octal or hexadecimal:

- The octal form contains 1 to 22 octal digits (0 through 7) followed by the letter B. 22 octal digits in a Boolean value correspond to the binary contents of a complete storage unit (64-bit word). In this case, the leftmost octal digit can be only 0 or 1, representing the content of the leftmost bit position (bit 0). Each successive octal digit specifies the contents of the next three bit positions until the last octal digit specifies the contents of the rightmost three bit positions (bits 61, 62, and 63). A Boolean value represented by fewer than 22 octal digits is right-justified; that is, it represents the rightmost bits of a Cray word: bits *x* through 63.

- The hexadecimal form contains the letter X followed by a string of 1 to 16 hexadecimal digits (0-9, A-F) enclosed by apostrophes or quotation marks. The hexadecimal digits may be preceded by an optional + or - sign. Blanks are ignored in hexadecimal values. When a Boolean value contains 16 hexadecimal digits, their binary equivalents correspond to the content of every bit position in the storage unit (64-bit word). A Boolean value containing fewer than 16 hexadecimal digits is right-justified, as in the octal representation.

- A Hollerith constant is of type Boolean.

Examples:

Boolean Constant	Internal Representation (Octal)
Octal notation:	
1274653312572676113745B	1274653312572676113745
0B	0000000000000000000000
1777777777777777777777B	1777777777777777777777
77740B	0000000000000000077740
00776B	0000000000000000000776
Hexadecimal notation:	
X'ABE'	0000000000000000005276
X"2F0"	0000000000000000001360
X"-340"	1777777777777777776300
X'1 2 3'	0000000000000000000443
X'FFFFFFFFFFFFFFFF'	1777777777777777777777

The ANSI FORTRAN Standard does not provide for Boolean values.

3.7 LOGICAL TYPE

A *logical* value is only true or false, and occupies one storage unit in a storage sequence (see section 4). The forms and internal representations of logical values are shown below.

Form	Value	Internal Representation
.TRUE. or .T.	True	Any negative value
.FALSE. or .F.	False	Zero or any positive value

The ANSI FORTRAN Standard does not provide for the .T. or .F. form of the logical value.

3.8 CHARACTER TYPE

A *character* value or *character string* consists of one or more 8-bit ASCII characters (listed in appendix A). A character constant is written as a sequence of characters preceded and followed by a delimiter, which can be an apostrophe or quotation mark. The delimiter can appear as a character within the string if it appears twice in succession; the double character is interpreted as a single character. Blanks in a character string are significant.

Examples:

```
    Constant 'ABC' or "ABC" is stored internally as ABC.
    Constant '''' or "'" is stored internally as '.
    Constant '""' is stored internally as "".
```

The length of a character constant is the number of characters between its delimiters, with each pair of consecutive delimiters counted as a single character. A character variable is declared with its length by the CHARACTER type statement, before any executable statements. The length of each variable is specified by an integer following an asterisk; this specifier can follow any individual variable name or can follow the word CHARACTER for all names lacking length specifiers. The following example shows a declaration for variables X and Z of length 5, W of length 2, and Y of length 7.

```
    CHARACTER*5 W*2,X,Y*7,Z
```

User-specified functions of type character must be declared in the same manner as character variables. Actual and dummy arguments of type character must agree in length.

The length of a character constant must be greater than 0 and less than 1317; this limitation derives from the number of lines allowed in a FORTRAN statement. The length of a character value must be greater than 0 and less than 16393 or 2^{14}-1.

Each character within a string has a position that is numbered ordinally from the first character. These positions are used in specifying character substrings, discussed in subsection 3.8.2, Character Substrings.

When all characters of a character entity become defined, the character entity becomes defined. See subsection 4.5.3, Definition.

The ANSI FORTRAN Standard does not provide for the use of quotation marks as delimiters, and does not specify a maximum length for a character value.

3.8.1 CHARACTER TYPE STATEMENT

The CHARACTER statement declares a symbolic name to be of type character and shows the length of each character entity declared.

```
CHARACTER [*len[,]]nam[*len][,nam[*len]]...
```

len Length specification (number of characters) for an entity

nam Variable name, symbolic name of a constant, function name, dummy procedure name, or array name or array declarator

The length specification following the word CHARACTER refers to each entity without a length specification. If the CHARACTER type statement does not include a length specification, the length is assumed to be one.

The length specification, *len*, can be an unsigned, nonzero integer constant or a positive, nonzero integer constant expression enclosed in parentheses. The value of *len* cannot be greater than 2^{14}- 1.

The ANSI FORTRAN Standard does not specify a maximum character length.

If the entity is an external function, a dummy argument, or a character constant, *len* can be specified as an asterisk enclosed in parentheses: for example, CHARACTER*(*). If the entity is an external function, the function name must appear in a FUNCTION or an ENTRY statement in the same subprogram, and the length is obtained from the referencing program unit. If the entity is a dummy argument, the dummy argument assumes the length of the associated actual argument. If the entity is a character constant with a symbolic name, the constant will assume the length of its corresponding constant expression defined later in a PARAMETER statement.

3.8.2 CHARACTER SUBSTRINGS

A character *substring* consists of one or more contiguous characters within a character variable or character array element (see later in this section under the heading Arrays). A substring name takes the following form:

```
 ______________________________
|                              |
| cvname ([begpos]:[endpos])   |
|______________________________|
```

cvname Name of a character variable or character array element.

([*begpos*]:[*endpos*])
Substring designator. *begpos* and *endpos* are integer expressions designating the beginning and ending character positions of the substring. The minimum and default value of *begpos* is 1; the maximum and default value of *endpos* is the last position.

Examples:

Substring:	Designates Characters:
STRINGA(6:9)	6 through 9 of variable STRINGA
STRINGB(4:)	4 through last of variable STRINGB
STRINGC(2,6)(1:3)	1 though 3 of array element STRINGC(2,6)
STRINGD(5,4)(:7)	1 through 7 of array element STRINGD(5,4)

3.8.3 ARGUMENTS OF TYPE CHARACTER

Arguments of type character can be character variables, substrings, or, in external procedures, character arrays. An actual argument associated with a character dummy argument must be of type character, with a length exceeding or equaling that of the dummy argument.

If a function subprogram name is of type character, each entry name in the function subprogram must be of type character; the function name and all entry names must have the same declared length, whether it is an integer or (*), denoting adjustable length. A character dummy argument whose length is specified as (*) must not appear as an operand for concatenation, except in a character assignment statement.

If the length *len* of a dummy argument of type character is less than the length of an associated actual argument, the leftmost *len* characters of the actual argument are associated with the dummy argument; that is, the rightmost characters are truncated. *len* must not exceed the actual argument's length (this can cause unpredictable execution-time errors).

If a dummy argument of type character for an external procedure is an array name, the restriction on the array's length (described in the previous paragraph) is for the entire array and not for each array element. The length of a dummy array element can differ from the length of an associated actual array, element, or element substring; but the dummy array must not extend beyond the end of the associated actual array.

When an actual argument is a character substring, the argument's length is the substring's length. If an actual argument is the concatenation of two or more operands, the argument's length is the sum of the operands' lengths. A character expression involving concatenation of an operand with a length specification of (*) cannot be used as an actual or dummy argument unless the operand is the name of a constant in a character assignment statement.

Substring expressions are evaluated immediately preceding argument association; the expression values remain constant as long as argument association continues.

3.9 POINTERS

A *pointer* is a type INTEGER variable whose value is used as the address of another entity, which is called a *pointee*. A pointer is declared by the POINTER statement, which also specifies its pointee.

Example:

```
    POINTER (P,B),(Q,C)
```

The above statement declares pointer P and its pointee B, and pointer Q and pointee C; the pointer's current value is used as the pointee's address whenever the pointee is referenced.

The pointee does not have an address until the pointer's value is defined: the pointee's value starts at the location specified by the pointer. A pointer cannot be assigned a literal constant number; its value must derive from the location of an entity or be defined in terms of other pointers:

- The LOC function returns the address of a variable. Example:

  ```
  P=LOC(X)
  ```

- A pointer can be defined in terms of another pointer. Example:

  ```
  Q=P+100
  ```

The CFT compiler assumes that a pointee's storage is never overlaid on another variable's storage; that is, a pointee should never be associated with another variable or array. The programmer is responsible for preventing such association. In the following example, B and C have the same pointer, so the result is likely to be invalid.

```
POINTER (P,B),(P,C)
REAL X,B,C
P=LOC(X)
B=1.0
C=2.0
PRINT *,B
```

3.9.1 POINTER STATEMENT (CFT EXTENSION)

The POINTER statement declares one variable to be a pointer (which is of type pointer), and another variable to be its pointee; that is, the pointer's value is the address of the pointee.

POINTER (*p*,*a*)[,(*p*,*a*)]...

p Pointer to the corresponding *a*. *p* contains the word address of the location of *a*.

a Variable pointed to by *p*. The content of *p* is used for any reference to *a*; therefore *a* is not assigned storage. *a* cannot be a dummy argument nor appear in a COMMON, EQUIVALENCE, or DATA statement. *a* cannot be of type CHARACTER.

a$_i$ can be dimensioned in a separate type or DIMENSION statement or dimensioned in the pointer list itself, as in the following example:

```
    POINTER (IX,X(N,0:M))
```

In a subroutine or function, the *a* dimension expression can contain references to variables in a common block or to dummy arguments.

The pointer, *p*, is a simple variable of type INTEGER and can appear in a common list or be a dummy argument in a subprogram. *p* can be set with a LOC function reference or as an absolute address, as in the following example:

```
    COMMON POOL (100000) INTEGER JCB (128), WORD64
    REAL A (1),B(1),C(1) POINTER
    (PJCB,JCB),(IA,A),(IB,B),(IC,C),(ADDRESS,WORD64)
    DATA ADDRESS/64/     PJCB = 0       IA = LOC(POOL)
    IB = IA + 1000       IC = IB + N
```

In effect, WORD64 refers to the contents of absolute address 64; JCB is an array occupying the first 128 words of memory; A is an array of length 1000 located in blank common; B follows A and is of length N; C follows B. A, B, and C are equated to POOL and possibly to each other, depending on the subscript usage. Similarly, WORD64 is the same as JCB(65). However, CFT makes no checks for possible equivalence overlap. Each *a* is assumed to be a distinct entity.

Any change in the value of a pointer causes subsequent references to the corresponding pointee to refer to the new location.

Besides providing a limited form of dynamic storage allocation, the POINTER statement can manipulate linked lists, as in the following example:

```
       SUBROUTINE FINDSAM (SAMSSPOT)
       POINTER (SAMSSPOT, RECORD (N))
       COMMON N
       INTEGER RECORD
  10   IF (RECORD (4) .EQ.'SAM') RETURN
       SAMSSPOT = RECORD (25)
       IF (SAMSSPOT .NE.0) GO TO 10
       PRINT 20
  20   FORMAT ("SAM'S NOT HERE")
       STOP
       END
```

3.9.2 RESTRICTIONS

A pointer can be only a scalar variable; that is, in addition to the other restrictions on variables, it cannot be a constant, array, array element, or function.

A pointer's value occupies one storage unit (discussed later in this section). Its range of values depends on the size of memory for the machine in use.

A pointee cannot appear in an EQUIVALENCE, FUNCTION, SUBROUTINE, ENTRY, SAVE, or COMMON statement; this does not apply to pointers.

The ANSI FORTRAN Standard does not provide for the pointer data type.

Table 3-1. Values Represented in Different Data Types

Value	Int.	Real	Double-prec.	Complex
0	0	0.	0.0D0	(0.,0.)
692	692	692.		(692.,0.)
		692.0		(692.0,0.)
		692E0	692D0	(692E0,0.)
		692.E0	692.D0	(692.E0,0.)
		692.0E0	692.0D0	(692.0E0,0.)
		6920E-1	6920D-1	(6920E-1,0.)
		.692E3	.692D3	(.692E3,0.)
		6.92E2	6.92D2	(6.92E2,0.)
6.128547472	6†	6.128547472		(6.128547472,0.)
		6.128547472E0	6.128547472D0	(6.128547472E0,0.)
		6128547472E-9	6128547472D-9	(6128547472E-9,0.)
		6128547472.0E-9	6128547472.0D-9	(6128547472.0E-9,0.)
		.6128547472E1	.6128547472D1	(.6128547472E1,0.)
		612.8547472E-2	612.8547472D-2	(612.8547472E-2,0.)

† This value, which is different than the original value, is the result of type conversion of the original value to the type shown in this column.

Table 3-1. Values Represented in Different Data Types (continued)

Value	Int.	Real	Double-prec.	Complex
.875i	0†	0.†	0.0D0†	(0.,.875) (0.,875E-3) (0.,.875E0) (0.,8.75E-1) (0.,.000000875E6)
692+.875i	692†	692.†	692.0D0†	(692.,.875) (692E0,0.875) (69.2E1,875E-3) (.692E3,875.E-3) (6.92E2,8.75E-1)

† This value, which is different than the original value, is the result of type conversion of the original value to the type shown in this column.

4. DATA STRUCTURES, STORAGE, AND ASSOCIATION

This section describes the methods of using and storing data: constants, variables, arrays, and common blocks. It also describes how entities are associated. Relevant CFT statements are included.

4.1 CONSTANTS

A *literal* constant is the ASCII representation of a constant value. Except within character constants, blank characters in a literal constant have no effect.

A constant can be given a symbolic name by the PARAMETER statement, in which the name's data type is determined the same way as for variables. That is, the name can be declared in a preceding type statement, or the type can be implicit. For example, in the following statement, R is real even though 1 is an integer:

```
PARAMETER (R=1)
```

A constant name is local to a program unit and cannot be used as the name of anything else except a common block.

A *signed* constant is a literal constant preceded by a sign (+ or -) indicating either a positive or negative number. An *unsigned* constant is a literal constant not preceded by a sign. An *optionally signed* constant can be either signed or unsigned. Arithmetic constants are optionally signed except where otherwise specified. The negative of a nonzero constant, exponent constant, or complex portion is formed by preceding it with a minus sign. If no sign is shown in this position, a positive constant is assumed. The constant zero is neither positive nor negative; a signed zero has the same value as an unsigned zero.

4.1.1 PARAMETER STATEMENT

A PARAMETER statement assigns a symbolic name to a constant.

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

p   Symbolic name of a constant

e   Constant expression. In an exponentiation operation x***y*, *y* must be an integer.

The type of a symbolic name in a PARAMETER statement is specified by its appearance in a previous type statement, by a previous IMPLICIT statement specifying its first letter, or by default. A symbolic name *p* must be followed by an arithmetic expression containing only arithmetic constants or the names of arithmetic constants that have been previously defined in the same or an earlier PARAMETER statement. If the type of a symbolic constant is not its implicit type, it must appear in a type statement before the PARAMETER statement.

The length of a character constant must be specified in a type statement or an IMPLICIT statement before the first appearance of its name. Otherwise, the length is assumed to be one. The length cannot be changed by subsequent statements. If the length is specified as (*), the parameter length is the length of the actual character string.

Type conversion of an arithmetic expression in a PARAMETER statement follows the same rules as assignment statements, discussed in section 6. A symbolic name *p* of type logical can be followed only by a logical constant expression. Similarly, a symbolic name *p* of type character can be followed only by a character constant expression. A symbolic name can be assigned an arithmetic or logical value only once in a program unit. Constants named in a PARAMETER statement can be referenced in any subsequent statement in the same program unit except in a format specification or to form a part of any other constant, such as any part of a complex constant.

Examples:

```
    PARAMETER (PI=3.1415926, C=1.86E5)

    PARAMETER (JOULE=10000000,KELVIN=-273)

    IMPLICIT LOGICAL(A-B)
    PARAMETER (BOOLEAN=.TRUE.,ABOOLEAN=.FALSE.)
```

4.2 VARIABLES

A *variable* is a name whose value can be changed during program execution; it is unsubscripted and is not an array or array element. The term *subscripted variable* is often used to apply to an array element, but should not be assumed to have the characteristics of other variables; see subsection 4.3, Arrays. *Scalar* and *simple* variables are merely variables as defined above, as distinguished from arrays or array elements.

A *static* variable's storage is allocated before program execution and remains in one location during execution; the default storage type for all CFT variables is static. A *stack* variable is dynamically allocated; that is, its storage is allocated at execution time. If stack mode is specified on the CFT control statement or command, all variables are stack variables except those appearing in COMMON, DATA, or SAVE statements, and any variables equivalenced to these variables.

A variable has one data type throughout a program, as specified by the rules governing symbolic names, described under the following heading, Types of Data. The value of a variable can be defined by means of the assignment operator (=) or used in operations according to the rules described in section 5.

A variable name has the following restrictions:

- It cannot appear in a PARAMETER, INTRINSIC, or EXTERNAL statement.
- It cannot be the name of an array, subroutine, main program, or block data subprogram, nor the entry name in an ENTRY statement for an external subroutine, or NAMELIST group name.
- A variable name can be followed by a left parenthesis only if it is preceded by the word FUNCTION in a FUNCTION statement or by the word ENTRY in an ENTRY statement; or if it begins a character substring name (see subsection 3.8, Character Type).

A variable name is local to a program unit; see under the heading later in this section, Scope of Symbolic Names.

A variable can have the same name as a common block. A variable or common block can have the same name as a dummy argument of a statement function, but the dummy argument name is local to the statement function definition statement. The data type is the same in these different uses, except the use as a common block name. If the type is character, the length is also the same. See the discussion of statement functions in section 2.

4.3 ARRAYS

An *array* is a nonempty, ordered sequence of data items, called *array elements*, that occupy consecutive locations in storage. An *array name* is the symbolic name of an array and obeys the same rules for data typing used for other symbolic names; all elements of a given array are of one type. An *array element name* identifies one element and consists of an array name with one or more subscripts indicating the element's position within the array. Each subscript corresponds to an *array dimension*. An *array declarator* is a list item that specifies an array's symbolic name and the size of each dimension of the array.

An array name with no subscript identifies the entire array in contexts where such use is permitted (see subsection 4.3.8, Use of Array Names). In an EQUIVALENCE statement, an array name with no subscript identifies the first element of the array.

An *array element substring* is a character substring of a character array element; see subsection 3.8, Character Type.

Arrays are classified as dummy or actual, depending on their use as dummy arguments; and as constant-size, adjustable, assumed-size, and pointee, depending on how the dimension sizes are determined and on their usage.

4.3.1 DUMMY AND ACTUAL ARRAYS

A *dummy* array is an array (constant-size, adjustable, assumed-size, or pointee; terms defined later) that is a dummy argument in a procedure subprogram. Each dummy array must be associated through one or more levels of procedure references with an actual array or an actual array element. (Association of a dummy array with an actual array element causes the element's address to be used as the dummy array's base address.) See subsection 4.3.7, Arrays as Arguments.

An *actual* array is an array that is not a dummy array. An actual array can be a constant-size or pointee array. All arrays in main programs are actual arrays; actual arrays can also be in procedure subprograms.

In a reference to a subprogram containing a dummy array, the actual argument corresponding to the dummy array name must be either an array name, an array element substring, or an array element name. Size requirements for dummy and actual arrays are discussed in subsection 4.3.6, Array Size.

4.3.2 CONSTANT, ADJUSTABLE, ASSUMED-SIZE AND POINTEE ARRAYS

An array is constant-size, adjustable, assumed-size, or pointee, depending on how its size is determined and how the array is used. Each category has a corresponding kind of declarator, described under the next heading.

A *constant-size* array has dimensions that do not vary in size; that is, the dimension bounds in the array declarator are constant expressions.

An *adjustable* array is a dummy array whose size is determined during program execution, as defined by the array declarator. Each reference to a subprogram can specify dimension sizes for an array in the subprogram; the sizes are constant during subprogram execution. Variables defining an adjustable dimension bound can be redefined or become undefined during execution of the subprogram with no effect on the bound or on array size.

An *assumed-size* array is a dummy array whose last dimension is of an unknown size (specified as "*") that is assumed to be large enough for all references made to the array. The size of an assumed-size array is determined as described under the heading Array Size. An assumed-size array cannot be used as an item in an input/output list or in a READ, WRITE, or PRINT statement containing a format specification.

A *pointee* array is an actual array that is the object of a pointer. Its dimensions are determined in the same way as those of an adjustable array.

The ANSI FORTRAN Standard does not provide for pointee arrays.

4.3.3 DIMENSION STATEMENT

The DIMENSION statement specifies the symbolic names and dimension specifications of arrays. Because arrays can be dimensioned in type statements, the DIMENSION statement is unnecessary.

DIMENSION *a*(*d*)[,*a*(*d*)]...

a(*d*)   Array declarator; see subsection 4.3.4.

Each symbolic name *a* appearing in a DIMENSION statement declares *a* to be an array in that program unit. An array name can appear only once as an array declarator in a program unit. Array declarators can also appear in COMMON statements, type statements, and in POINTER statements.

4.3.4 ARRAY DECLARATORS

An *array declarator* is an item (within a statement) that specifies an array's symbolic name and the size of each dimension in the array. This information is used to allocate storage for an array. Array declarators can be listed in DIMENSION, COMMON, or type declaration statements, or as pointees in a POINTER statement. Within one program unit, only one array declarator is permitted for a given array name. Figure 4-1 shows three array declarators and the arrays that they define.

The format of an array declarator is as follows:

a([l_1:]u_1[,[l_2:]u_2]...)

a Symbolic name of the array

[l_n:]u_n Dimension declarator, one for each dimension in the array. The dimension bounds, *u* and *l*, can be expressions.

l Lower bound of the dimension; default is 1.

u Upper bound of the dimension. This can be specified as "*" for an assumed-size array.

A *dimension declarator* specifies the number of array elements in one dimension of an array; this number is $(u-l)+1$. *u* and *l* can be positive, zero, or negative, so long as $u \geq l$. If the lower bound is omitted, its value is assumed to be 1. An array declarator has as many dimension declarators as the array has dimensions; there can be from one to seven.

A *dimension bound expression* is an expression that serves as the upper or lower bound of a dimension in an array declarator. The expression must be scalar and of integer type. Functions, array elements, and variables are used only in adjustable array declarators, described in subsection 4.3.4.2, Adjustable Array Declarators.

The ANSI FORTRAN Standard does not permit dimension bound expressions to contain function references, array elements, or noninteger variables.

Examples:

```
    DIMENSION ARRAY (34,0:24,1:34), VECTOR (64), Z7144X (5:10,-2:20)

    DIMENSION MATRIX (ROWS,COLUMNS), Y(2*N+1)

    DIMENSION TABLE (3,IVAL, MATRIX,2,2), TAB(6:IVALX,MAT:10)
```

In the last two examples, the use of variables defines adjustable dimensions and is permitted only in procedure subprograms.

4.3.4.1 Kinds of array declarators

Reflecting the different kinds of arrays, array declarators are classified as either actual or dummy; and as constant-size, adjustable, assumed-size, or pointee.

Actual, dummy - An *actual* array declarator declares an actual array. Actual array declarators are permitted in DIMENSION, COMMON, or type statements, with the exception that pointee declarators appear only in POINTER statements.

A *dummy* array declarator declares a dummy array, which can be either constant, adjustable, or assumed-size. Dummy array declarators appear only in procedure subprograms, within DIMENSION or type statements but not COMMON statements.

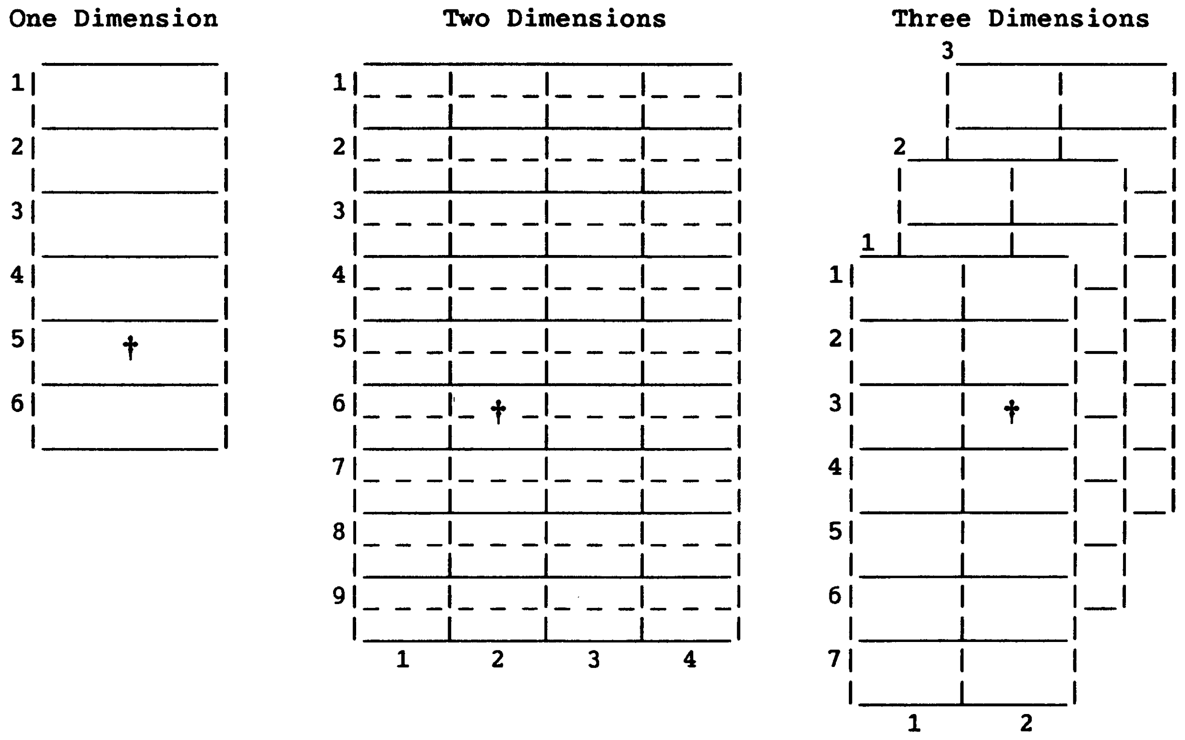
Constant, adjustable, assumed-size, pointee - In a *constant* array declarator, the dimension bound expressions contain no variables, functions, or array elements. *Adjustable* and *pointee* array declarators contain one or more variables, functions, or array elements. In an *assumed-size* array declarator, the upper bound of the last dimension declarator is an asterisk, which indicates an unknown value.

4.3.4.2 Adjustable array declarators

An adjustable array declarator defines an adjustable array; its dimension bound expressions can contain variables, array elements, or functions. A variable or array element in an adjustable dimension declarator must be named in the dummy argument list containing the array name or in a COMMON statement in the same subprogram. Array elements used in an adjustable array declarator must not be elements of the array being declared.

4.3.5 ARRAY ELEMENTS AND SUBSCRIPTS

The *subscript* of an array element name identifies the element's position in the array, and consists of a group of subscript expressions separated by commas and between parentheses. Within a subscript, each subscript expression specifies the element's position in one array dimension. The *subscript value* is the ordinal number of an element's position in the array's storage sequence, as discussed in this subsection.



Array declarator:	ARX(6)	ARY(9,4)	IARZ(7,2,3)
Data type:	Real	Double precision	Integer
Dimension sizes:	6 elements	9 and 4 elements	7,2, and 3 elements
Total elements:	6	36	42
Array element† reference:	ARX(5)	ARY(6,2)	IARZ(3,2,1)
Number of words:	6	72	42

Figure 4-1. Array Specification and Size

An array name designating an entire array causes sequential processing of all array elements; see subsection 4.3.8, Use of Array Names. The format of an array element name is as follows.

```
 _____________
|             |
| a(s[,s]...) |
|_____________|
```

a Array name

s Subscript expression; must be scalar and integer type.

NOTE

In ANSI usage, the term *subscript* refers to a group of subscript expressions that completely specify an element's position within an array. In common usage the term often refers to just one subscript expression within a group. In this manual an attempt is made to avoid ambiguity in the use of these terms.

4.3.5.1 Array subscripts and storage sequence

A subscript expression must yield an integer value when evaluated and can contain references to any arithmetic type of constant, variable, function, or array element. The evaluation of the subscript expression must not alter the value of other expressions within the same statement.

CFT uses 24-bit A registers for subscript calculations. Overflow on intermediate values greater than 2^{23}-1 is not detected; very large values in subscript expressions can produce unpredictable results.

Although an array is arranged in dimensions for programming purposes, it is stored in a single ordinal sequence. Each array element's position in the sequence is specified by its group of subscript expressions, each expression specifying the element's position within one array dimension. Figure 4-2 shows the storage sequence of a typical three-dimensional array. Table 4-1 illustrates the conversion of subscript values to ordinal positions within an array's storage sequence.

The number of subscripts normally equals the number of dimension declarators in the array declarator. When fewer subscript expressions are used, the lower bound is assumed for the missing subscript expressions, and a warning message is issued.

The ANSI FORTRAN Standard does not provide for fewer subscript expressions than declarators.

Array Declarator: X(5,3,2)

Array Arrangement

2

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

1

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

Array Characteristics

Array declarator: X(5,3,2)

Dimensions: 3

Dimension sizes: 5, 3, and 2 elements

Array size: 30 elements

Array Storage Sequence

Ordinal Position	
1	X(1,1,1)
2	X(2,1,1)
3	X(3,1,1)
4	X(4,1,1)
5	X(5,1,1)
6	X(1,2,1)
7	X(2,2,1)
8	X(3,2,1)
9	X(3,2,1)
10	X(5,2,1)
11	X(1,3,1)
12	X(2,3,1)
13	X(3,3,1)
14	X(4,3,1)
15	X(5,3,1)
16	X(1,1,2)
17	X(2,1,2)
18	X(3,1,2)
19	X(4,1,2)
20	X(5,1,2)
21	X(1,2,2)
22	X(2,2,2)
23	X(3,2,2)
24	X(4,2,2)
25	X(5,2,2)
26	X(1,3,2)
27	X(2,3,2)
28	X(3,3,2)
29	X(4,3,2)
30	X(5,3,2)

Figure 4-2. Storage Sequence for a Three-dimensional Array

Table 4-1. Subscript Evaluation

n	Dimension Declarator	Subscript	Subscript Value
1	$(l_1{:}u_1)$	(s_1)	$1+(s_1-l_1)$
2	$(l_1{:}u_1,l_2{:}u_2)$	(s_1,s_2)	$1+(s_1-l_1)$ $+(s_2-l_2)*d_1$
3	$(l_1{:}u_1,l_2{:}u_2,l_3{:}u_3)$	(s_1,s_2,s_3)	$1+(s_1-l_1)$ $+(s_2-l_2)*d_1$ $+(s_3-l_3)*d_2*d_1$
n	$(l_1{:}u_1,\ldots l_n{:}u_n)$	$(s_1,\ldots s_n)$	$1+(s_1-l_1)$ $+(s_2-l_2)*d_1$ $+(s_3-l_3)*d_2*d_1$ $+\ldots$ $+(s_n-l_n)*d_{n-1}$ $*d_{n-2}*\ldots*d_1$

n = Number of dimensions ($1 \le n \le 7$)
l = Lower bound of dimension declarator
u = Upper bound of dimension declarator
s_i = Subscript expression ($l_i \le s_i \le u_i$)
d_i = $(u_i-l_i)+1$

4.3.6 ARRAY SIZE

The size of an array (that is, the number of elements in the array) equals the product of the sizes of all dimensions for that array. The number of storage units (words) in an array is the product of the number of the elements in the array and the number of storage units required for each element. Figure 4-1 shows the sizes of typical arrays.

4.3.6.1 Size of an assumed-size array

The size of an assumed-size dummy array is determined as follows:

- If the actual argument corresponding to the dummy array is a noncharacter array name, the size of the dummy array is the size of the actual argument array.

- If the actual argument corresponding to the dummy array name is a noncharacter array element name with a subscript value of *r* in an array of size *x*, the size of the dummy array is *x*+1-*r*.

- If the actual argument is a character array name, character array element name, or character array element substring name and begins at character storage unit *t* of an array with *c* character storage units, then the size of the dummy array is (*c*+1-*t*)/*l* where *l* is the length of an element of the dummy array.

If an assumed-size dummy array has *n* dimensions, the product of the sizes of the first *n*-1 dimensions must be less than or equal to the size of the array, as determined by one of the above rules.

4.3.6.2 Maximum array size

CFT allows a maximum array size of 4,194,304 Cray computer words when not using extended memory addressing (EMA). The maximum array size when using EMA is 16,777,216 Cray computer words (see the EMA compiler option in section 1). The maximum array sizes are reduced by the computer system being used, the amount of executable code and data associated with the program, the external routines needed by the program, and the memory required for system-related activities.

The ANSI FORTRAN Standard does not specify a maximum for array size.

4.3.7 ARRAYS AS ARGUMENTS

A dummy array can be associated with an actual argument that is either an array name, an array element name, or an array element substring.

The number and size of dimensions in an actual array declarator can differ from those in an associated dummy array declarator, but the dummy array cannot be larger than the actual array. If an actual argument is an array element name with a subscript value of *s* in an array of size *n*, the size of the dummy array must not exceed *n*-*s*+1.

Actual and dummy array elements are associated via their subscript values. When an actual argument is an array element name with a subscript value of *p*, the dummy argument array element with a subscript value of *q* becomes associated with the actual argument array element that has a subscript value of *p*+*q*-1.

Dummy argument names of type integer can appear as adjustable dimension declarators in dummy array declarators. If an actual argument is associated with a dummy argument appearing in an adjustable dimension declarator, the actual argument must be defined with an integer value at the time the procedure is referenced.

An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any variable appearing in the adjustable array declarator is not currently associated with an actual argument or is not in a common block.

4.3.8 USE OF ARRAY NAMES

The use of an array name with no subscript to designate a whole array implies the following:

- The number of values to be processed equals the number of elements in the array.

- The elements of the array are to be taken in sequential order, except in the evaluation of array expressions and in array assignment statements.

In a program unit, each appearance of an array name must be as part of an array element name except when used in the following.

- Array expression

- Array declarator

- List of arguments

- COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statement

- Type statement

- POINTER statement, as a pointee

- Input/output statement: as a format identifier, in a list, or as a unit identifier for an internal file. These do not apply to an assumed-size array.

4.4 DATA STATEMENT

A DATA statement provides initial values for variables, arrays, and array elements. Only those entities named in DATA statements become defined prior to executable program execution; all other entities are undefined at this time.

A DATA statement can be intermixed with specification statements but must follow type statements for variables appearing in the DATA statement.

Entities appearing in DATA statements are assigned to static storage.

The ANSI FORTRAN standard specifies that the DATA statement follows all specification statements.

The ANSI FORTRAN Standard does not specify storage allocation methods.

Format:

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

nlist List of variable names, array names, array element names, substring names, and implied-DO lists (described below) separated by commas

clist List of the form [*r**]*c*[,[*r**]*c*]...

c Constant or the symbolic name of a constant

r Nonzero, unsigned, integer constant or the symbolic name of such a constant

The *r*c* form is interpreted to provide *r* successive values of the constant *c*.

Elements of arrays named in *nlist* must be individually accounted for in *clist*.

Example:

```
    INTEGER I(5)    DATA I/5*0/
```

Array I above has five elements, and the DATA statement assigns it five zeros.

Counting as separate items the elements of any arrays in *nlist*, and counting an *r*c* entry in *clist* as *r* items, the *i*th item in *nlist* becomes defined with the *i*th value from *clist*. The same number of items must be specified by each *nlist* and its corresponding *clist*. The initial values of the entities are defined by this correspondence. Exceptions to this rule are shown in appendix E, Outmoded Features.

Each subscript expression in the list *nlist* must be an integer constant expression except for implied-DO variables. Each substring expression in the list *nlist* must be an integer constant expression.

Names of constants, dummy arguments, and functions must not appear in *nlist*. Names of entities in a named common block or in blank common (including entities associated with an entity in blank common) can appear in *nlist*.

The ANSI FORTRAN Standard does not permit entities in blank common to be initialized in a DATA statement.

If a variable, an array element, or an entity associated with either is defined by a DATA statement more than once in an executable program, the one nearest the end of the program is the only definition to apply.

4.4.1 IMPLIED-DO LIST IN A DATA STATEMENT

An implied-DO list allows a DATA statement to initialize a group of values systematically as in a DO loop. The implied-DO is used frequently for initializing an array. The following format represents one item in an *nlist* as shown above; the values assigned are contained in a subsequent *clist*.

$(\mathit{dlist},i=e_1,e_2[,e_3])$

dlist List of array element names and implied-DO lists separated by commas. *dlist* cannot contain substring names, even if they are substrings of array elements.

i Name of an integer variable called the *implied-DO variable*

e_1, e_2, and e_3

Integer constant expressions containing only integer constants, the names of integer constants, and implied-DO variables of other implied-DO lists containing this implied-DO list within their ranges. If omitted, e_3 is assumed to be 1.

The range of an implied-DO list is the list *dlist*. The iteration count and values of the implied-DO variable *i* are established as for a DO-loop except that the iteration count must be greater than zero. Interpretation of an implied-DO list in a DATA statement causes each item in the list *dlist* to be specified once for each iteration, and for appropriate values to be substituted where implied-DO variables are referenced.

Each subscript expression in the list *dlist* must be an integer constant expression except that the expression may contain implied-DO variables of implied-DO lists that have the subscript expressions within their ranges.

Any declaratives affecting the variable or array names in *nlist* must precede the DATA statement.

Examples:

```
(1)  DIMENSION A(25)
     DATA (A(I),I=1,10)/10*1/
```

The first ten values of array A above are set to 1.

```
(2)  DIMENSION GRID (2,3),KBUF(10,200,2)
     PARAMETER (XCON=6.0)
     DATA GRID /11.0,21.0,12.0,22.0,13.0,23.0/,KBUF/4000*XCON/
     DATA I/1/K/2000/
     PARAMETER (NEG=-6)
     INTEGER NB (10)
     DATA NB/-3,7*-4,2*NEG/
```

4.4.2 DATA TYPES IN A DATA STATEMENT

When the *nlist* entity is of type integer, real, or double-precision, the corresponding *clist* constant is converted, if necessary, to the type of the *nlist* entity according to the rules for arithmetic conversion, as shown in table 2-4.

An *nlist* entity of type logical must correspond to a *clist* constant of the same type. A *clist* entity of type logical must correspond to an *nlist* entity of type logical.

An entity in the list *nlist* that corresponds to a character constant must be of type character. If the length of the character entity in the list *nlist* is greater than the length of its corresponding character constant, the additional rightmost characters in the entity are initially defined with blank characters. If the length of the character entity in the list *nlist* is less than the length of its corresponding character constant, the additional rightmost characters in the constant are ignored.

Other rules concerning data types in the DATA statement are shown in appendix E, Outmoded Features.

4.4.3 ENTITIES THAT CAN BE DEFINED

Any variable or array element can be initially defined in a DATA statement except for the following.

- An entity that is a dummy argument
- A variable in a function subprogram whose name is also the name of the function subprogram
- A pointee
- An entity declared to be in BLANK COMMON

4.5 STORAGE AND ASSOCIATION

This subsection discusses the storage of variables, arrays, and common blocks, and how entities become associated.

4.5.1 STORAGE UNITS AND SEQUENCES

A *numeric storage unit* is a Cray word of 64 bits; a *character storage unit* is an 8-bit byte. A *storage sequence* is a contiguous group of storage units with a consecutive series of addresses. Each array and each common block is stored in a storage sequence. The size of a storage sequence is the number of storage units it contains. Two numeric storage sequences are *associated* if they share at least one storage unit.

An integer, real, or logical value occupies one numeric storage unit. A character value is represented as an 8-bit ASCII code, packed eight characters per word; the storage size depends on the value's length specification. A double-precision or complex value uses a storage sequence of two numeric storage units, with the first storage unit containing the most significant bits of a double-precision value or the real part of a complex value, and the second storage unit containing the least significant bits of a double-precision value or the imaginary part of a complex value.

The ANSI FORTRAN Standard does not specify the relationship between storage units and computer words, nor does it specify any relation between numeric and character storage units.

4.5.2 STATIC AND STACK STORAGE

With *static* storage, any local variable that is allocated memory occupies the same address throughout program execution. Code using static storage is nonreentrant and does not adapt to multitasking. Static storage occurs unless stack mode is specified on the CFT control statement or command.

A *stack* is an area of memory where storage for variables is allocated when a subprogram or procedure begins execution and is released when execution completes. When stack mode is in effect for a program, stack storage is used for all local variables except those named in SAVE, DATA, or COMMON statements. The stack expands and contracts as procedures are invoked and return. The amount of memory available for the stack is determined by the STK parameter on the LDR call or the STACK directive on the call to SEGLDR. Stacks are also called *pushdown storage* or *last-in-first-out* (LIFO).

4.5.3 DEFINITION

A *defined* variable or array element has a value. An *undefined* variable or array element does not have a predictable value. Once defined, a variable or array element keeps a value until it becomes undefined or is redefined.

All variables and array elements are initially undefined and can be defined before or during during program execution. An *initially defined* variable or array element is one defined before program execution. Constants are always defined and are never redefined. A function's value is defined only when it is required.

When an entity of a given type becomes defined, all totally associated entities of different types become undefined. When an entity not of character type becomes defined, all partially associated entities become undefined. This does not apply to two entities of types real and complex or to two entities of type complex.

4.5.4 ASSOCIATION OF ENTITIES

Association of entities occurs when a storage location can be identified by different symbolic names or from different program units: that is, two entities are associated if their storage sequences are associated. *Totally associated* entities have the same storage sequence. *Partially associated* entities share part but not all of a storage sequence. All entities using a given storage unit are affected by the unit's value or undefined status; totally associated entities have the same values and definition status.

Within a program unit, an EQUIVALENCE statement associates entities. Across program units, entities are associated through arguments and COMMON statements. Character entities must not be associated with arithmetic or logical entities.

Partial association can exist between a double-precision or complex value and a second value of type integer, real, logical, double-precision, or complex; or between two character entities. Partial association can occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. (ENTRY statements in a function subprogram associate any variables in the subprogram that have the same names as the ENTRY statements.) Partial association must not occur through argument association.

Example:

```
    INTEGER I
    REAL R(4)
    COMPLEX C(2)
    DOUBLE PRECISION D
    EQUIVALENCE (C(2), R(2), I), (R,D)
```

The EQUIVALENCE statement above specifies that the following storage units are the same:

- The third storage unit of C (that is, the first unit of the second element of C)

- The second storage unit of R

- The storage unit of I

- The second storage unit of D

The storage sequences can be illustrated as follows:

```
                        Storage unit
                    | 1  | 2  | 3  | 4  | 5  |

Complex             |  C(1)   |  C(2)   |

Real                     |R(1)|R(2)|R(3)|R(4)|

Integer                       | I  |

Double precision         |   D     |
```

R(2) and I are totally associated. The following are partially associated: R(1) and C(1), R(2) and C(2), R(3) and C(2), I and C(2), R(1) and D, R(2) and D, I and D, C(1) and D, and C(2) and D. Although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

4.5.5 EQUIVALENCE STATEMENT

An EQUIVALENCE statement specifies the sharing of one or more storage units by two or more entities in a single program unit, in order to use storage more efficiently. This causes the association of those entities.

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

nlist List of two or more variable names, array element names, character substring names, or array names, separated by commas. *nlist* cannot include names of subprogram dummy arguments or variable names that are also function names.

An EQUIVALENCE statement specifies that the storage sequence of each entity in a list *nlist* shares the same first storage unit. This associates all entities in the list and can also indirectly associate other entities. If entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence.

Associated entities are assigned to the same kind of storage, static or stack; static storage is used unless the stack mode option is specified on the CFT control statement or command (see section 1).

The ANSI FORTRAN Standard does not specify storage allocation methods.

4.5.5.1 Array names and array element names

The use of an array name in an EQUIVALENCE statement has the same effect as using the name of the first array element. If a variable and an array are associated, the variable does not assume the properties of an array and the array does not assume the properties of a variable. If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must be less than or equal to the number of dimensions in the array declarator for the array. When the number of subscripts is less than the number of dimensions, the lower bounds are used for the unspecified subscripts; this does not conform to the ANSI standard, and a warning is issued.

4.5.5.2 Restrictions on EQUIVALENCE statements

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit. An EQUIVALENCE statement must not specify the same storage unit to occur more than once in a storage sequence. Example:

```
    DIMENSION A(2)
    EQUIVALENCE (A(1),B) , (A(2),B)     !Illegal statement
```

The above sequence is prohibited because it specifies the same storage unit for A(1) and A(2).

An EQUIVALENCE statement must not specify consecutive storage units to be nonconsecutive. For example, the following is prohibited:

```
    REAL A(2)
    DOUBLE PRECISION D(2)
    EQUIVALENCE (A(1),D(1)),(A(2),D(2)) !Illegal statement
```

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit.

Example:

```
    COMMON/A/X
    COMMON/B/Y
    EQUIVALENCE (X,Y) !Illegal statement
```

EQUIVALENCE statement association must not cause extending of a common block storage sequence by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block.

Example:

```
    COMMON /X/A
    REAL B(2)
    EQUIVALENCE (A,B(2))     !Illegal statement
```

The above sequence is not permitted since it would associate an array element B(1) with a storage unit preceding A in common block X.

An entity of type character can be equivalenced only with other entities of type character. Lengths are not required to be the same. Partial overlapping between character entities can occur through equivalance association.

Example:

```
    CHARACTER A*4,B*4,C(2)*3
    EQUIVALENCE (A,C(1)),(B,C(2))
```

The above sequence associates A with C(2) as shown in the following illustration.

```
Character number:    |01|02|03|04|05|06|07|
                     |---- A ----|
                              |---- B ----|
                     |--C(1)--|--C(2)--|
```

4.6 COMMON BLOCKS

A *common block* is an area of memory that can be referenced by any program unit in a program, and serves as a means of passing values among program units and external procedures. A common block can be referenced by any subprogram in which the common block is declared.

A *named* common block has a name specified in a COMMON statement, along with the names of variables or arrays stored in the block; its size is the same for use by all program units. Different subprograms can include COMMON statements specifying the same common block. A *blank* common block is declared in the same way but without a name; its size need not be constant for use by different program units. Only a named common block can be initialized by a DATA statement.

Within an executable program, the common block storage sequences of all common blocks with the same name share the same first storage unit. The same is true of all blank common blocks. This associates entities among different program units.

4.6.1 COMMON STATEMENT

The COMMON statement associates entities in different program units. This allows different program units to share storage units and to define and reference the same data.

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

cb Common block name. The blank (unnamed) common block is specified when a *cb* does not appear between slashes.

nlist List of variable names, array names, and array declarators separated by commas. Names of dummy arguments of a subprogram cannot appear in the list.

The entities occurring in *nlist* following a block name *cb* are declared to be in common block *cb*. If the first *cb* is omitted, its enclosing slashes are optional and all entities in *nlist* are specified to be in blank common.

Any *cb* (or an omitted *cb* for blank common) can occur more than once in one or more COMMON statements in a program unit. The *nlist* following each successive appearance of the same common block name continues the preceding list for that common block name.

If any entity in a common block is a character variable or character array, all entities in that common block must be of type character. If the common block is defined in any other procedure, the entities in that procedure must be character entities.

EQUIVALENCE statement association must not extend a common block storage sequence by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block.

4.6.2 REFERENCING COMMON BLOCKS

Because a common block associates entities by storage sequence rather than by name, the names and types of variables and arrays can differ between different subprograms. To be referenced, a datum in a common block must be defined, and the reference must be of the type which was declared for that datum in the subprogram where the reference appears.

Qualifications:

- An integer variable that has been assigned an executable statement label must not be referenced in any program unit other than the one in which it was assigned.

- Either part of a complex datum can be referenced as a real datum.
- If any entity in a common block is of type character, all entities in the block must be of type character, and the common block definitions in all subprograms must be of type character.

In a subprogram that has declared a named or blank common block, the entities in the block remain defined after the execution of a RETURN or END statement.

The ANSI FORTRAN Standard specifies that variables in a named common block become undefined if no executing program unit has declared the common block.

4.6.3 COMMON BLOCK NAMES

A symbolic name is the name of a common block if it appears as a block name in a COMMON statement. A common block name is global to the executable program.

A common block name in a program unit may also be the name of any local entity other than a constant, intrinsic function (unless the function is not referenced), or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. A common block name can be the name of a constant, intrinsic function, or external procedure.

The ANSI FORTRAN Standard does not allow a common block name to be the name of a constant, intrinsic function, or external procedure.

4.6.4 COMMON BLOCK STORAGE SEQUENCE

For each common block, a *common block storage sequence* is formed as follows.

- A storage sequence is formed, consisting of the storage sequences of all entities listed in a COMMON statement. The sequence order is determined by the order of these entities.

- This storage sequence is extended to include all storage units of any storage sequence associated with it by an EQUIVALENCE statement. The sequence can be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

4.6.5 COMMON BLOCK SIZE

The size of a common block is the size of its common block storage sequence, including any extensions of the sequence resulting from association by an EQUIVALENCE statement.

The maximum size of common blocks within a program depends on the loader being used: LDR uses the size of the first common block as a maximum; SEGLDR uses the maximum size specified in the whole program as a maximum. Blank common blocks within an executable program are not required to be of the same size and can increase, decrease or remain the same as each program unit is compiled.

The ANSI FORTRAN Standard does not include variable size for named common blocks.

4.6.6 EXTENDED MEMORY COMMON BLOCKS

CFT allows common blocks to contain more than 4 million words of memory by using the extended memory addressing (EMA) feature (see the EMA CPU characteristic in subsections 1.1 and 1.2). When the EMA CPU characteristic is specified, all variables declared in named and blank common blocks are addressed as though they are allocated beyond 4 million words of memory. The variables declared in an extended memory common block can be used like variables declared in a regular common block.

A fatal error message is issued when the NOEMA CPU characteristic is used and any common block is declared to contain more than 4 million words.

4.6.7 TASK COMMON STATEMENT

When multitasking is used, some common blocks may need to be local to a task. CFT allows common blocks to be declared local to a task by using the task common block extension. All variables declared in a task common block are considered local to a task. If multiple tasks execute code containing the same task common block, each task will have a separate copy of the block.

The keyword TASK must precede the keyword COMMON when a named common block is declared. A task common block is allocated at task invocation.

The format of a task common block is as follows.

Format:

```
 ________________________________________________
|                                                |
| TASK COMMON [/[cb]/]nlist[,/[cb]/nlist]...     |
|________________________________________________|
```

cb Task common block name

nlist List of variable names, array names, and array declarators, separated by commas. Names of subprogram dummy arguments cannot appear in the list.

The variables in *nlist* cannot be preset with data. With this exception, the variables can be used like the other variables declared in COMMON.

Stack allocation must be used with task common blocks (see the STACK option in the CFT control statement or command in subsections 1.1 and 1.2). If static allocation is used, all task common blocks are treated as regular common blocks.

The ANSI FORTRAN Standard does not provide for task common blocks.

4.7 SYMBOLIC NAMES

A symbolic name is the name of constant, variable, array, common block, main program, external function, subroutine, intrinsic function, statement function, block data subprogram, or dummy procedure. Each of these uses is referred to as a *class*.

A symbolic name consists of from one to eight alphanumeric characters, the first of which must be a letter (see below for a restriction on symbolic names identifying global entities). Some sequences of characters, such as format edit descriptors and keywords that uniquely identify certain statements (GO TO, READ, FORMAT, etc.) are not symbolic names, nor do they form the first characters of symbolic names in such occurrences.

The ANSI FORTRAN Standard provides for symbolic names of up to six alphanumeric characters.

4.7.1 SCOPE OF SYMBOLIC NAMES

The *scope* of a symbolic name is the range, within a program, where the name can be used. The scope can be an entire executable program, a program unit, a statement function statement, or an implied-DO list in a DATA statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program and are global to that program.

The names of variables, arrays, constants, statement functions, and intrinsic functions have the scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

The name of a variable appearing as a DO-variable in an implied-DO list in a DATA statement has a scope of that list.

4.7.1.1 Global entities

The main program, common blocks, subprograms, and external procedures are *global* entities of an executable program; that is, their scope is the entire program. A global name must not be used to identify any other global entity in the same executable program.

A global entity is identified by a symbolic name appearing in a common block, external function, subroutine, main program, or block data subprogram.

4.7.1.2 Local entities

A *local* entity's scope is a single program unit. A local entity is identified by a symbolic name appearing in an array, variable, constant, statement function, or intrinsic function. Names of local entities have the following restrictions:

- The name of a local entity must not also be the name of any other local entity in that program unit.

- The name of a global entity in a program unit must not also identify a local entity in that program unit except as noted in subsection 4.6.3, Common Block Names.

5. EXPRESSIONS AND ASSIGNMENT

An *expression* specifies either: a computation involving two or more *operands* with one or more *operators*; or, one operand and an optional + or -. Operands can be constants, symbolic names of constants, variables, array elements, substrings, or function references. Operators can specify arithmetic, assignment, character, relational, or logical operations.

Assignment statements, for defining variables and array elements during program execution, are arithmetic, logical, and character. Table 5-1 shows which data types can be used together in an assignment statement. The ASSIGN statement, for assigning statement labels, is described in section 6.

The use of functions in expressions is discussed in subsection 2.3, Functions.

An operand (a variable, array element, or function referenced in an expression) must be defined at the time the reference is executed. The name of a constant must be established in a PARAMETER statement preceding the statement where the constant is first referenced.

A parenthesized expression is treated as an entity. For example, in the expression A*(B*C), the product of B and C is evaluated and then multiplied by A. Parenthesized expressions can contain one or more parenthesized expressions, each of which can contain one or more parenthesized expressions, etc. The limit on the number of nested levels depends on the context.

The ANSI FORTRAN Standard does not limit the number of levels of nested parentheses.

Precedence among all types of operators is as follows:

Operator	Precedence
Arithmetic	Highest
Character	•
Relational	•
Logical	Lowest

An expression can contain more than one kind of operator. For example, the following logical expression contains arithmetic, relational, and logical operators:

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

This expression would be interpreted the same as the following expression:

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

The order in which primaries are combined using operators is determined by the following conditions, which are described in this section.

- Use of parentheses
- Precedence of operators
- Right-to-left interpretation of exponentiations in a factor
- Left-to-right interpretation of multiplications and divisions in a term
- Left-to-right interpretation of additions and subtractions in an arithmetic expression
- Left-to-right interpretation of concatenations in a character expression
- Left-to-right interpretation of conjunctions in a logical or masking term
- Left-to-right interpretation of inclusive disjunctions in a logical or masking disjunct
- Left-to-right interpretation of exclusive disjunctions, equivalences and nonequivalences in a logical or masking expression

5.1 ARITHMETIC EXPRESSIONS

An *arithmetic expression* specifies a numeric computation. Its evaluation produces a scalar or an array value.

The simplest form of an arithmetic expression is an unsigned constant or the symbolic name of a constant, variable, array element, or function. More complicated arithmetic expressions are formed by using one or more arithmetic operands with arithmetic operators and parentheses. As shown in table 5-3, arithmetic operands are normally of type integer, real, double-precision, or complex; type Boolean can also be used, with the qualifications shown in the table.

An *arithmetic constant expression* is an arithmetic expression that contains as operands any combination of arithmetic constants, symbolic names of arithmetic constants, or arithmetic constant expressions. Exponents (y in expression x**y) must be of type integer. References to variables, array elements, or functions are not permitted. Arithmetic constant expressions are required in some contexts such as PARAMETER statements.

An arithmetic operation that cannot be mathematically defined has unpredictable results. Terms that are not needed for an expressions's result are not always evaluated. Expressions that raise 0 to a 0 or negative power or that raise a negative value to a noninteger power cause run-time faults; these are detected and reported in different ways, depending on their nature.

5.1.1 ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement defines the entity on the left of the equal sign, *v*, to be the value of the expression on the right, *e*, whose type is converted to that of *v* if required. Table 5-1 shows which combinations of operands are legal and which require conversion.

```
v = e
```

v Name of a variable or array element of type integer, real, double-precision, or complex

e Arithmetic expression

Examples:

```
L = 12
C = (0.8,16.5) - (16.32,-6.1)
X = -B +(B**2-4*A*C)**0.5
A = B + L
ROOT = SQRT(65536.0)
ARRAY(6,2,1) = 0
MATRIX(I,J,K) = MATRIX(I,J,K)+1
```

Table 5-1. Allowed Assignment Statements: y=x

y=x	y=I	y=R	y=D	y=C	y=B	y=L	y=S
I=x	I=I	I=$\underline{R}$	I=$\underline{D}$	I=$\underline{C}$	I=B	I■L	I■S[1]
R=x	R=$\underline{I}$	R=R	R=$\underline{D}$	R=$\underline{C}$	R=B	R■L	R■S[1]
D=x	D=$\underline{I}$	D=$\underline{R}$	D=D	D=$\underline{C}$	D■B	D■L	D■S
C=x	C=$\underline{I}$	C=$\underline{R}$	C=$\underline{D}$	C=C	C■B	C■L	C■S
L=x	L■I	L■R	L■D	L■C	L■B	L=L	L■S
S=x	S■I	S■R	S■D	S■C	S■B	S■L	S=S

Boolean variables can never appear on the left side of an equal sign.

Legend:

= The assignment is legal.

_ An underscored letter indicates the x expression on the right side of the equal sign is converted to the type of the variable on the left side.

■ The expression on the right cannot be assigned to the variable on the left.

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character string		

[1] If S is a literal character string and length (S) ≤8 characters, the assignment is performed with S treated as a Hollerith constant; a warning message is issued. See the following heading, Character Expressions.

5.1.2 ARITHMETIC OPERATORS

Arithmetic operators and their interpretations are shown in table 5-2. Each arithmetic operator operates on a pair of operands and appears between them. The operators + and - can also operate on a single operand when it precedes that operand.

The interpretation of a division operation depends on the type of the operands. An *integer quotient* is the integer portion of a quotient having an integer divisor and dividend. For example, the expression -5/2 yields an integer quotient of -2.

5.1.2.1 Precedence of arithmetic operators

In an arithmetic expression, quantities enclosed in parentheses are evaluated first. If parentheses are within parentheses, the innermost quantity is evaluated first. Then the operations are evaluated in the following order:

Highest:	**
Second:	* and /
Lowest:	+ and -

For example, in the expression -A**2, ** has precedence over - . Therefore the result of the exponentiation is used as the operand for the negation. Thus, -A**2 is interpreted as -(A**2).

When an expression involves two or more operations of the same precedence, their position within the expression determines the order of their evaluation. Exponentiation (**) is evaluated right to left. Multiplication (*), division (/), addition (+), and subtraction (-) are evaluated left to right. There is one exception to the left-to-right evaluation of addition and multiplication operators; the CFT compiler uses the associative laws of addition and multiplication in order to optimize code, and freely moves operands within an expression that fits the associative law of addition or multiplication.

Table 5-2. Arithmetic Operators and Their Use in Expressions

Use of Operator	Interpretation
X**Y	Exponentiate X to the power Y
X/Y	Divide X by Y
X*Y	Multiply X by Y
X-Y	Subtract Y from X
-Y	Negate Y
X+Y	Add X to Y
+Y	(Same as Y)

5.1.3 ARITHMETIC OPERANDS

An *arithmetic operand* is an entity representing a number, which can be manipulated by an arithmetic operator. Arithmetic operands can be any of the following:

- Primaries
- Factors
- Terms
- Arithmetic expressions

The following subsections describe the forms of combining operands and operators in arithmetic expressions.

5.1.3.1 Primaries

A *primary* is the most basic unit at one level of syntax, but can itself be an expression with its own syntax. Primaries can be any of the following:

- Unsigned arithmetic constants
- Symbolic names of arithmetic constants
- Variable references
- Array element references
- Function references
- Arithmetic expressions enclosed in parentheses

Examples:

Primary	Description
23D9	Unsigned double-precision constant
KVALUE	Integer constant name if named in a PARAMETER statement
COUNTER8	Real variable name
IMAG(3,52,75)	Complex array element name if declared in a COMPLEX statement
EVAL(A,B,C)	Real function name if declared in a FUNCTION or statement function statement
(A/B**2)	Parenthesized arithmetic expression

5.1.3.2 Factors

A *factor* is a sequence of one or more primaries, separated by the exponentiation operator. The forms of a factor are:

- *primary*
- *primary* ** *factor*

The second form above indicates that for interpreting a factor containing two or more exponentiation operators, the primaries must be combined from right to left. For example, the factor 2**3**2 is interpreted as 2**(3**2).

5.1.3.3 Terms

A *term* is a factor or a sequence of factors separated by multiplication or division operators. A term can take any of the following forms:

- *factor*
- *term* / *factor*
- *term* * *factor*

The second and third forms show that a single term can include both the * and / operators. The factors are combined from left to right in interpreting a term containing two or more multiplication or division operators.

5.1.3.4 Arithmetic expressions

An *arithmetic expression* is a term or sequence of terms separated by addition (+) or a subtraction (-) operators. The forms of an *arithmetic expression* are as follows:

- *term*
- + *term*
- - *term*
- *arithmetic expression* + *term*
- *arithmetic expression* - *term*

The first term in an arithmetic expression can be preceded by an identity (+) or negation (-) operator. The last two forms show that terms are combined from left to right in interpreting an arithmetic expression containing two or more addition or subtraction operators.

These formation rules prohibit expressions containing two consecutive arithmetic operators such as A**-B or A+-B. However, expressions such as A**(-B) and A+(-B) are permitted.

5.1.4 DATA TYPE OF ARITHMETIC EXPRESSIONS

The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands. The data types of arithmetic expressions are given in table 5-3, and those for exponentiation in table 5-4. Each table item represents an expression and a result, and each capital letter represents an operand or result, as follows:

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character string		

A comma represents an arithmetic operator in table 5-3. If an arrow is shown, it points to the result type; a square symbol with no arrow indicates the two operands cannot be used in an arithmetic expression. In each row and column, one operand has the same data type throughout, and the other operand changes; the unchanging operand is indicated in the column head and at the beginning of each row.

Each table entry is similar in form to an assignment statement, but also applies to expressions within parentheses.

Example:

$R \leftarrow \underline{I},R$

To the right of the arrow, the comma is an operator and I and R are operands of types integer and real. To the left of the arrow, R indicates a real result; the underscored I indicates that integers in such an expression are converted to type real. The preceding entry would apply to the following expression:

```
CTEMP * 9/5 + 32
```

This would be treated as

```
CTEMP * 9./5. + 32.
```

When + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand.

Table 5-3. Use of Data Types with Arithmetic Operations: +, -, *, /

	y←I,x	y←R,x	y←D,x	y←C,x	y←B,x	y←S,x
y←x,I	I←I,I	R←R,$\underline{I}$	D←D,$\underline{I}$	C←C,$\underline{I}$	[1]I←B,I	[3]■ S,I
y←x,R	R←$\underline{I}$,R	R←R,R	D←D,$\underline{R}$	C←C,$\underline{R}$	[2]R←B,R	[3]■ S,R
y←x,D	D←$\underline{I}$,D	D←$\underline{R}$,D	D←D,D	C←C,$\underline{D}$	■ B,D	■ S,D
y←x,C	C←$\underline{I}$,C	C←$\underline{R}$,C	C←$\underline{D}$,C	C←C,C	■ B,C	■ S,C
y←x,B	[1]I←I,B	[2]R←R,B	■ D,B	■ C,B	[1]B←B,B	■ S,B
y←x,S	[3]■ I,S	[3]■ R,S	■ D,S	■ C,S	■ B,S	[3]■ S,S

Legend:

x One of two operands. A capital letter represents another operand of the indicated data type. Logical is never allowed.

, Arithmetic operator: + - * / . Two operands and an operator are an expression.

y Result of arithmetic operation. A letter to the left of the ← symbol represents a result of the indicated data type.

■ Error

_ Underscore indicates that the indicated operand's data type is converted before computation.

I	Integer	R	Real	D	Double-precision
C	Complex	B	Boolean	S	Character string

1 Integer operation; no conversion is performed on the Boolean item.

2 Real operation; no conversion is performed on the Boolean item.

3 If S is a literal character string and length (S) ≤8 characters, then the operation is performed with S treated as Hollerith; a warning message is issued. See subsection 5.2, Character Expressions.

The ANSI FORTRAN Standard does not allow mixing complex and double precision types in arithmetic operations.

Table 5-4. Data Types in Exponentiation: **

	y←I**x	y←R**x	y←D**x	y←C**x
y←x**I	I←I**I	R←R**I	D←D**I	C←C**I
y←x**R	R←$\underline{I}$**R	R←R**R	D←D**R	C←C**R
y←x**D	D←$\underline{I}$**D	D←$\underline{R}$**D	D←D**D	[1]C←C**$\underline{D}$
y←x**C	C←$\underline{I}$**C	C←$\underline{R}$**C	C←$\underline{D}$**C	C←C**C

Types Boolean, logical, and character cannot be used in exponentiation.

Legend:

x One of two operands. A capital letter represents an operand of the indicated data type.

** Exponentiation operator

y Result of exponentiation operation. A letter to the left of the ← symbol represents a result of the indicated type.

■ Error

_ Underscore indicates that the indicated operand's data type is converted before the operation is performed.

I Integer R Real D Double-precision C Complex

1 The double-precision exponent is converted to type Real.

The ANSI FORTRAN Standard does not allow mixing complex and double-precision types in exponentiation operations.

In general, conversion of data types is determined by a hierarchy of types, and is always upward within the hierarchy. The hierarchy is as follows:

Complex
- -
Double-precision
- -
Real
- -
Integer

Because type Boolean is not converted, it does not fit in the hierarchy; it is used only with types integer and real. Types character and logical are not allowed in arithmetic expressions.

In an expression operating on either a single operand or a pair of operands, the type and interpretation are independent of the context where the expression appears, and independent of the type of any other operand of a larger expression where the expression appears.

5.1.4.1 Type conversion

Type conversion of operands can occur during an expression's evaluation or when the results of an expression's evaluation are stored into a variable or array element.

Numerical type conversion is based on the following two operations.

(a) Integer-to-real conversion creates a real value from an integer value. The absolute value of the integer must be less than 2**46; no warning is issued if the value exceeds this range.

(b) Real-to-integer conversion creates a 64-bit integer value from a real value. The absolute value of the number being converted must be less than 2**46; no warning is issued if the value exceeds the range. The fractional part is truncated.

Type integer - Type integer is converted to type real as described in item (a) above. Integer is converted to double-precision by converting to real and adding zeros to extend the precision of the value. For converting integer to complex, the integer is converted to real as described in item (a) above; then the integer value becomes the real portion of a complex value, and zero is the imaginary portion.

Type real - Real is converted to integer as described in item (b) above. For converting to double-precision, zeros are added as the least significant portion to extend the real number's precision. For converting real to complex, the real value becomes the real portion of a complex value and zero is the imaginary portion.

Type complex - Complex is converted to integer by converting the real portion of the complex value as described for the real value in item (b) above. For converting complex to real, the real portion of the complex value becomes the real value. For converting complex to double-precision, zeros are added to extend the precision of the real portion of the complex value; this extended real portion becomes the double-precision value.

Type double-precision - Double-precision is converted to integer by converting the most significant portion as in item (b) above. For converting to real, the most significant portion of the double-precision value is used as the real value; no rounding occurs. For converting to complex, the most significant portion of the double-precision value becomes the real portion of the complex value, and zero is the imaginary portion; no rounding occurs.

The ANSI FORTRAN Standard does not provide for converting double-precision to complex.

5.1.5 CONSIDERATIONS IN EVALUATING ARITHMETIC EXPRESSIONS

In expressions that include integers, truncation can cause unintended results. Example:

```
    (3.0*10) / 3  equals  30.0 / 3  equals  10.0

    3.0*(10/3)  equals  3.0*3  equals  9.0
```

Because the expression 10/3 gives an integer quotient, the placement of parentheses in the two expressions above changes the result.

In addition to truncation in the use of integers, results can be affected by round-off errors and finite approximation of real numbers. For example, the difference between 5./10. and 5.*.1 is a computational difference; that is, although the two expressions are mathematically equivalent, the resulting values in a computer program could be different. Predicting and controlling computational differences is beyond the scope of this book but is discussed in various textbooks.

In addition to parentheses required for the intended interpretation of an expression, other parentheses can be included to control the magnitude and accuracy of intermediate values in the evaluation of an expression.

Example:

```
    A+(B-C)
```

The term (B-C) above is evaluated and then added to A. Removing parentheses could change the computed value.

5.2 CHARACTER EXPRESSIONS

A *character expression* is one character primary or a sequence of character primaries joined by the concatenation operator // . A *character primary* is one of the following:

- A character constant or the symbolic name of a character constant
- A character substring
- A variable, array element, or function reference of type character
- A character expression enclosed in parentheses

Relational expressions involving character operands are discussed in subsection 5.3, Relation Expressions.

5.2.1 CHARACTER ASSIGNMENT STATEMENT

Execution of a character assignment statement causes the evaluation of a character expression *ce* and the definition of character entity *cv* with the value of *ce*.

```
 ___________
|           |
| cv = ce   |
|___________|
```

cv Name of a variable, array element, or substring of type character

ce Character expression

Where appropriate, *ce* is either truncated or padded with blanks on the right to match the length of *cv*. No character positions defined in *cv* can be referenced in *ce*.

5.2.2 CHARACTER EXPRESSION EVALUATION

The result of evaluating a character expression is always of type character. Primaries are combined from left to right. Example:

```
    CHARACTER*2 VAR1,VAR2    VAR1='CR' VAR2='AY'   ...
    PRINT *,VAR1//VAR2
```

The preceding sequence produces the following printed result:

```
    CRAY
```

5.2.3 HOLLERITH TYPE

The older Hollerith type is described in appendix E, Outmoded Features. Because CFT supports the representation of Hollerith using the same conventions as the character type, the compiler must determine which type applies in ambiguous situations.

A character constant is treated as Hollerith in contexts where a character constant is illegal and a Hollerith constant is legal. Tables 5-1, 5-2, 5-5, and 5-8 indicate which categories of expressions fall into this category. This "passive" Hollerith typing (that is, letting the compiler determine that a constant is Hollerith) is not encouraged, and a warning message is issued. If you intend a constant to be Hollerith, specify it by following the second delimiter with H; for example 'ABC'H.

5.3 RELATIONAL EXPRESSIONS

A *relational expression* compares the values of two arithmetic or character expressions, producing a logical value of true or false. Relational expressions can appear within logical expressions. Relational operators are as follows:

Operator	Operation (Comparison)
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to
.GT.	Greater than

Relational operators have no precedence within this group because the use of more than one operator in the same relational expression is illegal.

Table 5-5 shows which data types can be used together in relational operations. The result type is logical for all relational operations.

Table 5-5. Data Types in Relational Operations: .EQ.,.NE.,.GT.,.GE.,.LT.,LE.

x∘z z x	I	R	D	C	B	L	S
I	L	L	L	L[1]	L[3]	■	■[2]
R	L	L	L	L[1]	L[3]	■	■[2]
D	L	L	L	L[1]	■	■	■
C	L[1]	L[1]	L[1]	■	■	■	■
B	L[3]	L[3]	■	■	L	■	■
L	■	■	■	■	■	■	■
S	■[2]	■[2]	■	■	■	■	L

Legend:

x∘z	Relational operation: .EQ.,.NE.,.GT.,.GE.,.LT.,LE.
■	Prohibited

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character string		

1 Only .EQ. and .NE. are allowed for complex comparisons.

2 If S is a literal and length (S) $\leq$ 8 characters, the operation is performed with S treated as a Hollerith constant; a warning message is issued. See the preceding heading, Character Expressions.

3 The comparison is performed without conversion of either operand.

5.3.1 ARITHMETIC RELATIONAL EXPRESSIONS

An *arithmetic relational expression* is a relational expression whose operands are arithmetic expressions or arithmetic array expressions. An arithmetic relational expression is interpreted as the logical value TRUE if the values of the expressions satisfy the relation specified by the operator; FALSE if they do not. If the operands are arrays, an array of logical values is returned.

A complex expression is permitted only when the relational operator is .EQ. or .NE.

In the relational expression *x relop y,* if arithmetic expressions *x* and *y* are of different types, the expression is evaluated as (*x*-*y*) *relop* 0. The expression in parentheses would be evaluated according to the type conventions for arithmetic expressions, as shown in table 5-3.

Examples:

```
    A .LE. B
    INDEX .EQ. ENDVALU
    J(1,6,6)*COS(ALPHA/10.) .GT. Z
    3.1415927 .LT. (22./7.)
    CMPLXM .NE. CMPLXN
```

5.3.2 CHARACTER RELATIONAL EXPRESSIONS

A *character relational expression* is a relational expression in which both operands are character expressions. The result is interpreted as the logical value TRUE if the values of the operands satisfy the relation specified by the operator; otherwise, the result is interpreted as the logical value FALSE.

The character expression that comes first in the collating sequence (see Appendix A) is considered to be of lower value. If the operands are of unequal length, the shorter operand is extended on the right with blanks to the length of the longer operand.

5.4 LOGICAL EXPRESSIONS

A *logical expression* specifies a logical computation. Evaluation of a logical expression produces a result of type logical with a value of either TRUE or FALSE. Tables 5-6 and 5-7 show the *logical operators* and their usage. Table 5-8 shows that operands for logical expressions are always of type logical; table 5-4 shows that only logical entities can be used in logical assignment statements.

5.4.1 LOGICAL ASSIGNMENT STATEMENT

Execution of a logical assignment statement causes the evaluation of logical expression *le* and the definition of logical entity *lv* with the value of *le*.

```
 __________
|          |
| lv = le  |
|__________|
```

lv Name of a logical variable, logical array, logical array section, or array element

le Logical expression

Examples:

All variable and array element names are assumed to be of type logical except E and F, which are type real.

```
T = .FALSE.
A = B
C = (A .AND. B) .OR. (C .AND. D)
T = .NOT. T
TRUTAB(I,J,K,L) = .T.
T = E.GE.F .OR. E/F .LT. .4
T = A .EQV. B
```

5.4.2 LOGICAL OPERATORS

The logical operators are shown in table 5-6; their use with operands is shown in table 5-7. The .NOT. or .N. operator produces the logical complement of its operand.

For logical expressions containing two or more logical operators, the precedence, as shown in table 5-6, determines the order in which they are to be combined (unless changed by the use of parentheses).

Example:

```
A .OR. B .AND. C
```

In the preceding expression, .AND. has higher precedence than .OR. Therefore the expression is interpreted as follows:

```
A .OR. (B .AND. C)
```

Logical operators can also be written as functions; for example A.AND.B can be written as AND(A,B). .NOT. is written as COMPL(), and .NOT..OR. is replaced by EQV(). See appendix B.

Table 5-6. Logical Operators

Operator	Operation	Precedence
.NOT. or .N.	Logical negation	Highest
.AND. or .A.	Logical conjunction	.
.OR. or .O.	Logical inclusive disjunction	.
.XOR. or .X. or .NEQV.	Logical exclusive disjunction or logical nonequivalence	.
.EQV.	Logical equivalence	Lowest

The ANSI FORTRAN Standard does not provide for the .XOR. operator or for .N., .A., .O., or .X. as abbreviations.

Table 5-7. Meanings of Logical Operators

X1	X2	X1.AND.X2	X1.)R.X2	X1.XOR.X2 X1.NEQV.X2	X1.EQV.X2
true	true	true	true	false	true
true	false	false	true	true	false
false	true	false	true	true	false
false	false	false	false	false	true

5.4.3 FORM AND INTERPRETATION OF LOGICAL EXPRESSIONS

A *logical operand* is an entity that can be operated on by a logical operator. Logical operands can be any of the following:

- Logical primaries
- Logical factors
- Logical terms
- Logical disjuncts
- Logical expressions

A *logical primary* is a primary in a logical expression. Logical primaries can be any of the following:

- Logical constants
- Symbolic names of logical constants
- Logical variable or array element references
- Logical function references
- Relational expressions
- Logical expressions enclosed in parentheses

A *logical factor* consists of a logical primary with or without the .NOT. operator. The form of a logical factor is:

[.NOT.] *logical-primary*

A *logical term* is a sequence of logical factors separated by an .AND. operator. If a logical term contains two or more .AND. operators, the logical factors are combined from left to right. The form of a logical term is:

[*logical-term* .AND.] *logical-factor*

A *logical disjunct* is a sequence of logical terms separated by an .OR. operator. If a logical disjunct contains two or more .OR. operators, the logical terms are combined from left to right. The form of a logical disjunct is:

[*logical-disjunct* .OR.] *logical-term*

A *logical expression* is a sequence of logical disjuncts separated by .XOR., .EQV., or .NEQV. operators. If a logical expression contains two or more .XOR., .EQV., and/or .NEQV. operators, the logical disjuncts are combined from left to right. The forms of a logical expression are:

[*logical expression* .XOR.] *logical disjunct*
[*logical expression* .EQV.] *logical disjunct*
[*logical expression* .NEQV.] *logical disjunct*

These forms allow the logical operator .NOT. to immediately follow any other logical operator. For example, the following logical term is permitted:

```
    LOGICALX .AND. .NOT. LOGICALY
```

5.5 MASKING EXPRESSIONS

A *masking expression* is an expression using a logical operator on integer, real, or Boolean operands, giving a result of type Boolean. Each operand is treated as a single storage unit (a 64-bit Cray word), and the result is a single storage unit.

The ANSI FORTRAN Standard does not provide for masking expressions.

Masking operators can also be written as functions; for example A.AND.B can be written as AND(A,B). .NOT. is written as COMPL(); .NOT..OR. is replaced by EQV(). See table B-8 in appendix B. The same table shows other functions that operate on Boolean values, such as shifting, parity count, and tallying 1's or leading 0's.

Table 5-8 shows which data types can be used together in masking operations. Letters in the table indicate the result type for each allowed operation. A masking expression cannot have operands of type logical, double precision, or complex.

Masking expressions can be combined with expressions of Boolean or other types by using arithmetic, relational, and logical (masking) operators. Evaluation of an arithmetic or relational operator processes a masking expression with no type conversion. Boolean data is never converted to another type.

A logical (masking) operator processing a masking expression performs the indicated logical operation separately on each bit. The interpretation of individual bits in masking factors, terms, and expressions is the same as described in the preceding subsection, Logical Expressions. The results of binary 1 and 0 correspond to the logical results TRUE and FALSE, respectively, in each of 64 bit positions. These values are summarized as follows:

$$\begin{array}{rrrrrrrrrr}
\text{.NOT.} & 1100 & & 1100 & & 1100 & & 1100 & & 1100 \\
 & =0011 & \text{.AND.} & \underline{1010} & \text{.OR.} & \underline{1010} & \text{.XOR.} & \underline{1010} & \text{.EQV.} & \underline{1010} \\
 & & & 1000 & & 1110 & & 0110 & & 1001
\end{array}$$

Table 5-8. Allowed Logical and Masking Operations and Result Types

y x	I	R	B	L	S
I	B	B	B	■	■[1]
R	B	B	B	■	■[1]
B	B	B	B	■	■[1]
L	■	■	■	L	■
S	■[1]	■[1]	■[1]	■	■

Types complex and double precision cannot be used in logical or masking operations.

Legend:

x,y Operands for a masking or logical expression, using operands .NOT., .AND., .OR., .XOR., and .EQV.

Entries in table:

B Masking operation with result type Boolean

L Logical operation with result type logical

■ Prohibited

I	Integer	R	Real	D	Double-precision	C	Complex
B	Boolean	L	Logical	S	Character string		

[1] If S is a literal and length (S) ≤8 characters, the operation is performed with S treated as a Hollerith constant. A warning message is also issued. See the preceding subsection, Character Expressions.

6. PROGRAM CONTROL STATEMENTS

Program control statements are used when two or more alternative sequences of statements exist and a decision is required, or when a statement sequence is to be repeated, interrupted, or terminated.

The following statements control an execution sequence.

- Unconditional GOTO
- Computed GOTO
- Assigned GOTO
- Arithmetic IF
- Logical IF
- Conditional block statements
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL (described in section 2)
- RETURN (described in section 2)

6.1 GOTO STATEMENTS

GOTO statements specify other statement within the same program unit to which control is transferred. Their use is discouraged for good program structure.

6.1.1 CONDITIONAL GOTO STATEMENT

Execution of an unconditional GOTO statement causes a transfer of control to the statement identified by the statement label. The space between GO and TO is optional.

```
 _________
|         |
| GOTO s  |
|_________|
```

s   Statement label of an executable statement in the same program unit

Example:

```
    GOTO 910
```

6.1.2 COMPUTED GOTO STATEMENT

Execution of a computed GOTO statement causes the expression *e* to be evaluated for an integer result, *i*. A transfer of control to the statement identified by the *i*th statement label in the list of *n* statement labels is then executed if $1 \le i \le n$. If $i<1$ or $i>n$, the execution sequence proceeds as though a CONTINUE statement were executed. If the evaluation of *e* for *i* produces a noninteger result, *e* is converted to integer as if *i*=*e* had been executed. A space between GO and TO is optional.

Format:

```
GOTO (s[,s]...)[,]e
```

e Expression

s Statement label of an executable statement that appears in the same program unit as the computed GOTO statement. A given statement label can appear more than once in a computed GOTO statement.

Examples:

```
    GOTO (2,4,8,16)A      (The value of A is truncated, if necessary, to
                           produce an integer value.)

    GOTO (0031,59,728)IX

    GOTO (0031,59,728)MSIZE/2

    GOTO (6,3,6,6,7,2,7),NBRANCH
```

The ANSI FORTRAN Standard specifies that the expression *e* in a computed GOTO statement must be an integer expression only.

6.1.3 ASSIGNED GOTO STATEMENT

At the time of execution of an assigned GOTO statement, the variable *i* must be defined with the value of a statement label of an executable statement appearing in the same program unit. The variable can be defined with a statement label value only by an ASSIGN statement in the same program unit as the assigned GOTO statement. Execution of the assigned GOTO statement causes a transfer of control to the statement identified by that statement label. The space between GO and TO is optional.

Format:

```
GOTO i[[,] (s[,s]...)]
```

i Integer variable name

s Statement label of an executable statement that appears in the same program unit as the assigned GOTO statement. A given statement label can appear more than once in this statement.

The ANSI FORTRAN Standard specifies that if the optional list is present, *i* must have been assigned a statement label from the list.

Examples:

```
(1)       ASSIGN 76 TO LAB
          ...
          GOTO LAB

(2)       ASSIGN 999 TO KFIN
          ...
          GOTO KFIN (997,997,999)

(3)       ASSIGN 1 TO JAIL
          ...
          GOTO JAIL,(1,2,3,4,5)
```

6.2 ARITHMETIC IF STATEMENT

Execution of an arithmetic IF statement causes evaluation of the expression *e*. Control is transferred to one of the statements identified by s_1, s_2, or s_3 if the value of *e* is less than zero, equal to zero, or greater than zero, respectively.

Format:

IF (*e*) s_1,s_2,s_3

e Integer, real, or double-precision expression

s_1,s_2, and s_3
Statement labels of executable statements that appear in the same program unit as the arithmetic IF statement. The same statement label can appear more than once in this statement.

Examples:

```
IF (VTEST)  20,21,20

IF (B**2-4*A*C) 70,80,90
```

6.3 LOGICAL IF STATEMENT

Execution of a logical IF statement causes evaluation of the expression *e*. If the value of *e* is true, statement *st* is executed. If the value of *e* is false, statement *st* is not executed and the execution sequence proceeds as though a CONTINUE statement were executed. The execution of a function reference in the expression *e* may affect related entities in the statement *st*.

Format:

IF (*e*) *st*

e Logical expression

st Any executable statement other than a DO, END, block IF, ELSE IF, ELSE, END IF, or another logical IF statement.

Examples:

```
IF(K) K=.NOT.K

IF (A.EQ.B) GOTO 100
```

6.4 CONDITIONAL BLOCKS

A *conditional block* is a group of executable statements delimited by *conditional block statements,* which specify conditions under which the block is executed. Following is a list of the conditional block statements:

- Block IF
- ENDIF
- ELSEIF
- ELSE

The *IF-level* of a given statement is the number of block IF statements from the beginning of the program unit to that statement minus the number of END IF statements from the beginning of the program unit up to but not including that statement. The IF-level of the END statement of each program unit must always be zero.

An *IF-block* is a group of executable statements that are executed if the condition specified in the block's IF statement is true. The block is preceded by a block IF statement and followed by another conditional block statement (ENDIF, ELSEIF, or ELSE) of the same IF-level. An IF-block can be empty.

An *ELSEIF-block* is a group of executable statements that are executed if the IF block and preceding ELSEIF blocks at the same IF-level were not executed. The block begins with an ELSEIF statement a conditional block statement (ENDIF, ELSEIF, or ELSE) of the same IF-level following the group. The IF-level of the ELSEIF-block must be greater than or equal to 1. An ELSEIF-block may be empty.

An *ELSE-block* is a group of executable statements that are executed if no preceding ELSEIF block at the same IF-level was executed. An ELSE-block begins with an ELSE statement and ends with an ENDIF statement of the same IF-level. No other conditional block statement at the same level can appear after the ELSE statement or before the ENDIF statement. ELSE-blocks can be empty. Statement labels on ELSE statements are ignored.

A group of blocks must begin with a block IF statement and end with an ENDIF statement. No more than one block is executed within each level of blocks. This execution depends on the sequential evaluation of the conditional block statements.

The ELSEIF and ELSE statements are not required to accompany block IF statements. A block begins with a block IF, an ELSEIF, or an ELSE statement and continues until an ENDIF or the beginning of the next block is encountered. Control must not be transferred to a location within a block from outside that block.

Each statement in a block has an IF-level number assigned to it. The first block IF encountered is assigned IF-level 1. All following statements retain that IF-level number until either another block IF or an ENDIF statement is encountered. If another block IF is encountered, the IF-level number of that statement is incremented by one. The following statements reflect that IF-level number until another block IF or ENDIF statement is encountered. If an ENDIF statement is encountered, the IF-level is decremented by 1 and all following statements retain that IF-level number until a block IF or ENDIF is encountered.

6.4.1 BLOCK IF STATEMENT

Execution of the block IF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution sequence continues with the first statement in the IF-block. If the value of *e* is false, control is transferred to the next ENDIF, ELSEIF, or ELSE statement of the same IF-level. The block IF statement must always have a corresponding ENDIF statement of the same IF-level.

Format:

```
 _____________
|             |
| IF (e) THEN |
|_____________|
```

e Logical expression

If a block IF statement appears within the range of a DO loop, the entire block must appear within that range.

Transfer of control into an IF-block from outside the IF-block is prohibited.

6.4.2 ENDIF STATEMENT

The ENDIF statement indicates the end of an IF-level and must always have a corresponding block IF statement of the same IF-level. The ENDIF statement consists of the word ENDIF alone.

6.4.3 ELSEIF STATEMENT

The ELSEIF statement is executed if none of the preceding blocks has been executed.

Format:

```
ELSEIF (e) THEN
```

e Logical expression

Execution of the ELSEIF statement causes evaluation of the expression *e*. If the value of *e* is true, normal execution sequence continues with the first statement of the ELSEIF-block. If the value of *e* is false, control is transferred to the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the ELSEIF statement. Statement labels on ELSEIF statements are ignored.

Transfer of control into an ELSEIF-block from outside the ELSEIF-block is prohibited.

6.4.4 ELSE STATEMENT

The ELSE statement begins an ELSE block, described under the heading Conditional Blocks. The statement consists of the word ELSE alone; any statement label on an ELSE statement is ignored.

6.5 DO LOOPS

A *DO loop* consists of a DO statement and a set of statements to be executed repeatedly, as indicated by the *iteration count*. The number of repetitions is controlled by the *DO variable* in the DO statement. The range of a DO loop consists of all executable statements from the first executable statement following the DO statement and ending with the terminal statement of the DO loop.

A DO loop can appear within a DO loop and must be entirely contained within the outer DO-loop range. More than one DO loop can have the same terminal statement. However, no more than 15 DO loops can terminate on the same terminal statement.

The ANSI FORTRAN Standard does not specify a limit to the number of DO loops that can terminate on the same terminal statement.

A DO loop can appear within a conditional block but must be entirely contained within that block. If a block-IF statement appears within the range of a DO loop, the corresponding ENDIF statement must appear within the same DO loop.

A DO loop is either active or inactive. A DO loop is initially inactive and becomes active only when its DO statement is executed. Control must not transfer into the range of an inactive DO loop. An active DO loop becomes inactive under any of the following conditions.

- Its iteration count is tested and determined to be zero.
- A RETURN or STOP statement is executed in the same program unit.
- Control is transferred to a statement that is in the same program unit and is outside the range of the DO loop.
- It is in the range of another DO loop whose DO statement is executed.

When a DO loop becomes inactive, the DO variable retains its last defined value unless it became undefined due to earlier action.

6.5.1 DO VARIABLE

The DO variable is an index which, during the execution of the DO loop, is set to an initial value and incremented (or decremented) until its value reaches or exceeds the limit value. The DO variable can be used in subscript or nonsubscript calculations within the DO loop. The absolute value of an integer DO variable must not exceed 2^{23} -1.

The ANSI FORTRAN Standard does not limit the value of a DO variable.

6.5.2 EXECUTING A DO STATEMENT

Executing a DO statement initiates the following sequence of steps.

1. The initial, limit, and increment value expressions (e_1, e_2, and e_3) are evaluated, producing the *initial parameter* m_1, the *terminal parameter* m_2, and the *incrementation parameter* m_3. If necessary, types are converted to the type of the DO variable, according to the rules for arithmetic conversion. If e_3 has been omitted from the DO statement, m_3 is assigned a value of 1. m_3 can be positive or negative but must not be 0. If the DO variable is of type integer, then m_1, m_2, m_3 and $(m_2-m_1+m_3)$ must all be less than $2^{23}-1$ in absolute value.

The ANSI FORTRAN Standard does not limit the values of m or of the quantity $(m_2-m_1+m_3)$.

---

2. The DO variable *i* becomes defined with the value of the initial parameter m_1.

3. The iteration count is established as an integer value equal to the integer portion of the following expression:

   $(m_2-m_1+m_3)/m_3$

   The iteration count is 0 when the following conditions occur:

   $m_1>m_2$, and $m_3>0$ or
   $m_1<m_2$, and $m_3<0$.

   $m_3=0$ is not explicitly detected, but results in a floating-point error when the iteration count is evaluated at run time.

The iteration count must be less than 2^{23}. Once the iteration count is established, entities named in the initial, limit, and incrementation value expressions e_1, e_2, and e_3 can be redefined with no effect on loop control processing. The DO variable cannot be redefined by a subsequent nested DO statement.

At the completion of DO statement execution, loop control processing begins.

The ANSI FORTRAN Standard does not specify a maximum iteration count.

6.5.3 LOOP CONTROL PROCESSING

Loop control processing determines if execution in the range of the DO loop is required. If the iteration count is not 0, control transfers to the first statement in the range of the DO loop. If the iteration count is 0, the DO loop becomes inactive. However, specifying ON=J in the COS CFT control statement or -e j in the UNICOS CFT command overrides this feature and causes all DO loops to execute at least once. If, as a result, all DO loops sharing the terminal statement of this DO loop are inactive, control is transferred to the first executable statement after the terminal statement. However, if any DO loops sharing the terminal statement are active, execution resumes with incrementation processing, described as follows.

Statements in the range of a DO loop are executed until the terminal statement is reached.

Incrementation processing has the effect of performing the following steps in sequence.

1. The value of the DO variable is incremented by the value of m_3.

2. The iteration count is decremented by 1.

3. Execution continues with loop control processing of the same DO loop whose iteration count was decremented.

A DO variable can increase or decrease in value during incrementation processing.

Examples:

```
(1)   PARAMETER(N=50)
      DIMENSION TABLE (N)
      DO 2 I=1,N
      IF(TABLE(I))2,2,1
 1    TABLE(I)=-TABLE(I)
 2    TABLE(I)=-TABLE(I+1)
      ...

(2)   PARAMETER(I=2,J=200)
      DIMENSION GRID(I,J), PGRID(I,J)

      DO 22 L=J,1,-1
      PGRID(K,L) = GRID(K,L)
      IF(PGRID(K,L))21,22,22
 21   PGRID(K,L) = -PGRID(K,L)
 22   GRID(K,L) = 0
      ...

(3)   M=0
      DO 100 I=1,10
      J=I
      DO 100 K=1,5
      L=K
100   M=M+1
```

In the above example, I=11, J=10, K=6, L=5, and M=50 after the last statement is executed for the last time.

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

After execution of the previous statements and at the execution of the CONTINUE statement, I=11, J=10, K=5, and N=0. L is not defined by these statements.

6.5.4 DO STATEMEMT

A DO statement specifies necessary information to control the repeated execution of a set of statements.

Format:

DO $s[,]i = e_1,e_2[,e_3]$

s Statement label of an executable statement, called the *terminal statement*

i Name of an integer, real, or double-precision variable, called the DO variable

e_1, e_2, and e_3
Integer, real, or double-precision expressions specifying the initial value, limit value, and increment value, respectively, of the DO variable. If e_3 is omitted, a value of 1 is assumed.

6.5.5 TERMINAL STATEMENT

The *terminal statement* is an executable statement that ends the DO loop. The terminal statement of a DO loop must not be an unconditional GOTO, assigned GOTO, arithmetic IF, conditional block, RETURN, STOP, END, or another DO statement. If the terminal statement of a DO loop is a logical IF statement, it may contain any executable statement except a DO conditional block, END, or another logical IF.

Execution of the terminal statement occurs during a normal execution sequence or through transfer of control. If execution of the terminal statement does not cause a transfer of control, execution continues with incrementation processing, described as follows.

6.5.6 CONTINUE STATEMENT

Execution of a CONTINUE statement has no effect. It is commonly used as the terminal statement of a DO loop. As with any statement so used, the next statement executed depends on the result of DO-loop incrementation processing. This action is the result of DO loop processing and not of CONTINUE statement execution.

Example:

```
         DIMENSION ARRAY6(16)
         DO 22,I=16,1,-1
         IF(ARRAY6(I).NE.0) ARRAY6(I)=1.0/ARRAY6(I)
     22  CONTINUE
```

6.6 STOP STATEMENT

A STOP statement terminates execution of a main program, subroutine subprogram, or function subprogram.

Specification or nonspecification of *id* has no effect on the executable program. The characters specified by *id* appear in a logfile message to identify the STOP statement encountered during program execution.

Format:

```
 ____________
|            |
| STOP [id]  |
|____________|
```

id Unsigned integer constant of up to 8 digits, a character constant of up to 8 characters, or the symbolic name of a variable, array element, or function containing (or providing) 8 characters

The ANSI FORTRAN Standard limits noncharacter *id* to 5 digits, sets no limit on the length of character constants, and does not permit *id* to be the name of a variable, an array element, or a function.

6.7 PAUSE STATEMENT

A PAUSE statement suspends or terminates a main program, subroutine subprogram, or function subprogram. The PAUSE statement is seldom used in current computer systems but is retained so that older programs can be run.

An installation parameter determines whether the execution can be resumed or is unconditionally terminated. Specification or nonspecification of *id* has no effect on the executable program. The characters specified by *id* appear in a logfile message to identify the PAUSE statement encountered during program execution.

The ANSI FORTRAN Standard does not provide for the option of resuming or terminating execution.

Format:

```
 ______________
|              |
| PAUSE [id]   |
|______________|
```

id Unsigned integer constant of up to 8 digits, a character constant of up to 8 characters, or the symbolic name of a variable, array element, or function containing or providing 8 characters

---

The ANSI FORTRAN Standard limits noncharacter *id* to 5 digits, sets no limit on the length of character constants, and does not permit *id* to be the name of a variable, an array element, or a function.

6.8 END STATEMENT

An END statement is required at the physical end of the sequence of statements and lines of every program unit. When executed in a subprogram, it has the effect of a RETURN statement. When executed in a main program, it has the effect of a STOP statement.

No other statement in a program unit can be expressed with an initial line containing only an END statement. Embedded comments can be included on an END statement when preceded by an exclamation point.

The last line of every program unit must be an initial line containing a complete END statement. This special form of initial line is called a terminal line. A single END statement can appear with one or more STOP statements or with one or more RETURN statements in the same program unit.

7. INPUT/OUTPUT STATEMENTS

Input statements transfer data from a file to program memory. This process is called *reading*. *Output statements* transfer data from program memory to a file. This process is called *writing*. Some input/output (I/O) statements allow *formatting* (or *editing*) of the data as it is transferred.

The CFT statements that deal with I/O operations are summarized in table 7-1.

Table 7-1. CFT Input/Output Statements

Statement	Description
READ	Transfers data from a file to the program
WRITE	Transfers data from the program to a file
PRINT	Transfers data from the program to a file
FORMAT	Formats data that is transferred between the program and a file
BACKSPACE	Repositions a file before the preceding record
ENDFILE	Writes an end-of-file mark at a file's current position
REWIND	Repositions a file to the beginning of information
INQUIRE	Returns information about a file's properties
OPEN	Initializes a file for I/O operations
CLOSE	Finishes processing on a file and returns it to the operating system
NAMELIST†	Enables data transfer between the program and a file containing lists of variables with assigned values

† CFT extension

Table 7-1. CFT Input/Output Statements (continued)

Statement	Description
BUFFER IN†	Transfers data from a file to the program; allows immediate execution of following program statement.
BUFFER OUT†	Transfers data from the program to a file; allows immediate execution of following program statement.

† CFT extension

I/O statements perform operations on FORTRAN files. A FORTRAN file can be either *internal* or *external*. Internal files are internal to the program and cease to exist when the program terminates. External files are associated with the operating system; external files are associated with files in UNICOS, and with datasets in COS.

This section describes I/O operations with blocked files. The structure of COS blocked datasets is described in the COS Version 1 Reference Manual, publication SR-0011. The structure of UNICOS blocked files is described in the UNICOS File Formats and Special Files Reference Manual, publication SR-2014. Limited I/O operations for unblocked files are described in appendix H. Nonstandard random access I/O operations are described in appendix E.

7.1 INPUT/OUTPUT RECORDS

A *record* is the smallest entity that can be read or written by a FORTRAN I/O statement. A record is a sequence of values or characters. For example, a punched card is usually considered a record. For printed output, each print line is a record. A record may or may not correspond to a physical entity.

Records can be of the following types.

- Formatted
- Unformatted
- End-of-file or endfile
- End-of-data

The ANSI FORTRAN Standard does not provide for end-of-data records.

7.1.1 FORMATTED RECORDS

A formatted record consists of a sequence of characters. Its length, measured in characters or 8-bit bytes, depends primarily on the number of characters transferred when written. The length also depends on the peripheral device characteristics (for example, line printer or card reader) serving as the origin or ultimate destination of the data. Formatted records can be read or written by formatted I/O statements, or prepared by means other than FORTRAN.

Unformatted and buffered I/O statements can also read and write formatted records, but in a manner ignoring their formatted characteristics. Because of record blocking, reading formatted records with unformatted I/O statements may not be practical. The structure of COS blocked records is described in the COS Version 1 Reference Manual, publication SR-0011.

The ANSI FORTRAN Standard allows reading and writing of formatted records only by formatted I/O.

7.1.2 UNFORMATTED RECORDS

An unformatted record consists of a sequence of character and/or noncharacter data. The length of an unformatted record is measured in storage units (words) unless the record contains character data items. In that case, each character entity of length *len* takes (*len*+7)/8 words.

Unformatted records can be read or written by unformatted and buffered I/O statements.

The ANSI FORTRAN Standard does not allow reading and writing of unformatted records with formatted I/O.

7.1.3 END-OF-FILE (ENDFILE) RECORDS

An end-of-file (endfile) record occurs as the last record of a file. An endfile record can be written at the end of a file by an ENDFILE statement. The endfile record has no length property.

7.1.4 END-OF-DATA RECORDS

An end-of-data (EOD) record occurs as the last record of a COS dataset. An EOD record has no significance in UNICOS. It cannot be explicitly written by a FORTRAN program.

7.2 INPUT/OUTPUT FILES

A *file* is a sequence of records. A CFT file can contain formatted records, unformatted records, or a combination of formatted and unformatted records. All files are terminated with an end-of-file record.

The ANSI FORTRAN Standard does not provide for the mixing of formatted and unformatted records in a file.

An external file is associated with a COS dataset or a UNICOS file when the external file is created or initialized (opened) in a program. Internal files are not associated with operating system files or datasets.

7.3 COS DATASETS

Under COS, a *dataset* is a sequence of all files associated with a particular unit during program execution. Association of a dataset with a particular unit is controlled by the executing program. However, datasets and units can be preassociated before program execution. Datasets are described in the COS Version 1 Reference Manual, publication SR-0011.

UNICOS does not have datasets or any other multiple file entity.

The ANSI FORTRAN Standard does not provide for datasets or other multiple file entities.

The following program, executing under COS, creates a FORTRAN file named CIRCLES; reads an input file containing values for the diameters of circles; and writes the the diameters, an end-of-file mark, then the circumferences, to file CIRCLES.

```
      PROGRAM TWOFILES
      PARAMETER(PI=3.14)
      DIMENSION CIRCUM(10)
      OPEN(UNIT=1,FILE='CIRCLES',STATUS='NEW')
      J=1
    5 READ(100,*,END=10) DIAM     ! READ DIAMETER FROM INPUT FILE
      WRITE(1,*) DIAM             ! WRITE DIAMETER TO FILE CIRCLES
      CIRCUM(J)=DIAM * PI
      J=J+1
      GOTO 5
   10 ENDFILE 1                   ! WHEN DONE READING, WRITE EOF MARKER
   15 DO 20 I=1,J-1
      WRITE(1,*) CIRCUM(I)        ! WRITE CIRCUMFERENCE TO FILE CIRCLES
   20 CONTINUE
      CLOSE(1)
      END
/EOF
   ...data...
/EOF
```

CFT associates file CIRCLES with a COS dataset having the same name. Upon termination of the program, dataset CIRCLES contains two files. The first file contains the diameters; the second file contains the circumferences.

7.4 INTERNAL RECORDS AND FILES

Internal records and *internal files* provide a way to transfer and convert data within internal storage. This feature is useful when input data must be converted before it can be used. An internal file identifier is used for internal files in place of an external unit identifier.

An internal file is a character variable, character array element, character array, or character substring. A record of an internal file is a character variable, character array element, or character substring. If the internal file is a character variable, character array element, or character substring, it consists of a single record with the same length as the file. If the internal file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the internal file. The ordering of the file records is the same as the ordering of the array elements in the array. Every record of the file has the same length, the length of an array element in the array.

The contents of a record of the internal file are defined by writing the record. If the number of characters written in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

The internal file record can be defined or redefined by using statements other than an output statement, such as a character assignment statement.

An internal file is always positioned at the beginning of the first record before data transfer. Reading and writing records is done only by sequential access formatted I/O statements not specifying list-directed formatting.

Internal records and files are useful for editing input data before using it. The following statements read numbers from an input file that have been entered with a dollar sign preceding them. The statements place an input value in an internal file, remove the dollar sign, then transfer the value to a type REAL variable.

```
      INTEGER POSN
      CHARACTER *12 TEMP            ! DECLARE TYPE OF INTERNAL FILE TEMP
      ...
      READ(6,1)TEMP                 ! READ A RECORD FROM DEVICE 6 TO TEMP
    1 FORMAT(A10)
      IF (INDEX(TEMP,'$').NE.0) THEN        ! REPLACE DOLLAR
         POSN = INDEX(TEMP,'$')             ! SIGN WITH
         TEMP(POSN:POSN) = ' '              ! A BLANK
      ENDIF
      READ(TEMP,2) VALUE            ! READ RESULT INTO REAL VARIABLE
    2 FORMAT(F10.2)
```

7.5 EXTERNAL FILE ACCESS METHODS

There are two methods of accessing external files: *sequential access* and *direct access.*

- *Sequential access operations* are based on the sequential storage of records within files. The order of the records is the order in which they are written.

- In *direct access operations,* records can be read or written in any order. The order of record numbers determines the order of the records. All records of the file have the same length and each record of a file has a unique record number. The record number is a positive integer that is specified when the record is written. Once established, the record number cannot be changed. A record can be overwritten but not deleted. Records must not be read or written with list-directed or NAMELIST formatting. Multifile direct access COS datasets are not allowed.

If both sequential and direct access are allowed on a given file, the records can be in any order. The first record accessed by sequential access is the record numbered 1 for direct access, the second record accessed by sequential access is the record numbered 2 for direct access, and so on.

While a file is connected for one type of access (sequential or direct), only I/O statements using that type of access may be used with the file.

See appendix E for random access extensions.

7.5.1 FILE POSITION AFTER DATA TRANSFER

File position determines the next record of a file to be read or written. File position depends on the method of access, sequential access or direct access.

7.5.1.1 Sequential access

When an input operation is performed on a file, the file is positioned at the beginning of the next record, which becomes the current record. When an output operation is performed on a file, a new record is created, becoming the last record of the file.

The position of an internal file is always at the beginning of the character variable, array, array element, or substring referenced by the I/O operation.

7.5.1.2 Direct access

When an input operation is performed on a file, the file is positioned at the beginning of the record specified by the next higher record number, which becomes the current record. When an output operation is performed on a file, a new record is created with a record number one higher than the previous record number, becoming the last record of the file.

7.6 UNITS

A *unit* is a means of referring to a file or dataset. At any given time, a set of units *exists* for an executable program; these are the preconnected units and the units corresponding to open files. I/O statements can refer only to existing units.

7.7 IDENTIFIERS

Identifiers assign names to units, internal files, files, and formats.

7.7.1 UNIT IDENTIFIERS

An *external unit identifier* is used to refer to an external file and is an integer constant or expression in the range 0 through 102, or the character *. In COS, the values 100, 101, and 102 refer to datasets $IN, $OUT, and $PUNCH, respectively; in UNICOS, they refer to files *stdin, stdout,* and *stderr,* respectively. These assignments cannot be changed. The character * can only specify a unit preconnected for formatted, sequential access; it can appear only in a READ or WRITE statement. In a READ statement, * refers to $IN under COS and to *stdin* under UNICOS; in a WRITE statement, * refers to $OUT under COS and to *stdout* under UNICOS. The defaults for units 5 and 6 under COS are $IN and $OUT, respectively; under UNICOS, the defaults are *stdin* and *stdout,* respectively. These assignments can be changed. (See the COS Version 1 Reference Manual, publication SR-0011, or the UNICOS File Formats and Special Files Reference Manual, publication SR-2014, for further details.)

The ANSI FORTRAN Standard does not specify a maximum value for the external unit identifier.

An *internal file identifier* is used to refer to an internal file and is the name of a character variable, character array, character array element, or character substring.

If the optional characters UNIT= are omitted from the unit identifier, the unit identifier must be the first item in a list of identifiers.

The ANSI FORTRAN Standard does not provide for the definition of unit identifiers 100, 101, or 102 or for the preconnection of units 5 and 6.

7.7.2 FILE IDENTIFIERS

A file identifier is a constant value that is used to identify a particular file. It can be

- A character constant
- An integer variable
- An integer array element containing Hollerith data of not more than 7 characters

External file identifiers of type character can only be used in OPEN and INQUIRE statements. Using character type variables as unit identifiers in READ/WRITE statements implies I/O operations on internal files.

The ANSI FORTRAN Standard does not provide for file identifiers.

7.7.3 FORMAT IDENTIFIERS

A format identifier must be one of the following:

- A FORMAT statement label appearing in the same program unit as the format identifier
- An integer variable name with the following restrictions:
  - The integer variable name cannot also appear as a dummy argument in the same program unit.
  - An ASSIGN statement must assign the format label.
- A character array name
- A character expression not involving concatenation of an operand with an asterisk length specification unless the operand is the symbolic name of a constant
- An asterisk, specifying list-directed formatting

7.8 DATA TRANSFER STATEMENTS

The READ statement is the data transfer input statement; execution of a READ statement causes values to be transferred from an external or internal file to the entities specified in the input list, if present. WRITE and PRINT statements are data transfer output statements; execution of these statements causes values to be transferred from the entities specified in the output list, if present, to an external or internal file. The READ, WRITE, and PRINT statements have the following formats.

Format:

```
READ (cilist)[iolist]

READ f [,iolist]

WRITE (cilist)[iolist]

WRITE f [,iolist]

PRINT f [,iolist]
```

The ANSI FORTRAN Standard does not provide for the WRITE *f* [*,iolist*] format.

cilist Control information list; includes a reference to the source or destination of the data to be transferred and an optional format identifier.

f Format identifier

iolist I/O list specifying the data to be transferred

7.8.1 CONTROL INFORMATION LISTS

The format of a *control information list* (*cilist*) is as follows.

Format:

```
[UNIT=] {u  }[,[FMT=]f][,END=sn][,REC=rn][,ERR=s][,IOSTAT=ios]
        {fin}
```

[UNIT=] {*u* / *fin*}

Unit identifier (*u*) or file identifier (*fin*). Either *u* or *fin* must be specified, but not both. If the UNIT= keyword is omitted, *u* and *fin* are positional parameters and must appear first.

The ANSI FORTRAN Standard does not provide for the [UNIT=]*fin* form.

[FMT=]*f* Format identifier. This parameter must be present for formatted I/O statements. If *f* is an asterisk, the statement is list-directed and a record identifier cannot be present. If the optional UNIT= keyword is specified with the unit or file identifier, the FMT= keyword must be specified with the format identifier. If both the UNIT= and the FMT= keywords are omitted, *f* must follow *u* or *fin*.

END=*sn* End-of-file identifier. *sn* is the number of the statement where execution continues after an EOF on a READ statement has been encountered. An end-of-file identifier must not appear in a WRITE statement or in the same control information list as a record identifier.

REC=*rn* Record identifier. *rn* must be an integer expression with a positive value. A record identifier appears only in direct-access I/O statements. A statement containing a record identifier cannot contain an end-of-file identifier.

ERR=*s* Error identifier. *s* is the statement label of the statement where control continues after a recoverable error occurs.

IOSTAT=*ios*

Status identifier that becomes defined when an I/O statement is executed. *ios* must be an integer variable or an integer array element. Following are the identifier values and their meanings.

ios Value	Meaning
=0	Transfer is complete; no error or end-of-file condition exists.
>0	Error message number; see coded $IOLIB messages in COS Message Manual, publication SR-0039.
<0	End of file was encountered; no error condition exists.

Examples of control information lists:

```
READ(10)...                   READ(10,IOSTAT=JOE)...
WRITE(10,430)...              READ(98,12345,ERR=42,END=75)...
WRITE(10,REC=J)...            READ(J,ARRAYF,ERR=10,END=25)...
READ('FILE1',30)...           WRITE(98,32,ERR=37)...
READ(10,*)...                 READ(END=100,FMT=20,UNIT=5)...
READ(*,*)...                  WRITE(98,'(6E11.4)',ERR=75)...
READ(*,*,END=200)...          READ(10,IOSTAT=JOE,ERR=100,END=200)...
```

7.8.2 I/O LISTS

An *I/O list* (*iolist*) specifies entities whose values are transferred by I/O statements. This list is composed of one or more I/O list items separated by commas. Optionally, one or more implied DO lists can be included in the list.

An array name appearing as an I/O list item is treated as if all elements of the array were specified in the order given by array element ordering.

7.8.2.1 Input list items

Only input list items can appear in an input statement. An *input list item* must be one of the following.

- Variable name
- Array element name
- Array name
- Character substring name

7.8.2.2 Output list items

An *output list item* must be one of the following.

- Variable name
- Array element name
- Array name
- Character substring name
- An expression (the expression may not be a character expression involving concatenation of an operand with a length specification of (*), unless the operand is the symbolic name of a constant)

Examples of input and output lists:

```
READ(23)X,Y                !VARIABLE NAMES

WRITE(23)A(1),A(4),X(2)    !ARRAY ELEMENT NAMES

DIMENSION Z(64)
  ...
READ(23)Z                  !ARRAY NAME
```

```
    CHARACTER*10 WORD
     ...
    READ(23)WORD(2:3) !CHARACTER SUBSTRING

    WRITE(23)A+B   !EXPRESSION
```

7.8.2.3 Implied DO lists

The format of an implied DO list is as follows.

Format:

$(\mathit{dlist}, i = e_1, e_2[, e_3])$

dlist An I/O list

i, e_1, e_2, and e_3
As specified for the DO statement (see section 6)

The range of an implied DO list is the list *dlist*. *dlist* can itself contain one or more implied DO lists. The iteration count and the value of the DO variable *i* are established from e_1, e_2, and e_3 exactly as for a DO-loop. Once the values of *i* and of the iteration count are established, *i*, e_1, e_2, and e_3 can be redefined with no effect on the loop control process. The DO variable *i* can be specified as a subscript to array elements named in *dlist* for both input and output list items. When an implied DO list appears in an I/O list, it is treated as if *dlist* were specified once for each iteration of the implied DO list. If a premature exit from an implied DO occurs due to an I/O error or end-of-file, the loop indices become undefined.

Examples:

```
    PRINT 311,(VECTOR(I),I=1,100)

    READ(12,345)((XREF(M,N),M=1,N),N=1,3)

    WRITE(6,350)(M,(N,XREF(M,N),N=1,3),M=2,1,-1)

    READ(5,1,END=50,ERR=60)((BUFFER(I,J),I=1,20),J=1,1000)
```

7.9 DATA TRANSFER OPERATIONS

When a data transfer I/O statement (READ, WRITE, or PRINT) is executed, the following operations are performed in the order specified.

1. The direction of data transfer is determined (input for READ, output for WRITE and PRINT).
2. The unit involved in the transfer is identified.
3. The format (if specified) is established.
4. Data is transferred between the external or internal file and the entities specified by the I/O list (if any).
5. The status identifier (if specified) is defined.

7.9.1 IDENTIFYING A UNIT

A READ statement without a unit or file identifier specifies unit 100, which is preconnected to the COS dataset $IN or the UNICOS file stdin. PRINT and WRITE statements similarly specify unit 101, preconnected to the COS dataset $OUT or the UNICOS file stdout. Unit preconnection for READ, PRINT, and WRITE statements without a unit identifier is not under the control of the executing program.

If the file specified by the output statement does not exist, a file is created and the write proceeds normally. If the file specified by an input statement does not exist, an empty file is created.

The ANSI FORTRAN Standard does not provide for reading or writing a nonexistent file or dataset.

7.9.2 ESTABLISHING A FORMAT

A format identifier in a control information list identifies a format specification.

7.9.3 TRANSFERRING DATA

Data is transferred between records and entities specified in the I/O list. List items are processed in the order of their left-to-right appearance in the I/O list.

All values needed to determine entities specified by an I/O list item are determined at the beginning of the processing of that item. For example, the following statements cause a value to be read into N(3).

```
      N(1)=3
      READ(8)N(N(1))
```

All values are transmitted to or from the entities specified by a list item before the processing of any succeeding list item. For example, the following statement causes two values to be read.

```
    READ(3)N,A(N)
```

The first value read is assigned to N, and the second is assigned to A(N), where the new value of N is used as the subscript.

A DO variable in an implied DO list becomes defined at the beginning of processing the implied DO list as an I/O list item.

An input list item, or any entity associated with it, must not affect any portion of the established format specification.

7.9.3.1 Unformatted data transfer

During unformatted data transfer, data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written.

On input, the file should be positioned so the record read is an unformatted record or an endfile record.† The number of values required by the input list must be less than or equal to the number of values in the record and must not require more values than the record contains.

7.9.3.2 Formatted data transfer

During formatted data transfer, data is transferred with editing between the entities specified by the I/O list and the file. The current record and possibly additional records are read or written.

On input, the record read should be a formatted record or an endfile record.†

† CFT allows formatted and unformatted records on the same file or dataset (non-ANSI).

The I/O list and format specification must not specify more than 152 characters. Some formats larger than 133 characters generate warning errors. If the input record length is less than the input list requires, the additional characters are defined as blanks.

The ANSI FORTRAN Standard does not provide for a maximum number of characters per record, nor for blank padding if the record is less than that required for the input list.

7.9.4 OUTPUT TO A PRINTER

The transfer of formatted record information to certain devices is called *printing*. The first character of a formatted record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines the vertical spacing to occur before printing. The character codes specifying vertical spacing (carriage) control are shown in table 7-2.

Table 7-2. Print Control Characters

Character	Vertical Spacing Before Printing
Blank	Advance one line
0	Advance two lines
1	Advance to first line of next page
+	No advance
All other	Advance one line

If the record contains no characters, an advance of one line occurs and nothing is printed in that line. A PRINT statement does not necessarily result in a printing operation.

7.9.5 ERROR AND END-OF-FILE CONDITIONS

If an error condition occurs during data transfer, the position of the file is indeterminate.

If an end-of-file (EOF) condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or EOF condition exists, the file is positioned after the last record read or written.

If an error condition or EOF condition is encountered during a read operation, the read terminates and the entities specified in the I/O list become undefined.

7.10 SEQUENTIAL FILE STATEMENTS

The BACKSPACE, ENDFILE, and REWIND statements perform operations on sequential files. The BACKSPACE statement positions a file at the beginning of the previous record from the current record; the ENDFILE statement writes an end-of-file (EOF) mark at the file's current position; and the REWIND statement positions the file to the first record of the file.

Formats:

$$\text{BACKSPACE} \left\{ \begin{matrix} u \\ fin \\ (alist) \end{matrix} \right\}$$

$$\text{ENDFILE} \left\{ \begin{matrix} u \\ fin \\ (alist) \end{matrix} \right\}$$

$$\text{REWIND} \left\{ \begin{matrix} u \\ fin \\ (alist) \end{matrix} \right\}$$

u External unit identifier

fin File identifier whose value specifies the name of an external file

alist The following set of identifiers:

[UNIT=] *u* or *fin*
IOSTAT= *ios*
ERR= *s*

alist must contain a single external unit identifier or file identifier and can contain at most one of each of the other identifiers. See the UNIT, IOSTAT, and ERR identifiers described for the OPEN and CLOSE statement in tables 7-4 and 7-5, respectively.

The external unit or file specified in a BACKSPACE or ENDFILE statement must not be connected for direct access. If the external unit or file specified by a BACKSPACE, ENDFILE, or REWIND statement is not connected, it becomes connected and the file is created.

BACKSPACE, ENDFILE, and REWIND operations on internal files are not allowed.

The ANSI FORTRAN Standard does not provide for positioning of an unconnected file.

The ANSI FORTRAN Standard does not provide for the *fin* parameter on BACKSPACE, ENDFILE, or REWIND statements.

7.10.1 BACKSPACE STATEMENT

A BACKSPACE statement causes the file related to the specified unit to be positioned at the beginning of the preceding record. If no preceding record exists, the position of the file is unchanged. If the preceding record is an endfile record, the file is positioned before it.

The ANSI FORTRAN Standard does not provide for backspacing a file that is not connected, a file that is connected but does not exist, or one that has been written with list-directed format.

7.10.2 ENDFILE STATEMENT

An ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record.

After the execution of an ENDFILE statement, a BACKSPACE or REWIND statement must reposition the file before execution of an input statement. An output statement creates another file on the same file. Under COS, execution of an output statement on a file that is positioned after an endfile record creates another file on the same dataset (see the example in subsection 7.3, COS Datasets). Under UNICOS, output statements are not allowed to execute on a file that is positioned after an endfile record. Execution of an ENDFILE statement for a file that is connected but does not exist creates the file.

The ANSI FORTRAN Standard does not provide for the writing of an endfile on a file that is not connected.

7.10.3 REWIND STATEMENT

A REWIND statement causes the specified file to be positioned at its initial point. If the file is already positioned at its initial point, execution of the statement has no effect on the file position.

The ANSI FORTRAN Standard does not provide for the rewinding of an unconnected file or a file connected for direct access, nor does it provide for the creation of a connected file when one does not exist.

7.11 INQUIRE STATEMENT

An INQUIRE statement determines the current status of an external file's attribute. Inquiry can be made by file name or by unit number.

7.11.1 INQUIRY BY FILE NAME

The format of the INQUIRE by file name statement is as follows.

Format:

```
 ______________________________
|                              |
| INQUIRE (FILE=fin,islist)    |
|______________________________|
```

fin A character expression that specifies the name of the file. The file need not be connected to a unit. *fin* is limited to seven characters, not counting trailing blanks. Any trailing blanks are discarded.

islist A list of inquiry identifiers that contains at most one of each of the inquiry identifiers described in table 7-3.

7.11.2 INQUIRY BY UNIT

The format of the INQUIRE by unit statement follows.

Format:

```
INQUIRE(u,islist)
```

u An external unit identifier. (See subsection 7.7.1, Unit Identifiers.) The unit specified need not be connected to a file. If it is connected to a file, however, the inquiry includes the connected file.

islist A list of inquiry identifiers that contains at most one of each of the inquiry identifiers described in table 7-3.

The following example illustrates the use of the INQUIRE statement. The INQUIRE statement determines if a file NAMES exits; if it does not, the program creates an empty file NAMES.

```
LOGICAL AROUND
 .
 .
 .
INQUIRE(FILE='NAMES',EXIST=AROUND)
IF(.NOT.AROUND)OPEN(UNIT=6,FILE='NAMES',STATUS='NEW')
```

Table 7-3. INQUIRE Specifiers and Their Meanings

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
IOSTAT=*ios*	Integer variable or array element	Error status	(RV) 0 if no error condition exists; error message number if error condition exists
ERR=*s*	Statement label	Statement label where control is transferred if error condition exists	(I) FORTRAN statement label

Table 7-3. INQUIRE Specifiers and Their Meanings (continued)

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
EXIST=*ex*	Logical variable or array element	Existence specifier	(RV) .TRUE. if unit or file exists; else, .FALSE.
OPENED=*od*	Logical variable or array element	Connection specifier	(RV) .TRUE. if unit and dataset are connected; else, .FALSE.
NUMBER=*num*	Integer variable or array element	External unit specifier	(RV) Unit currently connected; if no unit, *num* is undefined
NAMED=*nmd*	Logical variable or array element	Unit name specifier	(RV) .TRUE. if unit has a name; else, .FALSE.
RECL=*rcl*	Integer variable or array element	Record length of unit or file connected for direct access	(RV) Record length in characters. (For unformatted I/O, the record length is a positive integer multiple of eight.) If not connected for direct access, *rcl* is undefined.
NEXTREC=*nr*	Integer variable or array element	Next record	(RV) The record number that follows the last record read or written for direct access. If none have been written, *nr*=1. If access is not direct, *nr* is undefined.
NAME=*fn*	Character variable or array element	File name	(RV) File name if file has a name; else, *fn* is undefined.
ACCESS=*acc*	Character variable or array element	Access specifier	(RV) 'SEQUENTIAL' is access method; 'DIRECT' is access method.

Table 7-3. INQUIRE Specifiers and Their Meanings (continued)

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
SEQUENTIAL=*seq*	Character variable or array element	Sequential as possible access method	(RV) 'YES' if sequential is allowed; 'NO' if sequential is not allowed; 'UNKNOWN' if unable to determine.
DIRECT=*dir*	Character variable or array element	Direct as possible access method	(RV) 'YES' if direct is allowed; 'NO' if direct is not allowed; 'UNKNOWN' if unable to determine
FORM=*fm*†	Character variable or array element	Format specifier	(RV) 'FORMATTED' if file is connected for formatted I/O; 'UNFORMATTED' if file is connected for unformatted I/O.
FORMATTED=*fmt*†	Character variable or array element	Formatted as a possible allowed form	(RV) 'YES' if formatted is allowed; 'NO' if formatted is not allowed; 'UNKNOWN' if unable to determine.
UNFORMATTED=*unf*†	Character variable or array element	Unformatted as a possible allowed form	(RV) 'YES' if unformatted is allowed; 'NO' if unformatted is not allowed; 'UNKNOWN' if unable to determine.
BLANK=*blnk*†	Character variable or array element	Blank control specifier	(RV) 'NULL' if null blank control is in effect; 'ZERO' if zero blank control is in effect. Blank control applies only to formatted records.

† CFT allows formatted and unformatted records in the same file (non-ANSI).

7.11.3 INQUIRE STATEMENT RESTRICTIONS

A variable or array element that becomes defined or undefined as a result of its use as a identifier must not be referenced by any other identifier in the same INQUIRE statement.

Execution of an INQUIRE by file name statement causes *nmd, fn, seq, dir, fmt,* and *unf* (see table 7-3) to be assigned a value only if the value of *fin* is acceptable as a file name and if a file by that name exists. Otherwise, these identifiers become undefined. If *od* becomes defined with the value .TRUE., then *num, rcl, acc, fm, blnk,* and *nr* become defined.

Execution of an INQUIRE by unit statement causes *num, nmd, rcl, fn, acc, seq, dir, fm, fmt, unf, blnk,* and *nr* to be assigned values only if the specified unit exists and if a file is connected to the unit. Otherwise, these identifiers become undefined.

If an error condition occurs during execution of an INQUIRE statement, all of the inquiry identifiers except *ios* become undefined. *ex* and *od* always become defined unless an error condition occurs.

7.12 OPEN STATEMENT

The OPEN statement prepares an external file for use in a FORTRAN program. Depending on the status of the file when the OPEN statement is executed, the OPEN statement does one of the following:

- Connects an existing file to a unit
- Creates a file that is preconnected to a unit
- Creates a file and connects it to a unit
- Changes the characteristics of an existing connection between a file and a unit

Format:

```
 _______________
|               |
| OPEN (olist)  |
|_______________|
```

olist An external unit identifier and at most one of each of the other identifiers described in table 7-4

If a unit is connected to an existing file, execution of an OPEN statement for that unit is permitted. If the FILE= identifier is not included in the OPEN statement, the file to be connected to the unit is the same as the file to which the unit is connected.

If the file to be connected to the unit does not exist but is the same as the file to which the unit is preconnected, the specifications in the OPEN statement become a part of the connection.

If the file to be connected to the unit is not the same as the file to which the unit is connected, the effect is as if a CLOSE statement without a STATUS= identifier had been executed for the unit immediately before the execution of the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, only the BLANK= identifier can have a value that is different from the current value. Execution of the OPEN statement causes the new value of the BLANK= identifier to be in effect. The file position is unaffected.

If a file is connected to a unit, execution of an OPEN statement on that file and a different unit is not permitted.

7.13 CLOSE STATEMENT

A CLOSE statement terminates the connection of a particular file to a unit and rewinds the file.

Format:

CLOSE (*cllist*)

cllist External unit identifier and at most one of each of the other identifiers described in table 7-5

Execution of a CLOSE statement can occur in any executable program and need not occur in the same program unit as the OPEN statement that opened the file.

A disconnected file or unit can be reconnected within the same executable program either to the same file or unit, or to a different file or unit, provided the file still exists. If the file is memory resident, CLOSE deletes the file regardless of the STATUS identifier.

A file that is opened without a status specifier is deleted when it is closed. A subequent OPEN creates a new file.

The ANSI FORTRAN Standard provides an implicit CLOSE for all files upon normal program termination. CFT programs do not perform implicit CLOSE operations, and files are not automatically rewound during program termination.

The ANSI FORTRAN Standard does not allow memory resident files which are automatically deleted regardless of the STATUS identifier.

Table 7-4. OPEN Specifiers and Their Meanings

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
UNIT=*u*†	Integer	External unit specifier	(I) Unit number
IOSTAT=*ios*	Integer variable or array element	Error status specifier	(RV) 0 if no error condition exists; error message number if error condition exists.
ERR=*s*	Statement label	Statement label where control is transferred if error condition exists	(I) FORTRAN statement label
FILE=*fin*††	Character expression	File specifier	(I) Name of dataset to be connected

† UNIT= does not need to be included in the unit specification if *u* is the first item in *olist*.

†† *fin* is limited to seven characters, not counting trailing blanks.

Table 7-4. OPEN Specifiers and Their Meanings (continued)

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
STATUS=*sta*	Character expression	Disposition specifier (Default, 'UNKNOWN'	(I) 'OLD', dataset must exist and FILE= must be specified. 'NEW', dataset is created, status becomes 'OLD', FILE= must be specified. 'SCRATCH', dataset is deleted when CLOSE statement is executed or when program is terminated. Dataset must not be named. 'UNKNOWN', the status is 'SCRATCH' if no file specifier is supplied and the unit is not connected; otherwise, the status becomes 'OLD'.
ACCESS=*acc*	Character expression	Access specifier (Default, 'SEQUENTIAL')	(I) 'SEQUENTIAL' is access method; 'DIRECT' is access method.
FORM=*fm*†	Character expression	Form specifier (Default, 'UNFORMATTED' if access is direct; 'FORMATTED' if access is sequential.)	(I) 'FORMATTED', formatted I/O; 'UNFORMATTED', unformatted I/O
RECL=*rl*	Positive integer expression	Record length for direct access (omitted for sequention access	(I) For formatted I/O, number of characters per record; For unformatted I/O, 8 times the number of words
BLANK=*blnk*	Character expression	Blank specifier (Default, 'NULL')	(I) 'NULL' if numeric input blanks are ignored; 'ZERO' if all nonleading blanks are treated as zeros. This specifier permitted on datasets opened for formatted I/O only.

† CFT allows formatted and unformatted records in the same file (non-ANSI).

Table 7-5. CLOSE Specifiers and Their Meanings

Specifier	Data Type	Meaning	Input (I) or Return Value (RV)
UNIT=*u*†	Integer	External unit specifier	(I) Unit number
IOSTAT=*ios*	Integer variable or array element	Error status specifier	(RV) 0 if no error condition exists; error message number if error condition exists.
ERR=*s*	Statement label	Statement label where control is transferred if error condition exists	(I) FORTRAN statement label
STATUS=*sta*	Character expression	Disposition specifier (Default, 'KEEP' if OPEN status is 'OLD', 'NEW', or 'UNKNOWN'. Default, 'DELETE; if OPEN status is 'SCRATCH' or dataset is memory resident.)	(I) 'KEEP', the dataset continues to exist after CLOSE statement execution. Do not specify 'KEEP' for a dataset with 'SCRATCH' status on an OPEN statement. 'DELETE', the dataset does not exist after execution of the CLOSE statement.

† UNIT= does not need to be included in the unit specification if *u* is the first item in *cllist*.

7.14 NAMELIST STATEMENT (CFT EXTENSION)

The NAMELIST statement provides a format-free method of specifying I/O lists.

Format:

```
NAMELIST/group/v[,v]...[[,]/group/v[,v]...]...
```

group Group name for the following list *v*   Variable name or array name. *v* cannot be a dummy argument, cannot be referenced by a pointer, and cannot be of type CHARACTER.

The group name must be used only as a NAMELIST group name within the program unit. It can be used in place of the FORMAT statement in the following I/O statements only.

READ   (*unit,group* [,ERR=*sn*,END=*sn*])
WRITE   (*unit,group* [,ERR=*sn*])
READ   *group*
PRINT   *group*
PUNCH   *group*

Every occurrence of a group name in NAMELIST statements after the first occurrence is treated as a continuation of the first occurrence. Lists with the same group name are treated as a single group.

Variable or array names are separated by commas in the NAMELIST statement. These names can be members of more than one NAMELIST group.

The NAMELIST statement must follow all declaratives affecting the variable or array names and must precede the first use of the group name in any I/O statement.

7.14.1 NAMELIST INPUT

An input NAMELIST group record consists of one or more physical records. Column 1 is ignored, except for a possible echo flag. The first nonblank character following column 1 must contain a NAMELIST delimiter ($ or &), immediately followed by the group name and one or more blanks. The remaining portion of an input record contains as many variables desired with assigned values, separated by commas, in any order in one of the following forms.

variable=value

array=value[*,value,*]...

array(*subscripts*)=*value*[*,value,*]...

subscript An integer constant; multiple array values are assigned in storage order.

Any value can be repeated by:

```
 __________
|          |
| n*value  |
|__________|
```

n Repetition count

An input NAMELIST physical record can contain up to 160 characters. Blanks can be used for readability but must not be embedded in names or values. Names or values cannot be continued from one physical record to another. A delimiter $ or & terminates a group record. The next group record begins with the next delimiter.

An optional comment can appear between input NAMELIST group records. It can also appear within an input NAMELIST group record. A comment within the record must be preceded by a colon or semicolon. A comment, if included, is the last item in a physical record. An input NAMELIST group record can contain only comments, or can be entirely blank.

7.14.1.1 NAMELIST input variables

NAMELIST input variables can be of type integer, integer*2, real, double precision, complex or logical. If a type mismatch occurs across the equal sign, the value is converted to the declared type of the variable, following the rules of *v*=*e* in section 6, except that conversions between complex and double precision, or logical and any other type are not allowed. Octal and hexadecimal constants are considered to be Boolean. Character constants can be assigned to noncharacter variables, where they are treated as Boolean. Character constants cannot be assigned to a complex or double-precision variable.

Integer, real, and double-precision values are specified in the normal FORTRAN manner. Octal constants are specified as *dddd...d*B or as O'*ddd...d*['], where each *d* is a digit between 0 and 7. Hexadecimal constants are specified as Z'*hhh,,,h*['], where each *h* is a hexadecimal character between 0 and 9, or between A and F. Up to 22 *d*'s or 16 *h*'s can be specified. If fewer than 22 or 16 are specified, the values are right-justified in the input word.

Logical values are specified as

.T[*string*], or
.F[*string*], or
T[*string*], or
F[*string*]

string An optional string of characters that does not contain the following characters.

Replacement	(=)
Delimiter	($ or &)
Separator	(,)
Comment	(: or ;)
Array name indicator	(()

string is generally added for clarity. For example, T or .T can specify a logically true value, or, for clarity, .TRUE can be used.

Complex constants are represented as

(*real*,*imag*)

real and *imag* can be integer or floating-point constants.

7.14.1.2 NAMELIST input processing

The NAMELIST processor scans forward from the current position on the input file until it encounters a delimiter ($ or &) as the first nonblank character immediately followed by the group name.

If end-of-file or end-of-data is encountered before the group name is located, the job either aborts or branches to the END= address.

If the processor finds a NAMELIST record other than the one it is looking for, that record is skipped with an informative message to the logfile.

If the processor encounters an echo flag (E) in column 1 of any record, that record and all subsequent records processed by the current read are echoed to $OUT.

The job aborts or the ERR= branch is taken if one or more of the following conditions exists.

- The record contains a variable name that is not in a NAMELIST group.

- Punctuation is missing.

- The format of a constant is illegal.

7.14.1.3 User control subroutines

The following routines provide for control of the NAMELIST input defaults. The mode setting indicates the action to be taken.

CALL RNLSKIP(*mode*) — Determines action taken if NAMELIST sees a group name that is not the one being sought

- *mode* > 0 — Skips the record and issues a logfile message (default)
- *mode* = 0 — Skips the record
- *mode* < 0 — Aborts the job or goes to the optional ERR= branch

CALL RNLTYPE(*mode*) — Determines action taken if a type mismatch occurs across the equal sign

- *mode* ≠ 0 — Converts the constant to the type of the variable (default)
- *mode* = 0 — Aborts the job or goes to the optional ERR= branch

CALL RNLECHO(*unit*) — Specifies output unit for error message and echo lines

- *unit* < 0 — Specifies that error messages go to $OUT. Lines echoed because of an E in column 1 go to $OUT. (Default)
- *unit* ≥ 0 — Specifies that error messages go to *unit*. All input lines are echoed on *unit*, regardless of any echo flags present. (*unit*=6 or 101 imply $OUT.)

In the following user control subroutine argument lists, *char* is a character specified as 1L*x* or 1R*x*, and *mode* is a value which, if nonzero, adds the character to the set and which, if zero, removes the character from the set. No checks are made to determine the suitability or consistency of the changes.

CALL RNLFLAG(*char*,*mode*) — Adds or removes *char* from the set of characters that, if found in column 1, initiates echoing of the input lines onto $OUT. (*char* default is E.)

CALL RNLDELM(*char*,*mode*) — Adds or removes *char* from the set of characters that precede the NAMELIST group name and signal end of input. (*char* default is $ or &.)

CALL RNLSEP(*char*,*mode*) — Adds or removes *char* from the set of characters that must follow each constant to act as a separator. (*char* default is ,.)

CALL RNLREP(*char,mode*) Adds or removes *char* from the set of characters that occurs between the variable name and the value. (*char* default is =.)

CALL RNLCOMM(*char,mode*) Adds or removes *char* from the set of characters that initiates trailing comments on a line. (*char* default is : or ;.)

7.14.2 NAMELIST OUTPUT

An output NAMELIST group record is written in the following general form.

Format:

& *group name variable name* = *value*,...,
array name = *value*,...,*value*,...,&END

group name, variable name, and *array name*
Names corresponding to the names in the NAMELIST statement

For arrays, the values are listed in storage order and repeated values are listed as *n*value*.

Example:

```
&OUTPUT ARRAYX=3,7,4*5,2,&END
```

Logical values are listed as .T. or .F.

Example:

```
&OUTPUT LOGVAL=.T.,&END
```

Complex values are listed with real and imaginary portions, respectively.

Example:

```
&OUTPUT COMVAL=(2.5,3.),&END
```

An output NAMELIST group record can extend any number of lines (physical records). The first position of each line is normally blank. The first line contains the delimiter & in column 2, followed by the group name. The last line ends with the character string &END.

Default line width is 133 characters unless the unit is 102 ($PUNCH), in which case the default line width is 80 characters. NAMELIST output is readable as NAMELIST input.

7.14.2.1 User control subroutines

The following routines provide the user control of NAMELIST output format.

In the following subroutines, *char* can be any ASCII character specified by 1L*x*, or 1R*x*. No checks are made to determine if *char* is reasonable, useful, or consistent with other characters. If the default characters are changed, use of the output line as NAMELIST input might not be possible.

CALL WNLLONG(*length*) Sets the output line length to *length*. *length* must be greater than 8 and less than 161. If *length* is too short for an actual output line, the job aborts. Setting *length* to -1 restores the default line length (80 for $PUNCH; otherwise, 133).

CALL WNLDELM(*char*) Changes the character preceding the group name and END from & to *char*.

CALL WNLSEP(*char*) Changes the separator character immediately following each value from , to *char*.

CALL WNLREP(*char*) Changes the replacement operator that comes between *name* and *value*, from = to *char*.

CALL WNLFLAG(*char*) Changes the character written in column 1 of the first line from blank to *char*. Typically, *char* is used for carriage control if the output is to be listed, or for forcing echoing if the output is to be used as input for NAMELIST reads.

CALL WNLLINE(*value*) Allows each namelist variable to begin on a new line.

value = 0, no new line
value = 1, new line for each variable

The following example is a sample program, an input listing, and an output listing, showing the use of the NAMELIST statement.

PROGRAM

```
    PROGRAM EXAMPLE (TYPICAL NAMELIST I/O USAGE)
    LOGICAL ALL DONE
    REAL LENGTH
    DATA DENSITY,LENGTH,WIDTH,HEIGHT,ALLDONE /4*1.0,.FALSE./
    NAMELIST /INPUT/ LENGTH,WIDTH,HEIGHT,ALLDONE ,DENSITY
    NAMELIST /OUTPUT/ WEIGHT,LENGTH,WIDTH,HEIGHT,DENSITY
10  READ INPUT
    IF (ALLDONE) STOP
    WEIGHT = DENSITY*LENGTH*WIDTH*HEIGHT
    PRINT OUTPUT
    GO TO 10
    END
```

INPUT LISTING

```
          INPUT DATA FOR PROGRAM EXAMPLE
          NOTE THAT COMMENT CARDS MAY BE INTERSPERSED BETWEEN COMPLETE GROUPS
      $INPUT  $    USE DEFAULT VALUES
   $INPUT LENGTH = 3.0,      ; A LONG  WIDE CASE
         WIDTH = 3.   $
  &INPUT       DENSITY =   .5     &END
    &INPUT       ALLDONE = TRUE   &
/EOF
```

OUTPUT LISTING

```
&OUTPUT WEIGHT - 1., LENGTH = 1., WIDTH = 1., HEIGHT = 1., DENISTY = 1., &END
&OUTPUT WEIGHT - 9., LENGTH = 3., WIDTH = 3., HEIGHT = 1., DENISTY = 1., &END
&OUTPUT WEIGHT - 4.5, LENGTH = 3., WIDTH = 3., HEIGHT = 1., DENISTY = 0.1., &EN
```

7.15 BUFFER IN AND BUFFER OUT STATEMENTS (CFT EXTENSIONS)

Buffered I/O operations initiate a transfer of data and allow the subsequent execution sequence to proceed concurrently with the actual transfer. Either the UNIT or LENGTH utility function must be referenced to cause a delay in an execution sequence, pending completion of a buffered I/O operation. These functions can also determine certain characteristics of that operation upon its termination. The amount of data to be transferred is specified in terms of Cray computer words with no consideration given to the type or format of information contained.

Formats:

BUFFER IN $(\begin{Bmatrix} u \\ fin \end{Bmatrix},m)$ (*bloc,eloc*)

BUFFER OUT $(\begin{Bmatrix} u \\ fin \end{Bmatrix},m)$ (*bloc,eloc*)

u Unit identifier expressed as an integer or as a Hollerith expression of up to 7 characters

fin File name expressed as a character string or a character or integer variable containing a character string of up to seven characters

m Mode identifier expressed as an integer expression indicating full record processing if 0 or greater and partial record processing if less than 0

bloc Symbolic name of that variable or array element marking the beginning location of the buffered I/O transfer

eloc Symbolic name of that variable or array element marking the ending location of the buffered I/O transfer

BUFFER IN causes information to be read; BUFFER OUT causes information to be written. Execution of either statement initiates the transfer of data between the current record at unit *u* or file *fin* and the contiguous memory locations beginning with *bloc* and concluding with *eloc*. If unit *u* or file *fin* is completing a buffered I/O operation initiated earlier, a BUFFER IN or BUFFER OUT specification suspends the execution sequence until that earlier operation terminates. Upon termination, execution of the BUFFER IN or BUFFER OUT statement completes as though no delay occurred.

BUFFER IN and BUFFER OUT operations can proceed simultaneously on several units or files.

In determining the number of computer words to be transferred, consideration must be given to the data types of the symbolic names used for *bloc* and *eloc*. If *eloc* is of type double-precision or complex, the location of the second word in its 2-word form of representation marks the ending location of the data transfer.

Both *eloc* and *bloc* must be either elements of a single array (or equivalenced to an array) or must be members of the same common block. Otherwise, the results are undefined. Except for terminating a partial record, *bloc* following *eloc* in a storage sequence causes a run-time error. Neither *eloc* nor *bloc* can be character entities.

The mode identifier, *m*, controls the position of the record at unit *u* after the data transfer has taken place. Full record processing is indicated if the value of *m* is greater than or equal to 0. The record position following this mode of transfer is always between the current record (the record to or from which the transfer occurred) and the next record. For a value of *m* less than 0, partial record processing occurs.

In a BUFFER IN statement, *m* less than 0 specifies that the record be positioned ready to transfer its (*n*+1)th word if the *n*th word was the last transferred. In a BUFFER OUT statement, *m* less than 0 indicates the record is left positioned to receive additional words. A BUFFER OUT concludes a series of partial record buffered output transfers if *m* is greater than or equal to 0. A BUFFER OUT statement containing *bloc* equal to *eloc*+1 to produce a zero-word transfer also concludes the record being created.

File and record positioning for buffered I/O operations are as described for nonbuffered I/O operations. Buffered operations are allowed on all COS datasets except BUFFER OUT on COS blocked, random datasets. Buffered data transfers on COS unblocked datasets must be performed in multiples of 512 words. BUFFER IN and BUFFER OUT can be used with asynchronous SETPOS. See appendix E and the Programmer's Library Reference Manual, CRI publication SR-0113.

Example:

A BUFFER IN statement initiates the transfer of 1000 words from unit 32. Computation then proceeds on data not related to that transfer. A second BUFFER IN statement is encountered upon completion of this computation, causing a delay in the execution sequence until the last of the 1000 words is received. A transfer of another 500 words is initiated from unit 32. While these words are transferring, the execution sequence proceeds. A BUFFER OUT statement initiates the transfer of the first 1000 words to unit 22. The value of the mode identifier is 0 in all cases, indicating full record processing.

```
      PROGRAM XFR
      PARAMETER(INUNIT=32)
      DIMENSION A(1000), B(2,10,100), C(500)
      BUFFER IN(INUNIT,0) (A(1),A(1000))
      DO 10 I=1,100
 10   B(1,1,I)=B(1,1,I) + B(2,1,I)
      BUFFER IN(INUNIT,0) (C(1),C(500))
      BUFFER OUT(22,0) (A(1),A(1000))
             .
             .
             .
      END
```

7.15.1 THE UNIT FUNCTION (CFT EXTENSION)

After a BUFFER IN or BUFFER OUT statement has been executed, the normal execution sequence continues concurrently with the transfer of data. If the utility function UNIT is referenced in this execution sequence, continuation of the sequence is delayed pending completion of the transfer. After the BUFFER IN operation, a call to utility function UNIT or LENGTH is recommended before using storage locations where the information is placed.

Upon completion of the transfer, the UNIT function provides one of the following real data type values to the expression where it is referenced:

- -2.0 Partial record read operation (BUFFER IN with *m*{0) completed successfully without encountering the end of the current record

- -1.0 Operation other than a partial read completed successfully

- 0.0 End-of-file was read

- 1.0 Disk parity error occurred during reading

- 2.0 Other disk malfunction occurred during reading or writing

Example:

```
      PROGRAM TESTUNIT
      DIMENSION M(200,5)
 10   BUFFER IN (32,0) (M(1,1),M(200,5))
      IF (UNIT(32))11,13,13
 11   DO12 J=1,5
      DO12 I=1,200
 12   M(I,J)=M(I,J)*2
      BUFFER OUT (22,0) (M(1,1),M(200,5))
      IF (UNIT(22))10,13,13
 13   END
```

7.15.2 THE LENGTH FUNCTION (CFT EXTENSION)

If the utility function LENGTH is referenced while a buffered I/O operation is in progress, the execution sequence is delayed until the transfer is complete. LENGTH then returns an integer value reflecting the number of Cray computer words successfully transferred. This value is 0 if an end-of-file was read.

Example:

```
      PROGRAM PGM
      DIMENSION V(16384)
   10 BUFFER IN (32,-1) (V(1),V(16384))
      X= UNIT(32)
      K= LENGTH(32)
      IF(X)11,14,14
   11 DO 12  I=1,K,1
   12 IF(V(I).EQ.'KEY') GO TO 13
      IF(X.EQ.-2.0) GO TO 10
      STOP
   13
          ...
   14 END
```

7.16 RANDOM INPUT/OUTPUT OPERATIONS

FORTRAN-77 defines two access methods for unit/dataset connection: sequential and direct. CFT, in addition, supports random connection. Random connection is intended to meet the need for nonsequential input/output operations on a dataset with records of various lengths.

7.16.1 CREATING A DATASET FOR RANDOM ACCESS

The techniques for creating a dataset to be randomly accessed are as follows.

- The dataset can be created while the dataset is connected for sequential access.

- The WRITEDS control statement can create the dataset if the dataset is already connected for random access but no input/output to the dataset has yet occurred. The WRITEDS control statement is described in the COS Version 1 Reference Manual, publication SR-0011.

7.16.2 DATASET CONNECTION

A dataset is connected for random access through the ASSIGN control statement described in the COS Version 1 Reference Manual, publication SR-0011. CFT supports two methods of reading or writing on random access datasets: GETPOS/SETPOS and READMS/WRITMS.

7.16.2.1 Positioning while connected for random access (GETPOS/SETPOS)

The responsibility for positioning a random access dataset rests with the user. The user can position the dataset on a record boundary only. In addition, the user must maintain a log of record locations. The utility procedures provided for positioning are GETPOS and SETPOS (see appendixes B and C). SETPOS asynchronously positions a dataset to a record boundary. Similar to BUFFER IN and BUFFER OUT, SETPOS initiates a dataset position request and allows the subsequent execution sequence to proceed concurrently. An input/output request or dataset status request subsequent to a SETPOS call on the same unit waits until the SETPOS request is completed before processing. For more information on GETPOS and SETPOS see the Programmer's Library Reference Manual, CRI publication SR-0113.

Example:

In the main program below, up to 100 records containing from zero to 10 words each are written into a dataset associated with input/output unit 1. A final record of up to 201 words is added and contains length and location information for each preceding record plus a count of their number. The dataset is rewound. At a later point in the program, a subroutine is called, causing all records to be read in reverse order and all but the last record stored into 10-word vectors of a 100-vector array. Information in the last record directs this process. The subroutine then returns control to the main program.

The following are assumed preset.

- NRECS to the number of records to be processed ($1 \leq \mathrm{NRECS} \leq 100$)

- RLENGTH(i) to the number of words in the ith record written ($0 \leq \mathrm{RLENGTH}(i) \leq 10$)

- RECORD(j) to the jth word to be written in each nonempty record ($1 \leq j \leq \mathrm{RLENGTH}(i)$)

```
PROGRAM RANDOMIO
INTEGER RLENGTH(100),ADDRESS(100),RECORD(10),NRECS,LRA,RESULT
COMMON RESULT(100,10)
       .
       .
       .
    DO 20 I=1,NRECS
    ADDRESS(I)=GETPOS(1)
20  WRITE(1) (RECORD(J),J=1,RLENGTH(I))
    LRA=GETPOS(1)
    WRITE(1)NRECS,(RLENGTH(I),ADDRESS(I),I=1NRECS)
    REWIND(1)
       .
       .
       .
    CALL READIN(LRA,1)
       .
       .
       .
    END
```

7.16.3 MODIFYING A RECORD UNDER RANDOM ACCESS

Sequential read/write statements are used under random access. Formatted input/output is prohibited under random access. BUFFER IN and BUFFER OUT on COS random datasets is allowed except BUFFER OUT on COS blocked random datasets. When a record is being replaced, the length of the record being written must equal the length of the replaced record. COS blocked datasets cannot be extended while connected for random access.

7.17 RESTRICTIONS ON INPUT/OUTPUT STATEMENTS

A function must not be referenced in an I/O statement if it causes an I/O statement to be executed.

An I/O statement must not reference a unit or file not having all the properties required for its execution.

7.18 I/O ERROR RECOVERY

If an irrecoverable error occurs during the execution of an I/O statement, the operating system aborts the current job step. The current job step is aborted even if an error identifier (ERR=*sn*) appears in the I/O statement's control information list. Generally, error conditions detected by code in $FTLIB are recoverable and return control to the statement indicated by the error identifier; error conditions detected by the operating system are irrecoverable and abort the current job step.

The ANSI FORTRAN Standard does not distinguish between recoverable and irrecoverable errors.

7.19 CHANGING MAXIMUM LENGTH FOR I/O LISTS AND FORMAT SPECIFICATIONS

The maximum length for I/O lists and format specifications in CFT is normally 152 characters. The length of the $IOLIB read and write buffers can be changed in either of two ways: by using SEGLDR directives, or by regenerating the version of $IOLIB used with the CFT compiler.

7.19.1 CHANGING I/O BUFFER LENGTHS USING SEGLDR DIRECTIVES

The SEGLDR directives SET and COMMONS can be used to change the values of the buffer lengths and common block lengths in $IOLIB. The SET directives should specify the desired buffer length, and the common directives should specify a value 9 greater than the desired buffer length. For example, to change both the read and write buffer lengths to 256 words, the following sequence of SEGLDR directives can be used:

```
    SET=$RBUFLN:256
    COMMONS=$RFDCOM:265
    SET=$WBUFLN:256
    COMMONS=$WFDCOM:265
```

The SEGLDR directives affect the buffer lengths only for the job in which they are used.

For more information about the SET and COMMONS SEGLDR directives, see the Segment Loader (SEGLDR) Reference Manual, CRI publication SM-0066.

7.19.2 CHANGING I/O BUFFER LENGTHS BY REGENERATING $IOLIB

The values of the buffer lengths and common block lengths in $IOLIB can be changed by regenerating $IOLIB. This involves changing the values of $RBUFLN and $WBUFLN in deck RWFDCD in program library IOLIBPL. Changing these values and regenerating $IOLIB automatically resizes the common blocks $RFDCOM and $WFDCOM.

Changing the buffer lengths in this manner affects all jobs using the regenerated library.

8. INPUT/OUTPUT FORMATTING

CFT allows both formatted and unformatted input/output (I/O). Unformatted I/O is considerably faster than formatted I/O. Your decision to use formatted or unformatted I/O can have a serious impact on program performance; using unformatted I/O in CFT programs with large amounts of I/O can be the single most important factor in program optimization.

CFT provides two methods of formatting program input and output. *List-directed I/O* is easy to use, but allows relatively limited control over the format of input and output records. *Formatted I/O* is more complex, and allows greater control over the format of input and output records.

8.1 UNFORMATTED I/O

Unformatted I/O is accomplished by using a data transfer statement with no format specified. For example, the statement

```
    WRITE(9)X
```

writes one record containing the value of X to the file connected to unit 9.

With unformatted I/O, all items in the input or output list are written to or read from a single record. The statement

```
    WRITE(9)X,Y,Z
```

writes one record containing the values of X, Y, and Z to the file connected to unit 9.

List items are written and read based on their type. Integers, real numbers, and logical values occupy one word of memory; complex and double-precision numbers occupy two words of memory; and character strings are stored eight characters per memory word, left-justified. Unformatted READ statements read the appropriate number of memory words for the type of each variable in the input list.

For example, in the following program,

```
    LOGICAL TEST
    CHARACTER*24 STRING
    INTEGER Z
     ...
    READ(9)TEST,STRING,Z
```

one word from the file connected to unit 9 would be assigned to the variable TEST, the next three words would be assigned to STRING, and the next word would be assigned to Z.

Implied-DO lists can be used in unformatted I/O statements. For example, the statement

```
    WRITE(9)(VECTOR(I),I=1,55)
```

writes one record, containing the values of VECTOR(1) through VECTOR(55), to the file connected to unit 9.

8.2 LIST-DIRECTED I/O

List-directed I/O allows data editing to be performed according to the type of the list item instead of by a format identifier. List-directed records consist of values and value separators. Each value is either a constant, a null value, or one of the following forms.

Format:

```
     ______
    |      |
    | r*c  |
    | r*   |
    |______|
```

r Unsigned, nonzero, integer constant

The *r**c form is equivalent to *r* successive appearances of the constant *c*. The *r** form is equivalent to *r* successive null values. Neither of these forms can contain embedded blanks, except where permitted within the constant *c*.

Value separators can have one of the following forms.

- A comma optionally preceded and followed by one or more contiguous blanks

- A slash optionally preceded and followed by one or more contiguous blanks

- One or more contiguous blanks between two constants or following the last constant

8.2.1 LIST-DIRECTED INPUT

The form of a list-directed input value must be acceptable for the type of the input list item. Blanks cannot be used as zeros. Embedded blanks are permitted only in complex constants and character constants.

Type real or double-precision list items must be numeric and suitable for F editing.

A type complex list item consists of an ordered pair of numeric fields separated by a comma and enclosed in parentheses. The first numeric field is the real portion of the complex constant; the second numeric field is the imaginary portion. An end-of-record can occur between the real portion and the comma or between the comma and the imaginary portion. Each numeric field can be preceded or followed by blanks.

A list item of type logical must not include either slashes or commas among the optional characters permitted for L editing.

A type character list item has an input form with a nonempty string of characters enclosed in apostrophes. Each apostrophe in a character constant must be represented by two consecutive apostrophes without a blank or end-of-record. Character constants can be continued from the end of one record to the beginning of the next record. The end of the record does not cause a blank or any other character to become part of the constant. The constant can be continued on to as many records as needed. A blank, comma, and slash can appear in character constants.

For example, if *len* is the list item length, *w* is the character constant length and *len* is less than or equal to *w*, the leftmost *len* characters of the constant are transmitted to the list item. If *len* is greater than *w*, the constant is transmitted to the leftmost *w* characters of the list item and the remaining *len*-*w* characters of the list item are filled with blanks. The effect is as if the constant were assigned to the list item in a character assignment statement.

A null value has no characters before or between value separators. A null value has no effect on the definition status of the corresponding input list item. A single null value can represent an entire complex constant but it cannot be used as either the imaginary or the real portion alone. The end of a record following any other separator, with or without separating blanks, does not specify a null value.

A slash encountered as a value separator during execution of a list-directed input statement terminates execution of that input statement after the assignment of the previous value. If additional items are present in the input list, the effect is as if null values had been supplied for them.

All blanks in a list-directed input record are considered to be part of some value separator except for the following.

- Embedded blanks surrounding the real or imaginary portion of a complex constant

- Leading blanks in the first record read, unless immediately followed by a slash or comma

8.2.2 LIST-DIRECTED OUTPUT

The form of the values produced is the same as that required for input, except as noted otherwise. The values are separated by one of the following.

- One or more blanks

- A comma optionally preceded and followed by one or more blanks

New records begin as necessary but, except for complex and character constants, the end of a record does not occur within a constant and blanks do not appear within a constant.

Logical output constants are T for the value true and F for the value false.

Integer output constants are produced with the effect of an I*w* edit descriptor, for some value of *w*.

Real and double-precision constants are produced with the effect of either an F edit descriptor or an E edit descriptor, depending on the magnitude *x* of the value and a range $10{*}{*}^{-2466}\underline{\{}x\{10{*}{*}^{2466}$. If the magnitude *x* is within this range, the constant is produced with 0PF*w*.*d*; otherwise, 1PE*w*.*d*E*e* is used. Reasonable values of *w*, *d*, and *e* are used for each of the cases involved.

Complex constants are enclosed in parentheses, with a comma separating the real and imaginary portions. If two or more successive values in an output record have identical values, a repeated constant of the form *r***c* is produced instead of the sequence of identical values.

Character constants are not delimited by apostrophes and are not preceded or followed by a value separator. Each internal apostrophe in a character constant is represented externally by one apostrophe. Character constants have a blank character inserted by the processor for carriage control at the beginning of any record that begins with the continuation of a character constant from the preceding record.

Slashes as value separators and null values are not produced by list-directed formatting.

Each output record begins with a blank character for carriage control when the record is printed.

A *format specification* provides explicit editing information to direct the editing of data between its internal representation and the corresponding character strings required. Format specifications can be given in FORMAT statements, or as values of character arrays, character variables, or other character expressions.

A format identifier that is a statement label must be the label of a FORMAT statement in the same program unit. The format specification contained in that FORMAT statement is applied when the formatted I/O or assignment statement is executed.

A format specification begins with a left parenthesis and ends with a right parenthesis. A complete format specification can contain another complete format specification. Nesting of this type can be carried to nine levels. Character data following the right parenthesis of a complete format specification is ignored only when the specification is contained in an array.

8.3 FORMAT STATEMENTS

The FORMAT statement is a nonexecutable statement that specifies the formatting of data to be read or written with formatted I/O.

Format:

label FORMAT *flist*

label Statement label (required)

flist List of items, each having one of the following forms:

ned
[*r*]*ed*
[*r*](*flist*)

ned Nonrepeatable edit descriptor

ed Repeatable edit descriptor

r Nonzero, unsigned integer constant called a repeat specification; if not specified, a value of 1 is assumed.

Commas can separate list items in *flist* but are required only under the following conditions.

- Between two adjacent digits where each belongs to different list items
- Between two adjacent apostrophe or quotation mark delimiters of separate edit descriptors
- After a D, E, or G specification that precedes an E specification

---

The ANSI FORTRAN Standard does not provide for the optional use of commas except before and after the slash or the colon edit descriptor or between a P edit descriptor and an immediately following F, E, D, or G edit descriptor.

Examples:

```
    1999 FORMAT ('F',5X,6F6.2)

    1234 FORMAT ('ABC123',2X,"=",D15.5,2X,I6)
```

8.4 EDIT DESCRIPTORS

Edit descriptors specify the form of a record and direct the editing between characters in a record and their corresponding internal representation.

An edit descriptor is either a repeatable edit descriptor, *ed*, or a nonrepeatable edit descriptor, *ned*. Table 8-1 summarizes the format and function of the repeatable edit descriptors; table 8-2 summarizes the format and function of the nonrepeatable edit descriptors.

Table 8-1. Repeatable Edit Descriptors

Data Type	Edit Descriptor	Descriptor
Alphanumeric	A	Character with data-dependent length
	Aw	Character with specified length
Boolean	Ow	Octal integer
	Ow.m	Octal integer with leading zeros and minimum number of digits
	Rw	Boolean conversion
	Zw	Hexadecimal integer
	Zw.m	Hexadecimal integer with leading zeros and minimum number of digits
Logical	Lw	Logical
Numeric	Dw.d	Double-precision floating-point with exponent
	Ew.d	Single-precision floating-point with exponent
	Ew.dEe	Single-precision floating-point with specified exponent length
	Fw.d	Single-precision floating-point without exponent

w Field width in number of character positions in the external record. This width includes leading blanks, + or - signs, decimal point, and exponent.

d Number of digits to the right of the decimal point within the field. On output all numbers are rounded.

e Number of digits in the exponent; must not be greater than 6.

m Minimum number of digits to be output.

Table 8-1. Repeatable Edit Descriptors (continued)

Data Type	Edit Descriptor	Descriptor
Numeric (continued)	Gw.d	Single-precision floating-point with or without exponent
	Gw.dEe	Single-precision floating-point with or without exponent of specified length
	Iw	Decimal integer
	Iw.m	Decimal integer with minimum number of digits

w Field width in number of character positions in the external record. This width includes leading blanks, + or - signs, decimal point, and exponent.

d Number of digits to the right of the decimal point within the field. On output all numbers are rounded.

e Number of digits in the exponent; must not be greater than 6.

m Minimum number of digits to be output.

Table 8-2. Nonrepeatable Edit Descriptors

Data Type	Edit Descriptor	Descriptor
Blank Control	BN	Blanks ignored
	BZ	Blanks treated as zeros
Character Output	"	Output character string
	'	Output character string
Format Control	:	Terminate format control

k Integer constant scale factor

n Positive nonzero integer

Table 8-2. Nonrepeatable Edit Descriptors (continued)

Data Type	Edit Descriptor	Descriptor
Hollerith Data	nH	Output Hollerith string
Position Control	Tn	Position forward or backward
	TRn	Position forward
	TLn	Position backward
	nX	Position forward
	$	Suppress carriage control

k Integer constant scale factor

n Positive nonzero integer

Table 8-3 describes the correct usage of the CFT edit descriptors with data types. An * indicates legal usage for input and output. A + indicates legal usage for output. A - indicates illegal usage.

Table 8-3. Edit Descriptors with Data Types

Data Types	Edit Descriptors									
	I	F	E	D	G	L	A	O	Z	R
Character	-	-	-	-	-	-	*	-	-	-
Complex	-	*	*	*	*	-	*	*	*	*
Double-precision	-	*	*	*	*	-	-	+	-	-
Integer	*	-	-	-	-	-	*	*	*	*
Logical	-	-	-	-	-	*	*	*	*	-
Real	-	*	*	*	*	-	*	*	*	*

Format restrictions for integer, logical, and real variables can be lifted using SEGLDR and its EQUIV directive. To change the limitations for read and write operations, specify EQUIV=$RNOCHK($RCHK) or EQUIV=$WNOCHK($WCHK), respectively. Both of these EQUIV statements must be specified if changes are desired. Table 8-4 describes the edit descriptors and data types when SEGLDR and the EQUIV directive is used. An * indicates legal usage for input and output. A - indicates illegal usage.

Table 8-4. Edit Descriptors and Data Types When SEGLDR and the EQUIV Directive Are Used

	Edit Descriptors									
Data Types	I	F	E	D	G	L	A	O	Z	R
Integer	*	*	*	_	*	*	*	*	*	*
Logical	*	*	*	_	*	*	*	*	*	*
Real	*	*	*	*	*	*	*	*	*	*

8.4.1 INTERACTION BETWEEN I/O LISTS AND FORMAT SPECIFICATIONS

The beginning of execution of a formatted I/O statement initiates format control. Each action of format control depends on information from the next edit descriptor provided by the format specification, and the next item in the I/O list, if one exists.

If a statement has an I/O list, at least one repeatable edit descriptor must exist in the format specification.

An empty format specification of the form ( ) can be used unless contained within another format specification. An empty format specification causes one input or internal record to be skipped or one output or internal record containing no characters to be written. No I/O list items can correspond to an empty format specification.

Except for repeated edit descriptors and embedded format specifications, a format specification is interpreted from left to right.

An embedded format specification or edit descriptor preceded by an *r* is processed as a list of *r* format specifications or edit descriptors. An omitted repeat specification is treated the same as a repeat specification with a value of 1.

Each repeatable edit descriptor interpreted in a format specification corresponds to one item specified by the I/O list, except that an item of type complex requires the interpretation of two F, E, D, G, A, or R edit descriptors. An I/O list contains no items corresponding to nonrepeatable edit descriptors.

When format control encounters a repeatable edit descriptor, it determines whether the I/O list has specified a corresponding item. If it has, format control transmits appropriately edited information between the item and the record, then proceeds. If no corresponding item exists, format control terminates.

Format control also terminates if the rightmost parenthesis of a complete format specification is encountered and no additional I/O list items are specified. If another list item is specified, the file is positioned to the next record and format control reverts to the beginning of that format specification terminated by the next-to-last right parenthesis. If there is none, format control reverts to the first left parenthesis of the complete format specification. If reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is immediately preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (see P editing) or on S, SP, SS, BN, or BZ.

Examples:

In the following examples, the ↑ indicates the reversion point if list items remain when format control encounters the closing parenthesis.

```
1  FORMAT(10F10.3,1PE20.6)
         ↑

2  FORMAT(10F10.3,(1PE20.6))
                  ↑

3  FORMAT(I10,3(I5,2(I5,I7),3(L1,L2),I7))
              ↑

4  FORMAT(I5,2(I4,I6),3(I1,I2))
                      ↑
```

8.4.2 POSITIONING BY FORMAT CONTROL

If a T or X edit descriptor is the first edit descriptor encountered after format control is initiated, the action of the descriptor causes the next record to become the current record.

After the processing of each repeatable edit descriptor or an H, apostrophe, or quotation mark edit descriptor, the file is positioned after the last character read or written in the current record.

After a T, TL, TR, X, slash, or colon edit descriptor is processed, the file is positioned as separately described for each.

If format control reverts, the file is positioned in the same manner as when a slash edit descriptor is processed.

After a read operation, any unprocessed characters of the record read are skipped.

When format control terminates, the file is positioned after the current record.

8.4.3 APOSTROPHE AND QUOTATION MARK EDITING

An apostrophe or quotation mark edit descriptor has the form of a character constant and causes characters to be written from the delimited characters (including blanks) of the edit descriptor itself. These edit descriptors apply only to output. The width of the field is the number of characters contained between (but not including) the delimiting quotation marks or apostrophes. Within the field, two adjacent apostrophes or quotation marks are counted as one and not as members of a delimiting apostrophe or quotation mark character pair, respectively.

The ANSI FORTRAN Standard does not provide for quotation mark editing.

Example:

Execution of -

```
      WRITE(6,13)
   13 FORMAT(' ISN''T "*" BETTER'," THAN ""H""", 'IS')
```

results in the printing of -

```
   ISN'T "*" BETTER THAN "H" IS
```

8.4.4 H EDITING

The *n*H edit descriptor causes character information to be written from the *n* characters (including blanks) following the H of the edit descriptor. An H edit descriptor can be used only for output.

Examples:

```
       PRINT 22
    22 FORMAT(27H ABCDEFGHIJKLMNOPQRSTUVWXYZ,10H1234567890)
       WRITE(41,16)
    16 FORMAT(' LABEL',5H UNIT,' 41')
```

8.4.5 POSITIONAL EDITING (T, TL, TR, AND X)

The T, TL, TR, and X descriptors specify the position where the next character will be transmitted to or from the record.

An X edit descriptor specifies a position relative to the current position.

T edit'descriptors can specify a character position in either direction from the current position. This allows portions of a record to be read more than once, possibly with different editing.

T or X edit descriptors can replace a character that is already in the record. During transmission to the record, undefined positions are filled with blanks. The result is as if the entire record were initially filled with blank characters. On output, an X descriptor that specifies a move to position *c* causes the length of the record to be at least *c*-1 characters. T edit descriptors by themselves do not affect the length of an output record. Positions beyond the last character of the record can be specified if no characters are to be transmitted from such positions.

8.4.5.1 T, TL, and TR editing

The T*c* edit descriptor indicates the transmission of the next character to or from a record is to occur at the *c*th character position.

The TL*c* edit descriptor indicates the transmission of the next character is to occur at the character position *c* characters backward from the current position. If the current position is less than or equal to position *c*, the transmission of the next character occurs at position 1 of the current record.

The TR*c* edit descriptor indicates the transmission of the next character is to occur at the character position *c* characters forward from the current position.

8.4.5.2 X editing

During transmission from a record, the *n*X edit descriptor causes the skipping of *n* character positions following and including the current character position. During transmission to a record, blank characters are placed into *n* character positions beginning with the current character position. In both cases, the record becomes positioned to the first character following the last character processed.

Example:

Execution of -

```
      PRINT 12345
12345 FORMAT(1X,'ONE',16X,'FIVE',T6,'TWO',7X,4HFOUR,T10,'T','HR','E',1HE)
```

Results in the printing of -

```
Position:
1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

Content:
   O  N  E     T  W  O     T  H  R  E  E     F  O  U  R     F  I  V  E
```

The first output character controls vertical spacing. Although this character is not printed, it must be included in the edit descriptor character count. For example, (T6) in the above example represents the position of the fifth character to be printed.

8.4.6 SLASH EDITING

The slash edit descriptor indicates the end of a record. During transmission from a file, the remaining portion of any current record is skipped and the file is positioned at the beginning of the next record. If no current record exists, the file is positioned after the next record. During transmission to a file, an empty record is written as the last record of the file. Thus, an empty record can be written on output and an entire record can be skipped on input.

Slash edit descriptor processing of adjacent records can be specified by the appearance of as many consecutive slashes (optionally separated by commas) or by preceding a single slash with a *n* value equal to the number of records to be processed.

The ANSI FORTRAN Standard does not provide for a repeat count for slash editing.

Examples:

```
       PRINT 39

    39   FORMAT('1LINE 1',/,' LINE 2'/' LINE 3'///7H LINE 6)

         READ(99,42) RECORD3

    42   FORMAT(2/,...)
```

8.4.7 COLON EDITING

The colon prevents the printing of some or all text information by a format that is used with a varying number of list items. When encountered in a format specification, a colon edit descriptor terminates the formatted transfer of data if no I/O list items remain to be processed. If unprocessed I/O list items remain, the colon edit descriptor has no effect on format control. Termination of format control by a colon edit descriptor causes the record being processed to become the preceding record.

Example:

Execution of

```
          PRINT 10,X
       10 FORMAT(' X= 'F10.5,' Y= 'F10.5)
```

results in the printing of

```
       X= 1234.56789 Y=.
```

Whereas execution of

```
       PRINT 20,X
      20 FORMAT(' X= 'F10.5,:' Y= 'F10.5)
```

results in the printing of

```
       X= 1234.56789.
```

8.4.8 DOLLAR SIGN EDITING (CFT EXTENSION)

The dollar sign character ($) in a format specification modifies the carriage control specified by the first character of the record. In an output statement, the $ descriptor suppresses the carriage return/line feed. In an input statement, the $ descriptor is ignored. The $ descriptor is intended primarily for interactive I/O; it leaves the terminal print position at the end of the text (instead of returning it to the left margin), so a typed response follows the output on the same line.

Example:

Execution of

```
      WRITE (6,100)
    100  FORMAT('WHAT IS YOUR NAME?',$)
      READ (5,105)
    105  FORMAT (4A8)
```

Results in the printing of

```
      WHAT IS YOUR NAME?
```

The response (in this example, HARRY) can go on the same line

```
      WHAT IS YOUR NAME? HARRY
```

8.4.9 P EDITING

A scale factor is specified by a P edit descriptor of the form *k*P, where *k* is an optionally signed integer constant called the *scale factor*. *k*P represents 10^k as a multiplier.

The scale factor is 0 at the beginning of each I/O statement. It applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered and established. Note that reversion of format control does not affect the established scale factor.

The scale factor, *k*, affects editing in the following manner.

- With F, E, D, and G input editing (provided that no exponent exists in the field) and with F output editing, the scale factor causes the externally represented number to correspond to the internally represented number multiplied by 10 to the *k*th power.

- On input with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.

- On output with E and D editing, the basic real constant part of the quantity to be produced is multiplied by the *k*th power of 10 and the exponent is reduced by *k*.

- On output with G editing, the effect of the scale factor is suspended unless the magnitude of the data to be edited requires the use of E editing. In this case, the scale factor has the same effect as with E output editing.

Examples:

field		9876.54	98.7654E2	9876.54	987.654	.864786D-4	86.4786E2	
FORMAT statement	FORMAT (	2PF8.3,	-2PE9.4,	F9.4,	0PG9.4,	D9.4,	-2PE9.4	)
Internal representation		98.7654	9876.54	987654.	987.654	.0000864786	8647.86	
Internal representation		9.87654	9876.54	9876.54	987.654	864.786	8647.86	
FORMAT statement	FORMAT (	2PF12.2,	-2PE12.4,	F12.4,	1PG12.2,	D12.4,	-2PE12.4	)
Output field		987.65	.0099E+06	98.7654.	9.88E+02	8.6479D+02	.0086E+06	

1312-01

The scale factor *k* control decimal normalization. If -*d*{*k*{0, there are |*k*| leading zeros and *d*-|*k*| significant digits after the decimal point. If 0{*k*{(*d*+2), there are *k* significant digits to the left of the decimal point and *d*-*k*+1 significant digits to the right of the decimal point. Other values of *k* are not permitted.

8.4.10 NUMERIC EDITING (BN, BZ, S, SP, SS, I, F, E, D, AND G)

Numeric editing specifies I/O editing of integer, real, double-precision, and complex data. The following general rules apply.

- On input, leading blanks are not significant. Plus signs can be omitted. A field of all blanks has the value 0.

- On input with F, E, D, and G editing, a decimal point appearing in the input field overrides that portion of an edit descriptor specifying the decimal point location. The input field can have more digits than are used in approximating the value of the data. The excess digits are used to round to the approximation but are otherwise discarded.

- On output, a positive or zero internal value in the field is prefixed with blank characters except as described below for S, SP, and SS editing. A negative internal value in the field is prefixed with blank characters followed by a minus sign.

- On output, the representation is right-justified in the field.

  If the number of characters produced by the editing is smaller than the field width, leading blanks are inserted in the field.

- On output, if the number of characters exceeds the field width, the entire field is filled with asterisks.

8.4.10.1 BN and BZ editing

The BN and BZ edit descriptors specify the interpretation of blanks other than leading blanks. BN and BZ affect input fields only.

The BN edit descriptor causes blanks to be ignored. Ignoring blanks has the effect of removing blanks, right-justifying the remaining portion of the field, and replacing the removed blanks as leading blanks. A field of all blanks has the value 0.

The BZ edit descriptor causes all blank characters to be treated as zeros. The initial interpretation of blanks in numeric input fields depends on the value of the BLANK= identifier when the unit was opened. NULL (BN) is the default.

8.4.10.2 S, SP, and SS editing

The S, SP, and SS edit descriptors control plus signs in numeric output fields. Normally, the compiler suppresses plus signs. The SP edit descriptor causes plus signs to be produced on numeric output fields until either an S or an SS edit descriptor is encountered. The SS edit descriptor specifies suppression of plus signs; the S edit descriptor restores the normal compiler option, which, in this case, is also the suppression of plus signs.

8.4.10.3 Integer editing

The I*w* and I*w*.*m* edit descriptors indicate that the field to be edited occupies *w* positions. The specified I/O list item must be of type integer. On input, the specified list item becomes defined with an integer datum. On output, the specified list item must be defined with an integer datum.

In the input field, the character string must be in the form of an optionally signed integer constant. Leading blanks in the input field are ignored. The Iw.*m* edit descriptor is treated identically to the Iw edit descriptor.

The output field for the Iw edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. If the value (plus the possible minus sign) exceeds w digits, the field is filled with asterisks.

If the Iw.*m* edit descriptor is used on output, the unsigned, integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of w. If *m* is 0 and the value of the internal datum is 0, the output field consists of only blank characters.

Example:

Execution of -

```
      READ 20,I,J,K
  20  FORMAT(I2,I5,I3)
```

with an input line of -

```
  15bb-10bb
```

followed by -

```
  PRINT 10,I,J,K
10  FORMAT(I5,I3,I4)
```

yields -

```
  bbb15-10bbb0.
```

Where b indicates a blank character.

8.4.10.4 F editing

The Fw.*d* edit descriptor indicates that the field occupies w positions, the fractional part of which consists of *d* digits.

The input field consists of an optional sign followed by a string of digits optionally containing a decimal point. This basic form can be followed by an exponent of 10 having one of the following forms.

- Signed integer constant
- E followed by an optionally signed integer constant
- D followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

The output field consists of blanks, if necessary, followed by a minus sign if the internal value is negative, followed by a string of digits that contains a decimal point. This string of digits represents the magnitude of the internal value. This representation is modified by the established scale factor and is rounded to *d* fractional digits. If the output field value is less than 1, a single 0 is written immediately to the left of the decimal point, space permitting. If the output field value is 0 and *d* is 0, a single 0 is written. In no other cases are leading zeros written. If the value is too large to print in the specified field, the field is filled with asterisks. If the value is an out-of-range floating-point value, a single R is printed, right-justified in the field.

Examples:

Input Field Positions										F Edit	Internal
1	2	3	4	5	6	7	8	9	10	Descriptor	Representation
1	7	7	6	.	1	9	7	6		F9.4	1776.1976
-	1	7	7	6	.	1	9	7	6	F10.4	-1776.1976
-	1	7	7	6	.	1	9	7	6	F9.4	-1776.197
1	9	7	7							F4.0	1977.
1	9	7	7							F4.4	.1977
1	9	7	7							F2.0	19.
-	1	4	9	2	E	-	3			F8.0	-1.492
6	.	0	2	3	D	2	3			F8.3	602300000000000000000000.

The ANSI FORTRAN Standard does not specify output editing for values too large to be printed in the specified field.

Internal	F Edit	Output Field Positions									
Representation	Descriptor	1	2	3	4	5	6	7	8	9	10
3.1415926	F10.5				3	.	1	4	1	5	9
-3.1415926	F7.4	-	3	.	1	4	1	6			
747	F4.0	7	4	7	.						
0	F8.6	0	.	0	0	0	0	0	0		
0	F8.5		0	.	0	0	0	0	0		
0	F7.6	.	0	0	0	0	0	0			

8.4.10.5 E editing

The E*w*.*d* and E*w*.*d*E*e* edit descriptors indicate that the external field occupies *w* positions. The fractional portion consists of *d* digits unless the scale factor is greater than 1. The exponent portion consists of *e* digits. *e* has no effect on input. If the value is an out-of-range floating-point value, a single R is printed, right-justified in the field.

The format of the input field is the same as for F editing.

The format of the output field for a scale factor of 0 is

$$[-][0].x_1\ x_2\ \ldots\ x_{d-1}\ x_d\ exp$$

where $x_1 x_2 \ldots x_d$
are the *d* most significant digits of the rounded data, and

exp is a decimal exponent of one of the following forms.

Edit Descriptor	Absolute Value of Exponent	Output Form of Exponent
E*w*.*d*	*exp* = 0	E+00
E*w*.*d*	$0<\|exp\|\le 99$	$E\pm y_1y_2$
E*w*.*d*	$100\le\|exp\|\le 999$	$\pm y_1y_2y_3$
E*w*.*d*	$1000\le\|exp\|\le 2466$	$\pm y_1y_2y_3y_4$
E*w*.*d*E*e*	$exp\|\le(10**e)-1$†	$E\pm y_1y_2y_3\cdots y_e$

† If e is greater than the number of digits necessary to express *exp*, leading zeros are inserted.

An $|exp|\ge 1000$ value causes the entire field to be shifted left one position to provide for y_4. If space has not been provided, the entire field is replaced with asterisks.

The value of *w* must be greater than *d*+5 for output.

Examples:

Input Field Positions												E Edit	Internal
1	2	3	4	5	6	7	8	9	10	11	12	Descriptor	Representation
+	1	0	4	8	5	7	5	.	7	5		E11.2	1048575.75
-	1	0	4	8	5	7	5	.	7	5		E11.0	-1048575.75
									3	8		E11.11	.00000000038
		1	.	5	9	2	E	3				E12.3	1592.
6	5	5	3	6	E	-	5					E8.3	.00065536
6	5	5	3	6	.	E	-	5				E9.3	.65536
-	3	2	.	7	6	8	D	0	4			E10.3	-327680.

Internal	E Edit	Output Field Positions										
Representation	Descriptor	1	2	3	4	5	6	7	8	9	10	11
365.26	E10.2			0	.	3	7	E	+	0	3	
-365.26	E11.5	-	.	3	6	5	2	6	E	+	0	3
.000000099	E11.3			0	.	9	9	0	E	-	0	7
100.	E11.2E1						.	1	0	E	+	3
100.	E11.2E4			.	1	0	E	+	0	0	0	3

8.4.10.6 D (double-precision) editing

D editing is identical to E editing.

8.4.10.7 G editing

The G*w*.*d* and G*w*.*d*E*e* edit descriptors indicate that the field occupies *w* positions with *d* significant digits, and contains an exponent of *e* digits.

G input editing is the same as F input editing.

Representation in the output field depends on the magnitude of the data being edited. If N is the magnitude of the internal data, its value determines the editing as follows.

Magnitude of Data	Equivalent Edit Descriptors
$0.1 \leq N < 1$	F(*w*-4).*d*,4X
$1 \leq N < 10$	F(*w*-4).(*d*-1),4X
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(*w*-4).1,4X
$10^{d-1} \leq N < 10^{d}$	F(*w*-4).0,4X
$N < 0.1$ or $N \geq 10^{d}$	*k*P,E*w*.*d*

where *k* is the scale factor in effect. The scale factor is effective only if the magnitude of the data exceeds the range for effective F editing.

The value of *w* must be greater than *d*+5 for output.

Examples:

Input Field Positions												G Edit	Internal
1	2	3	4	5	6	7	8	9	10	11	12	Descriptor	Representation
6	2	9										G5.1	6290.
-	.	6	2	9	0	0	0	0				G10.2	-.629
+	8	7	8	.	4	9	2	1				G9.4	878.4921
4	7	2	1	.	0	E	-	2				G12.1	47.21
7	2	D	1	0								G5.0	720000000000.

Internal	G Edit	Output Field Positions											
Representation	Descriptor	1	2	3	4	5	6	7	8	9	10	11	12
-324.876	G12.6	-	3	2	4	.	8	7	6				
.487295343397	G10.5	.	4	8	7	3	0						
-72.59	G10.3		-	7	2	.	6						
.000000000019	G12.2						.	1	9	E	-	1	0
.000000000019	G9.1				.	2	E	-	1	0			
10000.	G12.2						.	1	0	E	+	0	5
10000.01	G12.2						.	1	0	E	+	0	5
10000.	G12.2E1							.	1	0	E	+	5
10000.	G12.2E4				.	1	0	E	+	0	0	0	5

8.4.11 COMPLEX EDITING

Complex data consists of a pair of separate real data. Data editing must be specified by two successively interpreted A, D, E, F, G, O, R, or Z edit descriptors. The first of the edit descriptors specifies editing for the real part; the second for the imaginary part. The two edit descriptors can differ. Nonrepeatable edit descriptors can appear between two successive A, D, E, F, G, O, R, or Z edit descriptors.

8.4.12 O (OCTAL) EDITING (CFT EXTENSION)

The O*w* edit descriptor indicates the processing of an input list item of type integer, real, complex, Boolean, or logical and a field width of *w* positions. A double-precision list item can be used with an O*w* descriptor for output only.

On input, the field contains a string of from 0 to 22 octal digits or blanks, representing a binary value to be stored into the list item. This value is right-justified in the list item if fewer than 22 octal digits are contained in the field. Unspecified bit positions are cleared to 0. A blank field is considered to be a field containing all zeros. If the first nonblank character in the field is a minus, the ones complement of the value is stored.

On output, the internal representation of the list item is converted to octal and the rightmost *w* octal digits are right-justified in the field.

If the list item is not of type double-precision and the field is larger than 22 positions, the output contains leading blank characters. If the list item is of type double-precision and *w* is greater than 45, the output contains leading blank characters. If *w* is greater than 22, a blank character occupies position (*w*-22) in the output field. This character indicates the beginning of the double-precision portion. To completely output a double-precision value, the value of *w* must be at least 45.

8.4.13 Z (HEXADECIMAL) EDITING (CFT EXTENSION)

The Z*w* edit descriptor indicates processing of a list item of type integer, real, complex, Boolean, or logical and a field width of *w* positions.

On input, the field contains a string of from 0 to 16 hexadecimal characters representing a zero or positive integral value (in the base-16 number system) to be stored into the list item. This value is right-justified in the list item if fewer than 16 hexadecimal characters are contained in the field; leading zeros are assumed. A blank field is assumed to be a field of all zeros. If the first nonblank character in the field is a minus, the ones complement of the value is stored.

On output, the internal representation of the list item is converted to a zero or positive hexadecimal value and the rightmost *w* digits are right-justified in the field. If the field is larger than 16 positions, leading blank characters are output.

8.4.14 L (LOGICAL) EDITING

The L*w* edit descriptor indicates processing of a logical list item and an input or output field width of *w* positions. The specified I/O list item must be of type logical. On input, the list item becomes defined with logical data. On output, the list item must be defined with logical data.

The input field consists of a T for true or an F for false, optionally followed by additional characters. The field can contain a leading period or leading blanks.

The output field consists of *w*-1 blanks followed by a T or F, depending on the value of the internal data.

Examples:

Input Field Positions												L Edit	Internal
1	2	3	4	5	6	7	8	9	10	11	12	Descriptor	Representation
T												L1	(true)
.	T	R	U	E								L4	(true)
		F										L3	(false)
		.	F	A	L	S	E					L12	(false)
				T	1	2	3					L7	(true)
					F	A	B	C				L9	(false)
									.	T		L12	(true)
										.	F	L12	(false)

Internal	L Edit	Output Field Positions											
Representation	Descriptor	1	2	3	4	5	6	7	8	9	10	11	12
(true)	L6						T						
(false)	L12												F
(true)	L10										T		
(false)	L1	F											
(true)	L1	T											
(false)	L3			F									

8.4.15 A (ALPHANUMERIC) EDITING

The A[*w*] edit descriptor is used with an I/O list item of type character, logical, integer, real, or complex. *w* specifies the field width. If *w* is not specified, the I/O list item must be of type character, in which case the number of characters in the field is the length of the character I/O list item. On input, the input list item becomes defined with character data. On output, the output list item must be defined with character data. Integer, real, and logical I/O list items can contain up to eight characters; complex, up to 16. *w* specifies a field of one to eight characters for list items not of type character.

len is the length of the character list item. If the specified field width for A input is greater than or equal to eight for noncharacter variables or greater than or equal to *len* for character list items, the rightmost eight or *len* characters of the input field form the internal representation. If the specified field width is less than eight or less than *len* in the case of character list items, the characters from the input field are left-justified with 8-*w* or *len*-*w* trailing blank characters added to form the internal representation.

If the specified field width for A output is greater than eight for noncharacter variables or greater than *len* for character list items, the output field consists of *w*-8 or *w*-*len* blanks followed by the characters from the internal representation. If the specified field width is less than or equal to eight (or less than or equal to *len* for type character), the output field consists of the leftmost *w* characters from the internal representation.

Input/output list items of type complex can contain up to 16 characters in two storage units (computer words). Two A edit descriptors are required to store a complex variable. In this case, each is applied to a single I/O list item; the first to the first storage unit, the second to the second storage unit.

The ANSI FORTRAN Standard does not provide for the use of A with noncharacter list items.

Examples:

Input Field Positions 1	2	3	4	5	6	7	8	9	10	11	12	Item Type	A Edit Descriptor(s)	Internal Representation
A	B	C	D	E	F	G	H	I	J	K	L	Integer	A8	'ABCDEFGH'
I	N	D	E	X	.	.	.	.	.	6		Complex	A8,A3	'INDEX.....6'
R	T	C										Integer	A3	'RTC '
A	B	C	D	E	F	G	H	I				Character*6	A	'ABCDEF '

Internal Representation	Item Type	A Edit Descriptor	Output Field Positions 1	2	3	4	5	6	7	8	9
8HABCDEFGH	Integer	A8	A	B	C	D	E	F	G	H	
8HABCDEFGH	Real	A9		A	B	C	D	E	F	G	H
8HA-FORMAT	Integer	A3	A	-	F						
'ABC'	Character	A	A	B	C						
'ABC'	Character	A1	A								
'ABC'	Character	A4		A	B	C					

8.4.16 R (RIGHT-JUSTIFIED) EDITING (CFT EXTENSION)

The Rw edit descriptor is used with an I/O list item of type logical, integer, real, or complex. On input, the input list item becomes defined with w characters of character data. On output, the output list item must be defined with *w* characters. R*w* edit descriptor actions are identical to those of the A*w* edit descriptor with the following two exceptions.

- Characters in an incompletely filled input list item are right-justified with the remainder of that list item containing binary zeros.

- Partial output of an output list item is from its rightmost character positions.

Examples:

Input Field Positions												Item	R Edit	Internal
1	2	3	4	5	6	7	8	9	10	11	12	Type	Descriptor	Representation
A	B	C	D	E	F	G	H	I	J	K	L	Integer	R8	'ABCDEFGH'
R	T	C										Integer	R3	'#####RTC'

(Where # is a null (0) character)

Internal	Item	R Edit	Output Field Positions								
Representation	Type	Descriptor	1	2	3	4	5	6	7	8	9
'ABCDEFGH'	Integer	R8	A	B	C	D	E	F	G	H	
'ABCDEFGH'	Real	R9		A	B	C	D	E	F	G	H
'A-FORMAT'	Integer	R6	F	O	R	M	A	T			

9. CRAY FORTRAN PROGRAMMING

The Cray FORTRAN Compiler (CFT) produces Cray machine language instructions from FORTRAN language statements with run-time efficiency as a prime objective. Its operations include the following.

- Providing the most effective instruction sequence for each FORTRAN statement compiled
- Making full use of all Cray Computer System capabilities and techniques, enhancing execution speed

CFT is particularly effective in compiling statements describing vector processing. When properly applied, vector processing affords dramatic decreases in computation time over equivalent scalar processing methods. The Cray FORTRAN programming techniques are described in this section with emphasis on vector processing.

9.1 VECTORIZABLE DO LOOPS

CFT analyzes the innermost DO loops of the FORTRAN programs it compiles to determine whether vector processing methods can be applied to improve overall program efficiency. If such efficiency can be improved, CFT produces a sequence of code containing vector instructions to drive the high-speed vector and floating-point functional units and the eight vector registers in their specified operation. This feature of CFT is automatically activated through compiler analysis of statements contained in certain DO loops without special notation on the part of the programmer. No special provisions are required that would encumber the programmer or affect the transportability of the programs. However, CFT does provide utility procedures that can enhance vectorization. (See the vectorization utilities in appendix C.)

9.1.1 QUALIFICATIONS FOR VECTORIZATION

Not all DO loops are vectorizable. In determining the qualifications of a loop for vectorization, CFT examines each statement and its relationship to others in that DO-loop range. The Cray FORTRAN programmer can enhance program performance by avoiding certain constructs inhibiting DO-loop vectorization.

To be vectorized, a DO loop must manipulate or perform calculations on the contents of one or more arrays and not have certain constructions that inhibit vectorization. Conditions inhibiting vectorization are:

- CALL statements
- I/O statements
- Inner DO loops
- Backward branches within the loop
- Statement numbers with references from outside the loop
- References to character variables, arrays, or functions
- IF statements which may not execute due to the effects of previous IF statements
- ELSEIF statements
- External function references not declared on a CDIR$ VFUNCTION directive
- RETURN, STOP, and PAUSE statements
- NOVECTOR and BLOCK compiler directives
- Bounds checking on any array referenced in the loop
- Specifying the DEBUG option
- Loop size exceeds the optimized MAXBLOCK size
- Loop has been unrolled or replaced by a $SCILIB routine

IF statements of the form

IF($variable_1$.*rop*. exp_1) $variable_1$ = exp_1
IF(exp_2 .*rop*. $variable_2$) $variable_2$ = exp_2

where $variable_1$ and $variable_2$ are variables, exp_1 and exp_2 are expressions of the same type, and .*rop*. is .GT., .GE., .LT., or .LE. are compiled as if written as

variable = MAX (*variable, exp*), or
variable = MIN (*variable, exp*).

See section 1 for the conditions under which IF statements of the form

IF(*logical exp*)*var=expression*

are compiled as conditional vector merges.

Blocks executed conditionally on the outcome of an IF statement are vectorized with a compressed index (see the description of compressed index references in this section).

9.1.2 ENTITY CATEGORIES

Loop analysis is performed to determine if all defined or referenced entities in the DO-loop range are in one of the following categories.

- Invariant - Constant or variable referenced but not redefined in the course of a DO loop

- Invariant expression - Arithmetic expression only with invariants

- Invariant array element - An array element where all subscript expressions are invariant expressions

- Constant increment variable (CIV) - A variable of type INTEGER, REAL, DOUBLE PRECISION, or COMPLEX that is incremented or decremented once during each pass through a DO loop by an invariant expression. The CIV definition must be in a statement which is executed during every DO-loop iteration. The expression defining a CIV can reference itself or another CIV. The expression must not use operators other than plus or minus or involve expressions containing parentheses. For example,

  CIV=CIV±(INVARIANT EXPRESSION)

- Vector array reference - An array element where one subscript expression contains one CIV reference and where any other subscript expression is an invariant expression. The subscript expression containing the CIV must be a linear expression algebraically reducible to the following form

  $$[\pm \textit{invariant expression}_1 *]\text{CIV}[\pm \textit{invariant expression}_2]$$

  where the only operators, if any, in invariant expression_1 are multiply operators. Using parentheses in the CIV subscript may prevent vectorization. Some of the more common forms of array references with nested parentheses are converted to vector array references. For example, the array reference A(3*(I-2)) is converted to the vector array reference A(I*3-6). The following format is generally used

  $$\begin{bmatrix} + \textit{ integer variable} & + \\ - \textit{ integer constant} & * \\ & - \end{bmatrix} \left(\begin{matrix} \textit{integer variable} \\ \textit{integer constant} \\ \end{matrix} \begin{bmatrix} + \textit{ integer variable} \\ * \textit{ integer constant} \\ - \end{bmatrix} \right)$$

  where at least one term inside the parentheses is a variable.

  The following examples show array references that are converted to vector array references:

Array References	Vector Array References
A((I+2))	A(I+2)
A(3-(I+2))	A(1-I)
A(J*(I+K))	A(J*I+J*K)
A((I-2)+3)	A(I+1)
A(K+(I*3))	A(K+I*3)

The following examples show array references that are not converted to vector array references:

Array References	Reasons
A((I+(3-(J)))	Parentheses are nested too deeply
A(I*(3-2))	One inner term must be a variable
A(2*(I+K+2))	Too many inner terms
A(2*I+(K+2))	Too many outer terms

- Scalar temporary - A variable set equal to a vectorizable expression during each pass through a DO loop. A scalar temporary cannot be defined before or used after a statement number reference in a vector loop.

- Variable or invariant array element used in a reduction array operation. The item must appear on both the left and the right sides of the equal sign. On the right, it must be a summand, multiplicand, dividend, or minuend. For example,

```
X = X+A(I)*B
X = X-A(I)*B

Y = Y*(A(I)+4)
Y = Y/(A(I)+4).
```

  The type of reduction variable must be INTEGER or REAL. Real operations between the reduction variable and the remaining expression are limited to addition, subtraction, multiplication, and division. Integer operations are limited to addition and subtraction. No other operations are allowed between the reduction variable and the remaining expression.

- Pseudo vector - An array reference which does not meet the previous requirements, but has a subscript expression that vectorizes and no dependencies will pseudo vectorize. The array reference is treated as a scalar subloop inside the vector loop, and the subscript expression is computed as a vector expression. A single instruction can be compiled instead of the scalar subloop on a Cray Computer System with the appropriate hardware. The subscript portion of a subscript reference is a vectorizable expression and partially vectorizes; that is, CFT generates a separate scalar loop to handle the subscript reference and vectorize the remaining loop.

In the following example, I, J, and K are CIVs; A, B, and C are vector array references; KDELTA, 107, 3, 2, 7, M, L, and X are invariants; D(L,M) is an invariant array element; and E is a pseudo vector.

```
         DO 10 I = 3,101,2
         K = K - KDELTA
         J = 107 - I
         A(3,I-2) = COS(B(J)) **C(M-2*K+L*M/7,L,M/L)*X*D(L,M)
         E(I,J) = 0
    10   CONTINUE
```

9.1.3 DEPENDENCIES

CFT inhibits vectorization of DO loops with dependencies. The following example of a DO loop shows a dependency within CFT. In this example, the first seven elements of array A are 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, and 7.0, respectively.

```
         DO 10 I = 2,7
         A(I) = A(I-1)
    10   CONTINUE
```

The results of array A differ, depending on the mode type the loop is executing, vector mode or scalar mode. Table 9-1 shows the first seven elements of array A in vector and scalar modes.

Table 9-1. Array A Elements in Vector and Scalar Modes

Array A Elements	1	2	3	4	5	6	7
Vector mode	1.0	1.0	2.0	3.0	4.0	5.0	6.0
Scalar mode	1.0	1.0	1.0	1.0	1.0	1.0	1.0

The scalar results are correct by default. CFT detects that the vector results may be different from the scalar results for the DO loop and inhibits vectorization. A dependency exists if the following two conditions are met.

- An array is referenced and defined in the DO loop
- An array element defined in a previous pass of the DO loop is referenced

CFT can internally change the order of the definition and reference, eliminating an apparent dependency, if the reference and definition are not conditionally executed.

In the previous example, A(I) is the definition and A(I-1) is the reference. A definition is on the left side of the = operator. The dependency detected is called a previous minus with an incrementing subscript.

The term previous means a reference in a statement occurs before the definition statement, or a reference is on the right side of the = operator. The term subsequent means a reference in a statement occurs after the definition statement. The following diagram describes the terms previous and subsequent.

```
_____________

 Previous

_________

 A(I) =  ____

_____________

 Subsequent

_____________
```

The following example also has a previous reference because the reference, A(I-1), is in a statement before the definition A(I).

```
         DO 20 I = 2,M
         B(I) = A(I-1)
         A(I) = C(I)
    20   CONTINUE
```

The term minus means the subscript of the reference (I-1) is less than the subscript of the definition (I). In this example, subscript I is incrementing. Both of the examples have a previous minus with an incrementing subscript dependency.

The following information is required to determine if there is a dependency.

- Previous or subsequent reference
- Plus or minus subscript difference
- Incrementing or decrementing subscript

Table 9-2 shows dependency information combinations and results.

Table 9-2. Dependency Information Combinations

Dependency Information Combinations	Results
Previous minus with an incrementing subscript	Dependency
Previous minus with a decrementing subscript	Vector
Previous plus with an incrementing subscript	Vector
Previous plus with a decrementing subscript	Dependency
Subsequent minus with an incrementing subscript	Vector
Subsequent minus with a decrementing subscript	Vector
Subsequent plus with an incrementing subscript	Vector
Subsequent plus with a decrementing subscript	Vector

The following two DO loops have previous plus with decrementing subscript dependencies.

```
      DO 30 I = 99,1,-1
      A(I) = A(I+1)
 30   CONTINUE

      DO 40 I = 99,N,-1
      B(I) = A(I+1)
      A(I) = C(I)
 40   CONTINUE
```

The following two DO loops also have dependencies.

```
      DO 50 I = 1,99
      A(I) = B(I)
      A(I+1) = C(I)
 50   CONTINUE

      DO 60 I = 99,1,-1
      A(I) = B(I)
      A(I-1) = C(I)
 60   CONTINUE
```

In these two examples, A never appears on the right side of the = operator, therefore, assigning the reference and definition labels is ambiguous. In this case, it is assumed the last statement has the definition. The definitions for the first and second examples are A(I+1) and A(I-1), respectively. This means the first example has a previous minus with an incrementing subscript dependency and the second example has a previous plus with a decrementing subscript dependency.

This example is a previous plus with a decrementing subscript dependency.

```
          DO 10 I = 1,10
          A(11-I) = A(12-I)
     10   CONTINUE
```

The DO control variable is an incrementing I and the subscripts (11-I) and (12-I) are decrementing. Therefore, this DO loop would cause CFT to detect a previous plus with a decrementing subscript dependency.

The following DO loops do not have dependencies.

Previous minus with a decrementing subscript

```
          DO 10 I = 100,2,-1
          A(I) = A(I-1)
     10   CONTINUE
```

Previous plus with an incrementing subscript

```
          DO 20 I = 1,99
          B(I) = A(I+1)
          A(I) = 3.0
     20   CONTINUE
```

Subsequent minus with an incrementing subscript

```
          DO 30 I = 2,100
          A(I) = B(I)
          C(I) = A(I-1)
     30   CONTINUE
```

Subsequent minus with a decrementing subscript

```
          DO 40 I = 100,2,-1
          A(I+1) = B(I)
          C(I) = A(I)
     40   CONTINUE
```

Subsequent plus with an incrementing subscript

```
          DO 50 I = 1,100
          A(I) = B(I)
          C(I) = A(I+1)
     50   CONTINUE
```

Subsequent plus with a decrementing subscript

```
          DO 60 I = 2,100
          A(I-1) = B(I)
          C(I) = A(I)
     60   CONTINUE
```

Two types of dependency messages are issued by CFT.

```
AT SEQUENCE NUMBER - m
PRNAME name COMMENT - DEPENDENCY INVOLVING ARRAY "name"
```

is issued when the definition and reference appear in the same statement with sequence number *m*.

```
AT SEQUENCE NUMBER - m
PRNAME name COMMENT - DEPENDENCY INVOLVING ARRAY "name" IN
SEQUENCE NUMBER n
```

is issued when the definition appears in a statement with sequence number *m* and the reference appears in a statement with sequence number *n*. When a dependency message is issued, an informative dependency message also appears, explaining why the dependency exists. See appendix D for a description of these messages.

All the previous examples had a subscript incremented or decremented by 1. The following three examples have DO loops with an increment and decrement other than 1.

```
     DO 10 I = 10,20,2
     A(I) = A(I-1)
10   CONTINUE
```

The first example has a previous minus with an incrementing CIV dependency. Because the increment is 2, A(I) and A(I-1) never access the same array elements. A(I) accesses elements 10, 12, 14, 16, 18, and 20. A(I-1) accesses elements 9, 11, 13, 15, 17, and 19. This type of DO loop vectorizes.

```
     DO 20 I = 20,10,-2
     A(I) = A(I+3)
20   CONTINUE
```

This example also has a dependency but A(I) and A(I+3) access different array elements. This DO loop also vectorizes.

```
     DO 30 I = 10,20,2
     A(I) = A(I-4)
30   CONTINUE
```

In this example, A(I) and A(I-4) access some of the same array elements. A(I) accesses elements 10, 12, 14, 16, 18, and 20. A(I-4) accesses 6, 8, 10, 12, 14, and 16. In this case, a dependency message is issued.

CFT cannot always determine if the subscript is incrementing or decrementing as shown in the following example:

```
     DO 40 I = M,N,J
     A(I) = A(I+2)
40   CONTINUE
```

If J is positive, the subscript is incrementing and there is no dependency. If J is negative, the subscript is decrementing and a previous plus with a decrementing subscript dependency exists. This increment is ambiguous and causes a dependency message to be issued.

CFT cannot always determine if the subscript difference is minus or plus.

```
         DO 50 I = 1,100
         A(I) = B(I)
         C(I) = A(I+J)
    50   CONTINUE
```

If J is positive, the subscript difference is plus. If J is negative, the subscript difference is minus. This example also causes a dependency message to be issued.

If the value of J is known not to cause a dependency, an IVDEP or IVDMO compiler directive can be used, allowing CFT to generate vector code for a DO loop with dependencies (see section 1 for the format of compiler directives).

Equivalenced arrays can introduce different dependency problems related to the storage overlap, as in the following example.

```
         DIMENSION A(6),B(6),X(6)
         EQUIVALENCE (B,A(3))
         DO 5 I = 2,5
         A(I) = ...
         X(I) = B(I)
    5    CONTINUE
```

This sequence causes the multiple-statement dependency message to be printed. The message refers to the dependency between A(I) and B(I). The message uses only one array name rather than both names. In general, messages concerned with equivalenced arrays print only the first name encountered in processing the declarative statement.

The inhibiting of vectorization because of such dependencies can be relaxed in the case of multiply-dimensioned array processing. CFT must be able to determine that the specified array elements are in different vectors (that is, rows, columns, planes, etc.) of the array. For example, the loop

```
         DO 10 I = 2,100
    10   A(I,J) = A(I-1,J-1)
```

is vectorizable, while the similar loop

```
         DO 20 I = 2,100
    20   A(I,J) = A(I-1,JMINUS1)
```

is conditionally vectorized with a run-time test to determine whether J and JMINUS1 are equal.

The compiler directive IVDEP can be placed in advance of an inner DO loop DO statement to cause vector dependencies to be ignored in determining whether or not to vectorize that loop. (See section 1 for a description of IVDEP and other compiler directives.)

9.1.4 CONDITIONAL VECTOR LOOPS

If CFT cannot determine at compile time that a loop can be correctly vectorized, a run-time test is performed to ensure correct vectorization. CFT generates scalar and vector versions of the loop with a run-time test to select which version will execute. CFT conditionally vectorizes loops with ambiguous dependencies from zero CIV increments, unequal invariant subscripts, and mismatched CIVs in subscript expressions.

For example, when OPT=ZEROINC is specified a loop such as

```
         DO 1 I = 1,N
         J = J+JINC
    1    A(J) = A(J)+B(I)
```

will be conditionally vectorized with a test for JINC=0. The loop

```
         DO 2 I = 1,N
    2    A(I,J) = A(I,JMINUS1)
```

will be conditionally vectorized with a test for J=JMINUS1. The loop

```
         DO 2 I = 1,N
         J = J+1
    2    A(J) = A(I)
```

will be conditionally vectorized with a test for J{1 or J↑N.

9.1.5 VECTORIZATION WITH ARRAYS

Because CFT allows only one subscript in an array reference to be variant, loops that reference the diagonal of an array are not fully vectorized. A loop such as

```
         DIMENSION A(N,N)
         DO 10 I = 1,N
     10  A(I,I) = ...
```

can be rewritten as

```
         DIMENSION A(N,N)
         J = 1
         DO 10 I = 1,N
         A(J)= ...
     10  J = J+N+1
```

or as

```
         DIMENSION A(N,N),B(N*N)
         EQUIVALENCE (A,B)
         J = -N
         DO 10 I = 1,N
         J = J+N+1
     10  B(J) =  ...
```

The first case is allowed by CFT but the use of one subscript rather than two causes a warning level diagnostic. Since array operations typically execute in times proportional to N^2 and diagonal operations execute in times proportional to N, vectorizing the diagonal operations might not have a significant effect on overall program execution time.

CFT allows variables and array elements to be defined within a vectorized loop as CIVs, scalar temporaries, or as recursively defined terms. (CIVs are discussed earlier in this section.) A scalar temporary is a variable set equal to a vectorizable expression. Recursively defined terms must be defined by integer addition or subtraction or real addition, subtraction or multiplication.

The following loop is vectorizable.

```
         REAL A(100),B(100,100),C(100)
         INTEGER II(100)
         DO 10 I = 1,100
         T = B(7,I) + A(I)*B(I,7)
         C(7) = C(7) + T*SQRT(T)
         T = B(I,7)*B(11,I)
         PROD = PROD*(T + A(101-I))
         II(I) = II(I) + 1
    10   ISUM = ISUM + II(I)
```

The following example describes pseudo vectorization. I, J, and K are CIVs and A, B, C, II, and JJ are arrays without dependencies.

```
         A(II(I)) = B(JJ(I))
         K = C(I/J)
         Y = A(INT(SIN(B(I))*X))
```

9.1.6 USING OPTIMIZED ROUTINES

The efficiency of the vectorization depends on the number of iterations of the loop and the complexity of the loop. In many cases, a loop with a large number of iterations and simple calculations producing a single scalar result (for example, a dot product or a sum) should be replaced with a call to an optimized routine in the $SCILIB library. The vector sum

```
         SUM = 0.0
         DO 10 I = 1,100
         SUM = SUM + A(I)
```

is better written as

```
         SUM = SSUM(100,A(1),1).
```

Nested DO loops producing a vector result should also be replaced with a call to an optimized routine in the $SCILIB library. For example, the following matrix multiply

```
         DO 10 I = 1,N
         DO 10 J = 1,M
         A(I,J) = 0.0
         DO 10 L = 1,K
         A(I,J) = A(I,J) + B(I,L) * C(L,J)
```

is better written as

```
         CALL MXM(B,N,C,K,A,M).
```

For more examples on optimized routines in $SCILIB library see the Library Reference Manual, CRI publication SR-0014.

9.1.7 USE OF OPTIMIZED ROUTINES BY CFT

Several one-line DO loops are recognized by CFT and automatically replaced by a call to the proper library routine. Vector sums and vector dot products are replaced by calls to library routines SSUM and SDOT, respectively. The vector sum

```
         DO 10 I = 1,N
    10   S = S + A(I)
```

is automatically replaced by

```
         S = S + SSUM(N,A(1),1)
```

and the vector dot product

```
         DO 10 I = 1,N
    10   C(J,K) = C(J,K) + A(I,K) * B(I,K)
```

is automatically replaced by

```
         C(J,K) = C(J,K) + SDOT(N,A(1,K),1,B(1,K),1).
```

For more information on SSUM and SDOT, see the Programmer's Library Reference Manual, CRI publication SR-0113.

One-line DO loops calculating a single vector result (for example, first order linear recurrences) are recognized and automatically replaced by a call to the $SCILIB library FOLR, FOLR2, FOLRP, and FOLR2P. The example,

```
         DO 10 I = 2,N
    10   B(I) = B(I) - A(I) * B(I-1)
```

is automatically replaced by

```
         CALL FOLR(N,A(1),1,B(1),1).
```

Similarly

```
         DO 10 I = 3,500,2
    10   C(I) = B(I) - A(I) * C(I-2)
```

is automatically replaced by

```
         CALL FOLR2(500,A(1),2,B(1),2,C(1),2).
```

Routines FOLRP and FOLR2P are called when the DO loop statement's additive operation is addition instead of subtraction. For more information on first-order linear recurrences, see the Programmer's Library Reference Manual, CRI publication SR-0113.

Follow these guidelines for writing one-line DO loops that will be optimized by replacement with library calls.

- The DO-loop body must be one and only one FORTRAN statement long. The terminating statement number can be on the same line or on the following line with a CONTINUE statement.

- The one-line DO-loop body must be a vector sum, vector dot product, or a first order linear recurrence.

- All terms must be single-precision real and not equivalenced.

- Keep array subscripts simple; that is, of the form A(*invariant* * *variable* + *invariant*). Other loops that vectorize are less restrictive with subscript complexity than one-line DO-loop replacement.

- The DO-loop increment (m_3) must be a positive constant value.

9.1.7.1 Conditional statements

On some machines, code generated for vectorizable IF statements is inefficient. The instruction functions CVMGT, CVMGP, CVMGM, CVMGZ, CVMGN, MAX, and MIN can often be used to replace the IF statement with more efficient statements.

CFT automatically replaces some IF statements. IF statements of the form IF(*var.op.expression*)*var=expression* can be optimized to produce code similar to *var=function*(*var,expression*) where *function* is a version of the MAX/MIN functions. For this form of optimization to occur, *.op.* must be one of the relational operators .GT., .GE., .LT., or .LE.; *var* and *expression* must be the same type, either REAL, INTEGER, or DOUBLE PRECISION. Examples of IF statements optimized by CFT in this way are

IF(A(I).GT.B(I))A(I)=B(I)
IF(I_1.LE.I_2)I_2=I_1
IF((I+3)*R_1.GT.R_2)R_2=(I+3)*R_1.

A more general form of conditional replacement statements can also be optimized by CFT. Statements of the form IF(*logical exp*)*var=expression* can be optimized to produce code similar to that for *var*=CVMG*x*(*expression,var,condition*) where CVMG*x* is a vector merge function, *var* is type INTEGER, REAL, or LOGICAL, and *condition* is a logical expression. Examples of IF statements which CFT can optimize in this manner are

IF($COND_1$.OR.$COND_2$)B(I)=C(I)
IF((B(I).GT.C(I)).OR.(B(I).LT.A(I))B(I)=ABS(A(I)*C(I))
IF(I.GT.R)I=R (the types are mismatched so a MAX/MIN optimization will not occur).

Conditional replacement statements of the form

IF(*cond*)*var=var op exp*

where *op* is the operator +, -, *, or /; and

IF(*cond*)*var=exp op var*

where *op* is the operator + or * may be restructured as if written as

var=*var* *op*CVMG*x*(*exp*,*ident*,*cond*)

where *ident* is 0 when *op* is + or - and *ident* is 1 when *op* is * or /. This form of restructuring occurs only if *exp* does not contain unparenthesized operators of lower priority than *op*. If *op* is - or /, *exp* cannot contain unparenthesized operators of the same priority as *op*. This form of the IF conversion allows a vector reduction loop to be generated if *var* is a scalar reference and the statement appears in a loop that otherwise would be vectorizable.

There are two possible drawbacks to performing this type of optimization in all cases. An illegal operation may occur. For example, IF(X.NE.0.0)R=R/X. If optimization occurred, an error would occur if X were equal to zero. Another example is, IF(X.GE.0.0)R=SQRT(X). An error would occur if X is negative.

The second drawback occurs when the IF statement appears in a DO loop and the logical expression is usually false. For example,

```
      DO 10, I = 1,100
      A(I) = B(I)*C(I)
      IF(A(I).GT.RMAX)A(I) = MAX(B(I),C(I))
30    CONTINUE
```

In this example, if A(I) were less than RMAX and the IF statement was not optimized, MAX(B(I),C(I)) would never be evaluated. If the IF statement were optimized, MAX(B(I),C(I)) would always be evaluated.

Because of these drawbacks, the user can control vector merge optimizations. By specifying OPT=NOIFCON on the CFT control card, this form of optimization is disabled (the default level of optimization). OPT=PARTIALIFCON allows the optimization to occur when the replacement expression does not involve division or an external function reference. OPT=FULLIFCON enables the optimization for all cases, including those involving division or external functions. The compiler directives CDIR$ NOIFCON and CDIR$ RESUMEIFCON may be used to turn the optimization off and on around unsafe cases (such as, division by zero if OPT=FULLIFCON) or when the logical expression of the IF statement is usually false. The following examples may be helpful in performing optimizations.

The simple case

```
      DO 10 I = N,M
      X(I) = C(I)
      IF(B(I).GT.C(I))X(I) = B(I)
10    CONTINUE
```

could be rewritten as

```
      DO 10 I = N,M
      X(I) = CVMGT(B(I),C(I),B(I).GT.C(I))
   10 CONTINUE
```

or as

```
      DO 10 I = N,M
      X(I) = AMAX1(B(I),C(I))
   10 CONTINUE
```

to produce vectorizable loops.

Similarly,

```
      DO 10 I = N,M
      IF(X(I).GE.10.)X(I) = X(I) + 1.0
   10 CONTINUE
```

could be rewritten as

```
      DO 10 I = N,M
      X(I) = CVMGP(X(I) + 1.0,X(I),X(I)-10.)
   10 CONTINUE.
```

The library routines SENSEFI, SETFI, and CLEARFI, or the EFI and DFI CAL instructions can be used to control floating-point interrupts on a loop-by-loop basis and MAX or MIN functions can be used to protect function references.

Example:

```
                .
                .
                .
            CALL SENSEFI (MODE)
            CALL CLEARFI
            DO 10 I = 1,100
            X(I) = CVMGN(1./X(I),X(I),X(I))
            Y(I) = CVMGP(SQRT(AMAX1(Y(I),0.0)),Y(I),Y(I))
         10 CONTINUE
            IF(MODE.NE.0) CALL SETFI
                .
                .
                .
```

9.1.8 COMPRESSED INDEX REFERENCES

Memory references are in blocks of code which may or may not be executed depending on the outcome of an IF statement, and are compiled as compressed index references. The variable values for the loop iterations executed by the IF statement are collected as vector values, and the subscripts are computed as vector expressions which generate a pseudo vector memory reference. For example,

```
      DO 1 I = 1,N
      IF(MOD(I,2).EQ.0)THEN
      A(I) = B(I)
      ENDIF
 1    CONTINUE
```

When I is even, the elements of B are moved to the corresponding elements of A.

If there are no appropriate vector instructions to compress an index, the instructions are simulated with a compiler-generated instruction sequence.

9.1.9 GENERAL GUIDELINES FOR VECTORIZATION

Follow the general guidelines given below to promote vectorization of DO-loop operations.

- Keep subscripts simple and explicit; do not use parentheses in subscripts.

- Do not use GO TO or CALL statements.

- Use the Cray FORTRAN intrinsic functions where appropriate.

- Make judicious use of the Cray FORTRAN intrinsic functions CVMGT, CVMGP, CVMGM, CVMGZ, CVMGN, and the MAX and MIN functions instead of IF statements. For more information, see the subsection on using optimized routines described earlier in this section.

- Rewrite large loops containing a few unvectorizable statements as two or more loops, one or more of which will vectorize.

9.2 BIDIRECTIONAL MEMORY

Bidirectional Memory is memory which can be read from and written to simultaneously. The CRAY X-MP Computer Systems have Bidirectional Memory (multiple ports to memory) and CFT uses it to enhance the performance of FORTRAN programs. This section describes how CFT uses Bidirectional Memory, beginning with CFT version 1.11.

CFT attempts to use Bidirectional Memory in all vectorizable loops. In loops where Bidirectional Memory may cause incorrect results, CFT inserts instructions forcing sequential, instead of bidirectional, accesses to memory. Using Bidirectional Memory is only a concern when using vector loops, since scalar memory operations are always sequential.

Bidirectional Memory can always be used if there are no overlaps between the arrays in memory. For example,

```
      PROGRAM EXAMPLE1
      COMMON B(10),C(10)
      DIMENSION D(10),E(10)
```

The values of B, C, D, and E occupy different areas in memory and therefore, have no Bidirectional Memory errors.

CFT assumes that subscripts are within bounds, dummy arguments are independent, and arrays referenced by pointer variables are independent. In the following example,

```
      SUBROUTINE EXAMPLE2(B,C)
      REAL B(100),C(100)
      COMMON D(100),E(100)
      POINTER (IF,F(100)),(IG,G(100))
```

statements such as

```
      CALL EXAMPLE2 (X(1),X(2))
```

or

```
      IG = IF + 1
```

or

```
      DO 10 I = 1,50
   10 D(I+99) = E(25-I)
```

do not follow the CFT assumptions and produce unpredictable results. The errors are not Bidirectional Memory errors but programming errors caused by the incorrect use of FORTRAN.

Chaining operation results will always be correct when Bidirectional Memory is used. In the loop:

```
      DO 10 I = 1,100
10    A(I) = A(I) + 1.0
```

the first A(I) will be loaded before the second A(I) is stored.

Multiple stores into the same array will always be correct, since there is only one write port to memory. For example, the loop:

```
      DO 10 I = 1,N
      A(K*I+J) = ...
10    A(J*I+M) = ...
```

will be correct when Bidirectional Memory is used.

When CFT generates code, it ensures that previous dependent load and store operations are complete before subsequent operations begin. This may be ensured by the nature of the code (as in the previous examples); if it is not, CFT generates protective code that guarantees completion of previous dependent operations.

CFT considers two types of dependencies when looking for Bidirectional Memory dependencies.

- A store operation preceded by a load operation. The two load operations most recently compiled are examined (because there are two read ports to memory). If either load operation has an order dependency and does not chain into the store operation, protective code is generated.

- A load operation preceded by a store operation. The store operation most recently compiled is examined (because there is one write port to memory). If the store operation has an order dependency on the load operation, protective code is generated.

CFT also considers two special cases when looking for Bidirectional Memory dependencies: loop wrap-around dependencies and loop-to-loop dependencies. In a loop such as

```
CDIR$ IVDEP
      DO 10 I = 1,N
      ... = A(I)
      ...
      ...
10    A(I+J) = ...
```

the store operation at the end of one iteration must be completed before the load at the beginning of the next iteration begins. CFT 1.11 unconditionally inserts a CMR instruction (complete memory references) as the first instruction of a loop preceded by an IVDEP directive (CFT does not normally vectorize a loop with a wrap-around dependency). CFT 1.13

and 1.14 perform a load/store analysis on the first two load operations and the last store operation. If a potential dependency is found, protective code is generated.

In the two adjacent loops

```
         DO 10 I = 1,64
    10   A(I) = ...

         DO 20 I = 63,64
    20   B(I) = A(I)
```

the store operation in line 10 may not be completed before the load operation in line 20 begins.

Before each vector loop, CFT generates a CMR instruction as the last instruction in the loop preamble. Therefore, no vector memory operations are in progress when a loop begins processing.

In a loop such as

```
   CDIR$ IVDEP
         DO 10 I = 1,10
         ... = A(I+M)
    10   A(I) = ...
```

where M↑10, CFT adds unnecessary and undesirable protective code. The IVDMO directive can be used to prevent CFT from adding this code (see section 1).

APPENDIX SECTION

A. CHARACTER SET

The ASCII character set contains 128 control and graphic characters shown in the following table. Numbers, letters, and special characters that form the Cray FORTRAN character set are identified by the appearance of the letter C in the fourth column. All other characters are members of the auxiliary character set. The letter A in the fourth column of the table indicates those characters belonging to the ANSI FORTRAN character set (see table A-1).

The letters that appear in parentheses following the descriptions in the fifth column indicate the following control character usage.

- CC - Communication control
- FE - Format effector
- IS - Information separator

Table A-1. Character Set

Character	ASCII Octal Code	ASCII Punched-card Code	FORTRAN (A=ANSI) (C=CRAY)	Description
NUL	000	12-0-9-8-1		Null
SOH	001	12-9-1		Start of heading (CC)
STX	002	12-9-2		Start of text (CC)
ETX	003	12-9-3		End of text (CC)
EOT	004	9-7		End of transmission (CC)
ENQ	005	0-9-8-5		Enquiry (CC)
ACK	006	0-9-8-6		Acknowledge (CC)
BEL	007	0-9-8-7		Bell (audible or attention signal)
BS	010	11-9-6		Backspace (FE)
HT	011	12-9-5		Horizontal tabulation (FE)
LF	012	0-9-5		Line feed (FE)
VT	013	12-9-8-3		Vertical tabulation (FE)
FF	014	12-9-8-4		Form feed (FE)
CR	015	12-9-8-5		Carriage return (FE)
SO	016	12-9-8-6		Shift out
SI	017	12-9-8-7		Shift in
DLE	020	12-11-9-8-1		Data link escape (CC)
DC1	021	11-9-1		Device control 1
DC2	022	11-9-2		Device control 2
DC3	023	11-9-3		Device control 3
DC4	024	9-8-4		Device control 4 (stop)
NAK	025	9-8-5		Negative acknowledge (CC)
SYN	026	9-2		Synchronous idle (CC)
ETB	027	0-9-6		End of transmission block (CC)
CAN	030	11-9-8		Cancel
EM	031	11-9-8-1		End of medium
SUB	032	0-8-7		Substitute
ESC	033	0-9-7		Escape
FS	034	11-9-8-4		File separator (IS)
GS	035	11-9-8-5		Group separator (IS)
RS	036	11-9-8-6		Record separator (IS)
US	037	11-9-8-7		Unit separator (IS)
(Space)	040	(None)	A,C	Space (blank)
!	041	12-8-7		Exclamation mark
"	042	8-7	C	Quotation marks (diaeresis)
#	043	8-3		Number sign
$	044	11-8-3	A,C	Dollar sign (currency symbol)
%	045	0-8-4		Percent
&	046	12		Ampersand
'	047	8-5	A,C	Apostrophe (single close quotation)

Table A-1. Character Set (continued)

Character	ASCII Octal Code	ASCII Punched-card Code	FORTRAN (A=ANSI) (C=CRAY)	Description
(	050	12-8-5	A,C	Opening (left) parenthesis
)	051	11-8-5	A,C	Closing (right) parenthesis
*	052	11-8-4	A,C	Asterisk
+	053	12-8-6	A,C	Plus
,	054	0-8-3	A,C	Comma (cedilla)
-	055	11	A,C	Minus (hyphen)
.	056	12-8-3	A,C	Period (decimal point)
/	057	0-1	A,C	Slant (slash, virgule)
0	060	0	A,C	Zero
1	061	1	A,C	One
2	062	2	A,C	Two
3	063	3	A,C	Three
4	064	4	A,C	Four
5	065	5	A,C	Five
6	066	6	A,C	Six
7	067	7	A,C	Seven
8	070	8	A,C	Eight
9	071	9	A,C	Nine
:	072	8-2	A,C	Colon
;	073	11-8-6		Semicolon
<	074	12-8-4		Less than
=	075	8-6	A,C	Equal
>	076	0-8-6		Greater than
?	077	0-8-7		Question mark
@	100	8-4		Commercial at-sign
A	101	12-1	A,C	Uppercase letter
B	102	12-2	A,C	Uppercase letter
C	103	12-3	A,C	Uppercase letter
D	104	12-4	A,C	Uppercase letter
E	105	12-5	A,C	Uppercase letter
F	106	12-6	A,C	Uppercase letter
G	107	12-7	A,C	Uppercase letter
H	110	12-8	A,C	Uppercase letter
I	111	12-9	A,C	Uppercase letter
J	112	11-1	A,C	Uppercase letter
K	113	11-2	A,C	Uppercase letter
L	114	11-3	A,C	Uppercase letter
M	115	11-4	A,C	Uppercase letter
N	116	11-5	A,C	Uppercase letter
O	117	11-6	A,C	Uppercase letter
P	120	11-7	A,C	Uppercase letter
Q	121	11-8	A,C	Uppercase letter

Table A-1. Character Set (continued)

Character	ASCII Octal Code	ASCII Punched-card Code	FORTRAN (A=ANSI) (C=CRAY)	Description
R	122	11-9	A,C	Uppercase letter
S	123	0-2	A,C	Uppercase letter
T	124	0-3	A,C	Uppercase letter
U	125	0-4	A,C	Uppercase letter
V	126	0-5	A,C	Uppercase letter
W	127	0-6	A,C	Uppercase letter
X	130	0-7	A,C	Uppercase letter
Y	131	0-8	A,C	Uppercase letter
Z	132	0-9	A,C	Uppercase letter
[	133	12-8-2		Opening (left) bracket
\	134	0-8-2		Reverse slant (backslash)
]	135	11-8-2		Closing (right) bracket
	136	11-8-7		Circumflex
-	137	0-8-5		Underline
'	140	8-1		Grave accent (single open quotation)
a	141	12-0-1	C	Lowercase letter
b	142	12-0-2	C	Lowercase letter
c	143	12-0-3	C	Lowercase letter
d	144	12-0-4	C	Lowercase letter
e	145	12-0-5	C	Lowercase letter
f	146	12-0-6	C	Lowercase letter
g	147	12-0-7	C	Lowercase letter
h	150	12-0-8	C	Lowercase letter
i	151	12-0-9	C	Lowercase letter
j	152	12-11-1	C	Lowercase letter
k	153	12-11-2	C	Lowercase letter
l	154	12-11-3	C	Lowercase letter
m	155	12-11-4	C	Lowercase letter
n	156	12-11-5	C	Lowercase letter
o	157	12-11-6	C	Lowercase letter
p	160	12-11-7	C	Lowercase letter
q	161	12-11-8	C	Lowercase letter
r	162	12-11-9	C	Lowercase letter
s	163	11-0-2	C	Lowercase letter
t	164	11-0-3	C	Lowercase letter
u	165	11-0-4	C	Lowercase letter
v	166	11-0-5	C	Lowercase letter
w	167	11-0-6	C	Lowercase letter
x	170	11-0-7	C	Lowercase letter
y	171	11-0-8	C	Lowercase letter
z	172	11-0-9	C	Lowercase letter

Table A-1. Character Set (continued)

Character	ASCII Octal Code	ASCII Punched-card Code	FORTRAN (A=ANSI) (C=CRAY)	Description
{	173	12-0		Opening (left) brace
|	174	12-11		Vertical line
}	175	11-0		Closing (right) brace
~	176	11-0-1		Overline (tilde, general accent)
DEL	177	12-9-7		Delete

B. CRAY FORTRAN INTRINSIC FUNCTIONS

The tables in this section show the intrinsic functions available with CFT. Conventions used in the tables are described below.

The rightmost column in each table includes letter codes to indicate conformance with the ANSI standard, the level of vectorization, and the kind of code generated. The codes use the following format:

First letter: ANSI standard
- A The function is specified in the ANSI FORTRAN 77 standard.
- C The function is a Cray extension to the standard.

Second letter: level of vectorization
- F Full vectorization
- P Pseudo vectorization
- N No vectorization.

Third letter: code generation
- E External code is generated.
- I In-line code is generated.

Unless noted otherwise, the first function name in each group of functions can be used as a *generic function name*. A generic function name can be used to call any of a group of related functions, so that one name can be used with arguments of different data types (see section 2). Groups of related functions are separated by horizontal lines, with the arguments and results shown for each specific function.

In the Definition column, y is the function's result, and the x values are function arguments separated by parentheses in the function call. Square brackets indicate truncation of a term: if x has a value of 5.67, [*x*] equals 5.

Data types shown in the Function and Argument columns are as follows:

B	Boolean
C	Complex
CH	Character
D	Double precision
I	Integer
L	Logical
P	Pointer
R	Real

Table B-1. General Arithmetic Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Truncation	$y=[x]$ (fraction lost; no rounding)	AINT R DINT D	1 1	R D	$\lvert x\rvert<2^{46}$ $\lvert x\rvert<2^{95}$	A F I A F E
Nearest whole number	$y=[x+.5]$ if $x\geq 0$ $y=[x-.5]$ if $x<0$	ANINT R DNINT D	1 1	R D	$\lvert x\rvert<2^{46}$ $\lvert x\rvert<2^{95}$	A F I A F E
Nearest integer	$y=[x+.5]$ if $x\geq 0$ $y=[x-.5]$ if $x<0$	NINT I IDNINT I	1 1	R D	$\lvert x\rvert<2^{46}$	A F I A F E
Absolute value	$y=\lvert x\rvert$ $y=\lvert(x_r{}^2+x_i{}^2)^{1/2}\rvert$	ABS R IABS I DABS D CABS†† R	1 1 1 1	R I D C	$\lvert x\rvert<$infinity	A F I A F I A F I A F E
Divide for remainder of x_1/x_2	$y=x_1-x_2[x_1/x_2]$	MOD I AMOD R DMOD D	2 2 2	I R D	†††	A P E A F I A F E
Transfer sign	$y=\lvert x_1\rvert$ if $x_2\geq 0$ or $y=-\lvert x_1\rvert$ if $x_2<0$	SIGN R ISIGN I DSIGN D	2 2 2	R I D	$\lvert x\rvert<$infinity	A F I A F I A F I

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† $x=x_r+i\cdot x_i$

††† Argument ranges for MOD functions:

MOD:	$\lvert x_1\rvert<2^{63}$	$0<\lvert x_2\rvert<2^{63}$	$2^{-63}<\lvert x_1/x_2\rvert<2^{63}$
AMOD:	$\lvert x_1\rvert<2^{47}$	$0<\lvert x_2\rvert<2^{47}$	$2^{-47}<\lvert x_1/x_2\rvert<2^{47}$
DMOD:	$\lvert x_1\rvert<2^{95}$	$0<\lvert x_2\rvert<2^{95}$	$2^{-95}<\lvert x_1/x_2\rvert<2^{95}$

Table B-1. General Arithmetic Functions (continued)

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Positive difference	$y=x_1-x_2$ if $x_1>x_2$ $y=0$ if $x_1 \le x_2$	DIM R IDIM I DDIM D	2 2 2	R I D	|x|<infinity	A F I A F I A F E
Double-precision product	$y=x_1*x_2$	DPROD D	2	R	|x|<infinity	A F I
Imaginary portion of complex value	$y=x_i$	AIMAG†† R	1	C	|x|<infinity	A F I
Conjugate of complex value	$y=x_r-i\cdot x_i$	CONJG†† C	1	C	|x|<infinity	A F I
Obtain random number	y=r where r is the first or next in a series of random numbers ($0<y<1$)	RANF R	0			C F E
Obtain rndm seed	The current random number seed	RANGET I	1	I	|x|<infinity	C N E
Establish rndm seed	y=x	RANSET R	1	B	|x|<infinity	C N E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† $x=x_r+i\cdot x_i$

Table B-2. Trigonometric Functions (Angles in Radians)

Purpose	Definition	Function Name,Type	Argument(s)			Codes
			No.	Type	Range	†
Sine	$y=\sin(x)$	SIN R	1	R	$\lvert x\rvert<2^{24}$	A F E
		DSIN D	1	D	$\lvert x\rvert<2^{48}$	A F E
		CSIN†† C	1	C	$\lvert x_r\rvert<2^{24}$,	A F E
					$\lvert x_i\rvert<2^{13}\cdot\ln2$	
Cosine	$y=\cos(x)$	COS R	1	R	$\lvert x\rvert<2^{24}$	F E
		DCOS D	1	D	$\lvert x\rvert<2^{48}$	F E
		CCOS†† C	1	C	$\lvert x_r\rvert<2^{24}$,	F E
					$\lvert x_i\rvert<2^{13}\cdot\ln2$	
Tangent	$y=\tan(x)$	TAN R	1	R	$\lvert x\rvert<2^{24}$	A F E
		DTAN D	1	D		A F E
Cotangent	$y=\cot(x)$	COT R	1	R	$\lvert x\rvert<2^{24}$	C F E
		DCOT D	1	D		C F E
Arcsine	$y=\arcsin(x)$	ASIN R	1	R	$\lvert x\rvert\leq1$	A F E
		DASIN D	1	D		A F E
Arccosine	$y=\arccos(x)$	ACOS R	1	R	$\lvert x\rvert\leq1$	A F E
		DACOS D	1	D		A F E
Arctangent	$y=\arctan(x)$	ATAN R	1			A F E
		DATAN D	1	D	$\lvert x\rvert<$infinity	A F E
	$y=\arctan(x_1/x_2)$	ATAN2 R	2	R	$\lvert x_1\rvert<$infin.,	A F E
		DATAN2 D	1	D	$\lvert x_2\rvert\neq0$	A F E
Hyperbolic sine	$y=\sinh(x)$	SINH R	1	R	$\lvert x\rvert<2^{13}\cdot\ln2$	A F E
		DSINH D	1	D		A F E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† $x=x_r+i\cdot x_i$

Table B-2. Trigonometric Functions (Angles in Radians) (continued)

Purpose	Definition	Function Name,Type	Argument(s) No.	Type	Range	Codes †
Hyperbolic cosine	y=cosh(x)	COSH R	1	R	$\|x\|<2^{13}\cdot\ln 2$	A F E
		DCOSH D	1	D		A F E
Hyperbolic tangent	y=tanh(x)	TANH R	1	R	$\|x\|<2^{13}\cdot\ln 2$	A F E
		DTANH D	1	D		A F E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-3. Exponential Functions

Purpose	Definition	Function Name,Type	Argument(s) No.	Type	Range	Codes †
Square root	$y=x^{1/2}$	SQRT R	1	R	$0\le x\le\text{infinity}$	A F E
		DSQRT D	1	D		A F E
		CSQRT†† C	1	C	$x_r\ge 0$, $x_i<\text{infinity}$	A F E
Exponent	$y=e^x$	EXP R	1	R	$\|x\|<2^{13}\cdot\ln 2$	A F E
		DEXP D	1	D		A F E
		CEXP†† C	1	C	$\|x_r\|<2^{13}\cdot\ln 2$ $\|x_i\|<2^{24}$	A F E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† $x=x_r+i\cdot x_i$

Table B-4. Logarithmic Functions

		Function	Argument(s)			Codes
Purpose	Definition	Name,Type	No.	Type	Range	†
Natural	*Generic is LOG*	ALOG R	1	R		A F E
logarithm	y=ln(x)	DLOG D	1	D	0<x<infinity	A F E
		CLOG†† C	1	C		A F E
Common	*Generic is LOG10*	ALOG10 R	1	R		A F E
logarithm	y=log(x)	DLOG10 D	1	D	0<x<infinity	A F E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† $x=x_r+i\cdot x_i$

Table B-5. Maximum/Minimum Functions

		Function	Argument(s)			Codes
Purpose	Definition	Name,Type	No.	Type	Range	†
Select	*Generic is MAX*	MAX0 I		I		A F I
maximum		AMAX1 R		R		A F I
value	y=The largest of all x	DMAX1 D	$64>n\geq2$	D	|x|<infinity	A F I
		AMAX0 R		I		A F I
		MAX1 I		R		A F I
Select	*Generic is MIN*	MIN0 I		I		A F I
minimum		AMIN1 R		R		A F I
value	y=The smallest of all x	DMIN1 D	$64>n\geq2$	D	|x|<infinity	A F I
		AMIN0 R		I		A F I
		MIN1 I		R		A F I

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-6. Character Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Lex .GE.	$y=a_1\geq a_2$ in collating seq	LGE L	2	CH	Any string	A N E
Lex .GT.	$y=a_1>a_2$ in collating seq	LGT L	2	CH	Any string	A N E
Lex .LE.	$y=a_1\leq a_2$ in collating seq	LLE L	2	CH	Any string	A N E
Lex .LT.	$y=a_1\leq a_2$ in collating seq	LLT L	2	CH	Any string	A N E
Length of string	Number of characters in a character entity	LEN I	1	CH	Any string	A N I
Index of a substrng	Returns the position of a character within a string	INDEX I	2	CH	Any string	A N E

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-7. Type Conversion Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Conversion to integer	Truncation toward zero (fraction lost)	INT I	1	C	$\|x_r\|<2^{46}$	A F I
		I	1	I	$\|x\|<2^{63}$	
		I	1	R	$\|x\|<2^{46}$	
		I	1	I††	$\|x\|<2^{23}$	
		I	1	B†††	$\|x\|<2^{63}$	
		IFIX I	1	R,B†††	$\|x\|<2^{46}$	A F I
		IDINT I	1	D		A F I

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† 24-bit integer

††† Type conversion routines assign appropriate types to Boolean arguments without shifting or manipulating the bit patterns they represent.

Table B-7. Type Conversion Functions (continued)

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Conversion to real	$y=x_r$ ¶	REAL R	1	C	$\|x_r\|<2^{46}$	A F I
		R	1	R	$\|x\|<$infinity	
	$y=x$	R	1	I	$\|x\|<2^{46}$	
		I	1	I††	$\|x\|<2^{23}$	
		FLOAT R	1	B†††	$\|x\|<$infinity	A F I
	$y=x$ Accuracy may be lost	SNGL R	1	D	$\|x\|<$infinity	A F I
Conversion to double-precision	$y=x_r$ ¶ Accuracy may be lost.	DBLE D	1	C	$\|x_r\|<$infin.	A F I
		D	1	I	$\|x\|<2^{46}$	
		I	1	I††	$\|x\|<2^{23}$	
	$y=x$ Accuracy may be extended.	D	1	R		
		D	1	D	$\|x\|<$infinity	
		D	1	B†††		
Conversion to complex	$y=x$	CMPLX C	1	C	$\|x\|<$infinity	A F I
		C	1,2	I	$\|x\|<2^{46}$	
		I	1	I††	$\|x\|<2^{23}$	
	$y=x_1 + i\cdot x_2$	C	1,2	B†††		
	or	C	1,2	R	$\|x\|<$infinity	
	$y=x_1 + i\cdot 0$	C	1,2	D		
Character to integer	ICHAR(x) returns position of character x in collating sequence	ICHAR I	1	CH	any legal character	A N E
Integer to character	CHAR(x) returns the xth character in collating sequence	CHAR CH	1	I	0-255	A N I
		CH	1	B		

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

†† 24-bit integer

††† Type conversion routines assign appropriate types to Boolean arguments without shifting or manipulating the bit patterns they represent.

¶ $x=x_r+i\cdot x_i$

Table B-8. Boolean Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes †
			No.	Type	Range	
Logical product	Bit-by-bit logical product (AND) of x_1 and x_2	AND B	2	B,I, R,P		C F I
		L	2	L		
Logical sum	Bit-by-bit logical sum (OR) of x_1 and x_2	OR B	2	B,I, R,P		C F I
		L	2	L		
Logical difference (not equiv.)	Bit-by-bit logical difference (XOR or NEQV) of x_1 and x_2	XOR, NEQV B	2	B,I, R,P		C F I
		L	2	L		
Equivalence	Bit-by-bit equivalence (XOR) of x_1 and x_2	EQV B	2	B,I, P		C F I
		L	2	L		
Complement	Bit-by-bit logical complement of x_1	COMPL B	1	B,I, R,P		C F I
		L	2	L		
Mask	x=number of one-bits to be left-justified if $0 \le x \le 63$. (128-x)=number of 1-bits to be right-justified if $64 \le x \le 128$.	MASK B	1	I		C F I

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-8. Boolean Functions (continued)

Purpose	Definition	Function	Argument(s)			Codes
		Name,Type	No.	Type	Range	†
Circular shift	x_1 shifts left x_2 positions; leftmost positions replace vacated positions.	SHIFT B	2	x_1: B,I, R,L x_2:I	$0 \leq x_2 < 64$	C F I
Logical shift	x_1 shifts left x_2 positions; leftmost positions lost; rightmost positions set to zero.	SHIFTL B	2	x_1: B,I, R,P x_2:I	$0 \leq x_2 < 64$	C F I
	x_1 shifts right x_2 positions; rightmost positions lost; leftmost positions set to zero.	SHIFTR B	2	x_1: B,I, R,P x_2:I		C F I
Leading zeros	Tallies number of leading zeros in x	LEADZ I	1	B,I, R,L		C F I
Population count	Tallies number of ones in x	POPCNT I	1	B,I, R,L		C F I
Population parity count	0, if x has an even number of ones; 1, if x has odd number of ones.	POPARR I	1	B,I, R,L		C F I
Cray Scalar Merge	Bit-by-bit selective merge: (x_1 AND x_3) OR (x_2 AND NOT x_3). See the end of this section.	CSMG B	3	B,I, R,L		C F I

† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-9. Time and Date Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes [†]
			No.	Type	Range	
Real-time clock	Low order 46 bits of clock register expressed as floating point integer	RTC R	0			C N I
	Current clock register content	IRTC I	0			C N I
Time	Current time in ASCII code (hh:mm:ss)	CLOCK[††] B	0			C N E
Julian date	Current Julian date in ASCII code (yyddd)	JDATE[††] B	0			C N E
Date	Current date in ASCII code (mm/dd/yy)	DATE[††] B	0			C N E

[†] A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

[††] These procedures can be called as subroutines also. An integer or real argument is passed to return the result.

Table B-10. Miscellaneous Functions

Purpose	Definition	Function Name,Type	Argument(s)			Codes [†]
			No.	Type	Range	
Location	Returns memory address of specified variable or array	LOC I	1	B,I, R,L, C,D, P		
Number of arguments	Tallies number of arguments used to call a subprogram	NUMARG I	0			C N I

[†] A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

Table B-11. Vectorization Functions†

Purpose	Definition	Function	Argument(s)			Codes
		Name,Type	No.	Type	Range	††
Vectorization	x_1 returned if $x_3 \geq 0$ x_2 returned if $x_3 < 0$	CVMGP B	3	B,I, R,P		C F I
		L	3	L		
	x_1 returned if $x_3 < 0$ x_2 returned if $x_3 \geq 0$	CVMGM B	3	B,I, R,P		C F I
		L	3	L		
	x_1 returned if $x_3 = 0$ x_2 returned if $x_3 \neq 0$	CVMGZ B	3	B,I, R,P		C F I
		L	3	L		
	x_1 returned if $x_3 \neq 0$ x_2 returned if $x_3 = 0$	CVMGN B	3	B,I, R,P		C F I
		L	3	L		
	x_1 returned if x_3 is true x_2 returned if x_3 is false	CVMGT B	3	x_1,x_2: B,I, R,P x_3:L		C F I
		L	3	L		

† These functions cannot be passed as arguments.

†† A=ANSI standard, C=Cray extension; F=full, P=pseudo, N=no vectorizing; E=external code, I=in-line code.

B.1 CSMG (CRAY SCALAR MERGE) FUNCTION

CSMG(x1,x2,x3) = (x1 .AND. x3) .OR. (x2 .AND. .NOT.x3)

CSMG merges x1 and x2, controlled by the bit mask in x3. When a 1 bit appears in x3, the corresponding bit of x1 is taken, and when a 0 bit appears in x3, the corresponding bit of x2 is taken.

CSMG compiles to very efficient in-line code because the Cray hardware includes a scalar merge instruction, which CFT generates for CSMG. Because the uses of CSMG are generally machine-dependent, it should not normally be used where portability is important.

Some typical uses of CSMG follow.

To select one of the arguments:

```
              For any x1 & x2:  CSMG(x1,x2,MASK(64)) = CSMG(x1,x2,-1) = x1.


    CSMG (x1,x2,0) = x2.
```

To replace bit-fields:

```
              CSMG( 'ABCDEFGH'H, '12345678'H, X'0000FFFFFF0000FF' ) =
'12CDE67H'H

              INTEGER EXPONENT, EXPMASK
              PARAMETER( EXPMASK = X'7FFF000000000000' )
              EXPONENT(X) = SHIFTR( X.AND.EXPMASK, 48 )  ! Statement function
              DIVBY2(X) =
    ! Statement function
      &   CSMG(  SHIFTL(EXPONENT(X)-1,48),
      &         X,
      &         EXPMASK )

    Then, for example, DIVBY2(-28.0) = -14.0.
```

C. CRAY FORTRAN UTILITY PROCEDURES

The Cray FORTRAN Compiler (CFT) includes a set of utility procedures which, like intrinsic functions, are predefined by name and function. Unlike intrinsic functions, however, utility procedures are not provided for by the ANSI FORTRAN Standard. They include subroutines as well as functions; some have arguments of mixed type; and some modify these arguments' contents.

The CFT utility procedures generate calls to operating system routines. All of the utility procedures are implemented under COS. Some of the procedures are not implemented under UNICOS; these procedures are so indicated in table C-1.

Table C-1 describes the utility procedures. Data types are abbreviated as follows: integer (I), real (R), double precision (D), complex (C), logical (L), Boolean (B), and Hollerith (H). Unless noted, 24-bit integer variables can be used as arguments to a function accepting integer arguments. 24-bit variables are sign extended and treated as 64-bit variables.

NOTE

Although correct argument types are specified, it is the user's responsibility to ensure that actual arguments are of the correct type. No type conversion occurs automatically.

Table C-1. CFT Utilities

Function or Subrtn. Name	Type (fnct only)	Description	Arg. Type
†SYSTEM	I	Selects a COS function. See the description of the Exchange Processor in the COS EXEC/STP/CSP Internal Reference Manual, publication SM-0040	I
†EXIT		Terminates job step. Job continues at next control statement.	
SSWITCH		x_2 = 1 if sense switch x_1 is ON. x_2 = 2 if sense switch x_1 is OFF, if $x_1 < 1$, or if $x_1 > 6$.	I
ABORT		Terminates job step after program detects error. Gives error exit to COS. Job continues at next EXIT statement. Prints traceback in logfile. Argument x_1 is optional; if present, it is written to the logfile before the traceback, and is subject to same restrictions as in REMARK2.	B,I,D,C,L
ERREXIT		Terminates job step after program detects error. Gives error exit to COS. Job continues at next EXIT control statement. Prints no traceback.	
TRBK	I	Writes a traceback through subroutine calls to file x_1. If x_1 is '$LOG' or is missing, the traceback is written to the logfile.	I
REMARK		Writes an ASCII message to both the user logfile and system logfile. The R descriptor is prohibited. For a variable or array name of type I, R, C, or D (not 24-bit integer), the caller must signal the end of the message by a null character (0). x_1 = 8 words maximum.	B,I,D,C,L
REMARK2		Same as REMARK except that x_1 may have a maximum of 9 words. The first 5 characters are a code identifying the message for machine processing.	B,I,D,C,L
REMARKF		Enters a message in the logfile. A format and up to 12 variables can be passed, each occupying only 1 word. The first argument is the format label.	x_1: I,C x_2:I,R,L,B

† This procedure is not available under UNICOS.

Table C-1. CFT Utilities (continued)

Function or Subrtn. Name	Type (fnct only)	Description	Arg. Type

SENSEFI		Determines current interrupt mode.
		Mode=1, interrupts permitted;
		Mode=0, interrupts prohibited.
			I

CLEARFI		Temporarily prohibits floating-point interrupts	
SETFI		Temporarily permits floating-point interrupts	
CLEARFIS		Permanently prohibits floating-point interrupts	
SETFIS		Permanently permits floating-point interrupts	
†EODW		Writes EOD and, as required, EOF and EOR record (s)on unit x_1	I††
†IEOF	I	Returns -1 if EOD processed at unit x_1; +1 if EOF read at unit x_1; 0, otherwise.	I††
†EOF	R	Returns -1.0 if EOD processed at unit x_1; +1.0 if EOF read at unit x_1; 0.0, otherwise.	I††
†UNIT	R	Returns -2.0 if record at unit x_1 partly read; -1.0 if unit x_1 transfer successful; 0.0 if EOF or EOD read at unit x_1; +1.0 if disk parity error while reading unit x_1; +2.0 if unit x_1 error indicated. Applies only to buffered input/output operations.	I††
†LENGTH	I	Returns number of Cray words transferred to or from unit x_1. Returns 0 if EOF or EOD read from unit x_1.Applies only to buffered input/output operations.	I††

† This procedure is not available under UNICOS.

†† x_1 can be a unit number or a Hollerith unit name. If no argument is specified for DUMPJOB, $DUMP is used.

Table C-1. CFT Utilities (continued)

Function or Subrtn. Name	Type (fnct only)	Description	Arg. Type
†GETPOS	B	Returns starting address of current record in the dataset associated with unit x_1	I††
†SETPOS		Sets starting address of current record in dataset associated with unit x_1 to beginning address of dataset if x_2 = 0, to ending address if x_2 = -1, or to word address x_2.	x_1:I†† x_2:I
TIMEF	R	Returns current clock register content in milliseconds	
SECOND	R	x_1 = Cumulative CPU time for job in seconds.†††	R
†ENDRPV		Resumes error exit processing suspended by SETRPV	
†SETRPV		Transfers control to a specified routine upon encountering a user-selected system error. x_1 = Name of subroutine to which control is passed x_2 = Array to receive the Exchange Package and abort conditions x_3 = Mask defining the error class from which reprieve is desired	x_1:H x_2: I,R,L x_3: I,B
†SYMDEBUG		Dumps the contents of specified program variables. The character string argument is identical to the parameters for the DEBUG control statement. See COS Version 1 Reference Manual, publication SR-0011	H
†DUMPJOB		Creates an unblocked dataset containing the user job area image (including register states) suitable as input to the DUMP programs	I††

† This procedure is not available under UNICOS.

†† x_1 can be a unit number or a Hollerith unit name. If no argument is specified for DUMPJOB, $DUMP is used.

††† This procedure can be used either as a function or as a subroutine, but not both within the same program unit.

D. CFT MESSAGES

The Cray FORTRAN Compiler (CFT) produces seven types of compile-time messages. From least severe to most severe, they are

NOVECTOR - Message describes why a DO loop did not vectorize. NOVECTOR messages are issued only when the LOOPMARK utility is enabled on the CFT compiler call. The NOVECTOR messages are described following the logfile message descriptions in this appendix.

COMMENT - Message reports a programming inefficiency.

NOTE - This usage may cause problems with other compilers.

CAUTION - Message reports a possible programmer error.

WARNING - Message reports a probable programmer error.

NON-ANSI - Message reports when the ANSI parameter is specified on the CFT control statement. Specifying ANSI issues nonstandard code messages as NON-ANSI instead of NOTE, CAUTION, or WARNING. When ANSI is not specified on the CFT control statement (the default), the message is issued as a NOTE, CAUTION, or WARNING.

ERROR - Message reports a fatal error.

CFT produces these messages in the following general format.

```
severity type          message          location number
```

(Location number is the internal CFT location where the message was generated.)

If a message in this appendix is preceded by †, the print position at which the error occurred is specified by the following.

```
NEAR >>      <<
```

Error messages can occur under the following conditions.

- After the first nondeclarative statement - An error message occurring after the first nondeclarative statement usually indicates contradictions between declarative statements.

- After an END statement - An error message occurring after an END statement usually refers to missing statement or format numbers.

- <u>After any other statement</u> - An error message occurring after any other statement usually refers to a syntax error in that statement. If no source list is being generated, the first line of the statement is listed. Continuation lines, if any, are not listed but are indicated by four plus signs appended to the first line of the statement.

Logfile messages follow the compile-time messages in this appendix.

D.1 <u>COMPILE-TIME MESSAGES</u>

"*name*" ALREADY SAVED AT SEQUENCE NUMBER *n*
"*name*" is in a SAVE statement list and the preceding SAVE statement list at sequence number *n*.

"*name*" APPEARS TWICE IN DUMMY ARGUMENT LIST
A symbolic name appears twice in the dummy argument list of a FUNCTION or SUBROUTINE statement.

"*name*" CANNOT BE DECLARED EXTERNAL
"*name*" has appeared in both an EXTERNAL statement and in an array declarator, DATA, COMMON, or PARAMETER statement.

"*name*" CANNOT BE DECLARED INTRINSIC
The symbolic name was used in the program unit, giving the name a type; for example, a dummy argument.

"*name*" DOUBLY ASSOCIATED IN EQUIVALENCE AT SEQUENCE "*number*"
EQUIVALENCE statements using "*name*" are incorrect and specify an illegal storage sequence. To correct the error, the following two situations must be true. The same storage unit cannot occur more than once in a storage sequence. The example

```
REAL A(2)
EQUIVALENCE (X,A(1)), (X,A(2))
```

is incorrect because A(1) and A(2) have the same storage unit. Consecutive storage units cannot be specified as nonconsecutive. The example

```
REAL A(2)
DOUBLE D(2)
EQUIVALENCE (A(1),D(1)), (A(2),D(2))
```

is incorrect because A(1) and A(2) are nonconsecutive.

"*name*" IS STATICALLY ALLOCATED
CFT assigns "*name*" to static storage; otherwise, the name can be removed from SAVE and DATA statements.

"*name*" NO LONGER INTRINSIC

An intrinsic function name appeared in a compiler directive, giving external subprograms special attributes (for example, VFUNCTION and NO SIDE EFFECTS). The intrinsic function loses its intrinsic properties. The intrinsic function should appear in an EXTERNAL statement before the compiler directive declarations.

"*name*" NOT AN INTRINSIC FUNCTION

A symbolic name in the list of an INTRINSIC statement is not an intrinsic function.

"*name*" NOT LOCAL OR COMMON BLOCK VARIABLE

List may contain only local or common block variables.

"*name*" PREVIOUSLY DECLARED AS AN ARRAY

The symbol *name* has already been used in the program as an array name.

"*name*" STATEMENT IS A NONSTANDARD STATEMENT

The statement type indicated is a CFT extension to the FORTRAN language specified by the ANSI FORTRAN Standard.

†"*name*" UNEXPECTED CHARACTER IN FORMAT

When a format specifier list was parsed, an unknown edit descriptor or premature end of an edit descriptor was found. CFT attempts to recover at the next character following a comma or at the next parenthesis or string delimiter.

"*name*" USED AS SYMBOLIC CONSTANT AND AS COMMON BLOCK NAME

The same identifier was used for a common block name and a symbolic constant name. The ANSI FORTRAN Standard prohibits this duplicate usage within the same program unit.

†ABBREVIATION OF "*name*" IS NONSTANDARD

The logical operator or constant indicated by "*name*" was abbreviated. The ANSI FORTRAN Standard prohibits the abbreviation of logical constants or operators.

ADJUSTABLE DIMENSION ILLEGAL IN MAIN PROGRAM

Arrays in main program must have constant subscripts.

ALTERNATE RETURNS ARE NOT ALLOWED IN FUNCTIONS

An alternate return was encountered in a function.

AMBIGUOUS CHARACTER EXPRESSION

A character expression of the form *v*=*e* is illegal if the same character position is referenced by *v* and *e*. A character expression is ambiguous if subscripts or substrings are defined with variables. For example, A(I:J)=A(K:L) is ambiguous to the compiler. The expression must be legal or unexpected results may occur.

ARGUMENTS IGNORED FOR ZERO ARGUMENT INTRINSIC

An intrinsic function defined to have zero arguments was referenced with one or more arguments. Reference zero argument intrinsics with a null argument list.

ARITHMETIC EXPRESSION WITH DOUBLE PRECISION AND COMPLEX IS NONSTANDARD

An arithmetic or relational expression was used with a complex and double-precision operand. CFT will convert the double-precision operand to a complex operand. The ANSI FORTRAN Standard prohibits these conversions.

ARITHMETIC EXPRESSIONS WITH BOOLEAN OPERANDS ARE EVALUATED LEFT TO RIGHT
RESULTS MAY DIFFER FROM PRE-1.15 CFT RESULTS

Arithmetic expressions with Boolean operands are evaluated so that the result is the same as if the expression were evaluated from left to right within precedence levels. This is not necessarily the case in CFT versions prior to 1.15.

ARITHMETIC WITH TWO BOOLEAN OPERANDS PRODUCES BOOLEAN RESULTS
RESULTS MAY DIFFER FROM PRE-1.15 CFT RESULTS

In CFT versions prior to 1.15, arithmetic with two Boolean operands produced integer results.

ARRAY NAMED FORMAT IS POTENTIALLY AMBIGUOUS

An array with the symbolic name 'FORMAT' was declared using the DIMENSION, COMMON, INTEGER, REAL, DOUBLE, or COMPLEX statement.

ASSUMED SIZE DIMENSION ILLEGAL IN MAIN PROGRAM

Arrays in main program must have constant subscripts.

AT SEQUENCE NUMBER *n*: "*name*" CANNOT BE SAVED

"*name*" is in a SAVE statement list at sequence number *n*, but "*name*" cannot appear in a SAVE statement list.

†BAD CONSTANT LIST IN DATA STATEMENT

A DATA statement is missing a constant list or contains an illegal constant or separator character.

BAD HEXADECIMAL CONSTANT

Illegal separator between hexadecimal digits or the length of the constant is greater than 16 characters.

BAD REPETITION FIELD

The repetition count in a DATA constant list must be an integer greater than 0.

†BAD STATEMENT FUNCTION PARAMETER LIST

The formal parameter list in an arithmetic statement function definition statement contains an illegal element.

BAD SUBSCRIPT IN DATA STATEMENT

A subscript must be an integer constant or constant name in a DATA statement.

BAD TRIP COUNT IN IMPLIED DO

Incrementation parameter m_3 has been assigned a value of 0 or $(m_2 - m_1+m_3)/m_3$ is negative or 0.

BLOCK IF STATEMENTS NESTED TOO DEEPLY

The nesting of block IF statements exceeds 511. CFT does not allow more than 511 nested block IF statements.

BRANCH INTO IF, ELSE, OR ELSE IF BLOCK; LABEL "*l*"

A branch into an IF-block, ELSE-block, or ELSE IF-block to label "*l*" from outside the block has been detected. The ANSI FORTRAN Standard prohibits the transfer of control into an IF-block, ELSE-block, or ELSE IF-block.

BUFFERED IO IS NONSTANDARD

A BUFFER IN or BUFFER OUT statement was used. Buffered input and output are CFT extensions to the ANSI FORTRAN Standard.

CALL OF NON EXTERNAL FUNCTION "*name*"

Called external procedure does not exist or a name has been used for both a variable and an external procedure.

†CHARACTER COUNT TOO LARGE

An R-form Hollerith constant is specified with more than eight characters or an H- or L-form Hollerith constant is specified with more than eight characters in other than a DATA statement or an actual argument list.

CHARACTER LENGTH MUST BE >ZERO and <16384

A character entity must be assigned a length greater than 0 and less than 16,384.

COMBINATION OF CODE AND DATA EXCEEDED MAXIMUM SIZE

The combination of code and data has exceeded 16 million words if Extended Memory Addressing (EMA) was used, or 4 million words if EMA was not used.

COMMA EXPECTED

A required comma has not been specified.

COMMA EXPECTED IN EQUIVALENCE AT SEQUENCE "*number*"

A required comma has been omitted in an EQUIVALENCE statement.

†COMMA OR RIGHT PARENTHESIS EXPECTED

A required comma or right parenthesis was omitted.

COMMA OR RIGHT PARENTHESIS EXPECTED IN EQUIVALENCE AT SEQUENCE "*number*"
A required comma or right parenthesis was omitted in an EQUIVALENCE statement.

†COMMA REQUIRED BY STANDARD BETWEEN FORMAT FIELDS
The ANSI FORTRAN Standard requires a comma between most format edit descriptor fields, while most commas in CFT format specifier lists are optional.

COMMON BLOCK "*name*" IS VERY LARGE; USE EXTENDED MEMORY ADDRESSING
A named or blank common block was declared with more than 4 million words of storage.

COMMON BLOCK NAME "*name*" MORE THAN 6 CHARACTERS
The common block name has seven or eight characters. ANSI FORTRAN Standard common block names cannot have more than six characters.

COMPILER ERROR
The CFT compiler detected an error in its internal tables; please show your listing to a Cray Research systems analyst.

COMPILER ERROR - INTERNAL TABLE OVERFLOW, RECOMPILE WITH SMALLER BLOCK SIZE
A CFT internal table has overflowed when compiling. The optimization block where the error occurs must be reduced in size by inserting a CDIR$ BLOCK compiler directive in the block or reducing the maximum block size with the MAXBLOCK control statement parameter.

CONFLICTING TYPE FOR INTRINSIC FUNCTION "*name*" IGNORED
A type statement cannot change the type of an intrinsic function. The function must be declared EXTERNAL before its type can be changed.

CONFLICTING USE OF INTRINSIC FUNCTION "*name*"
An intrinsic function name has been used to reference the function in another way within the program unit.

CONSTANT DIMENSION TOO LARGE
All dimensions must be less than 2^{22}.

CONSTANT LIST LONGER THAN VARIABLE LIST
Constants and variables in a DATA statement must have a one-to-one correspondence.

†CONSTANT SUBSCRIPT TOO LARGE
Statement contains a subscript which, when evaluated, yields a result greater than the size of the named array.

CONTROL LIST MUST INCLUDE ONE FILE OR UNIT OPTION
INQUIRE statement must specify either a file or a unit.

CONTROL LIST MUST INCLUDE ONE UNIT OPTION
The I/O statement must specify a unit.

†DATA ENTRY IN BLANK COMMON ILLEGAL
The DATA statement cannot be used to initialize blank common.

DATA INITIALIZATION OF COMMON VARIABLE "*name*" NOT IN BLOCK DATA SUBPROGRAM
"*name*" appeared in a common block and was initialized in a DATA statement in an executable subprogram. The ANSI FORTRAN Standard allows data initialization of common block entities only in a block data subprogram.

DATA STATEMENT PRECEDES SPECIFICATION STATEMENT
The ANSI FORTRAN Standard specifies an order for nonexecutable statements. A DATA statement must appear after every specification statement. CFT detected a DATA statement preceding a specification statement.

DECLARATOR "*name*" MUST BE DUMMY ARGUMENT OR IN COMMON
The dimension declarator "*name*" in an adjustable array declarator was not a dummy argument or a variable in common.

DEPENDENCY INVOLVING ARRAY "*name*"
A dependency exists involving two array references in the same statement. This dependency inhibits DO-loop vectorization.

DEPENDENCY INVOLVING ARRAY "*name*" IN SEQUENCE NUMBER *nnnnnnnn*
A dependency exists involving an array reference in sequence number *nnnnnnnn* and an array reference in the AT SEQUENCE NUMBER *n* line preceding this message. This dependency inhibits DO-loop vectorization.

DIMENSION COUNT > SEVEN
More than seven dimensions appear in an array declarator.

DIMENSION EXCEEDED
A subscript in a DATA statement element exceeds the corresponding array declaration.

DIRECTIVE NO LONGER SUPPORTED
The SCHED and NOSCH directives are no longer supported. The directives have no effect if used.

DIVIDE BY ZERO
Dividing by the constant 0 is illegal.

DO ILLEGAL ON CONDITIONAL STATEMENT
This type of statement is not allowed as the conditional statement of a direct logical IF statement.

DO INDEX ACTIVE
The loop control variable is already active from a previous loop.

DO INDEX IN INPUT LIST
Attempt to read data into a DO variable.

DO LOOP MAY NOT CROSS BLOCK BOUNDARY
A DO loop that begins within an IF-block, ELSE-block or ELSE IF-block must be totally contained within that block. A block that begins within a DO loop must be totally contained within the loop.

DO TERMINATOR ILLEGAL IN CONDITIONAL BLOCK STATEMENT
DO-loop must not terminate on an IF (*e*)THEN, ELSE, ELSE IF, or ENDIF statement.

DO TERMINATOR PRECEDES DO STATEMENT
The statement label that terminates a DO-loop precedes the corresponding DO statement.

†DOUBLE PRECISION CONSTANT IN COMPLEX CONSTANT IS NONSTANDARD
A double-precision constant is used to form a complex constant. CFT allows this when double precision is disabled (OFF=P option in the CFT control statement), but it is prohibited by the ANSI FORTRAN Standard.

DOUBLY DEFINED FUNCTION OR MISSING DIMENSION
An arithmetic statement function is defined more than once or an array was not dimensioned.

DOUBLY DEFINED STATEMENT NUMBER
Statement labels cannot be defined more than once in a program unit.

DUMMY ARGUMENT IN EXECUTABLE STATEMENT PREVIOUS TO ENTRY
A dummy argument name in an executable statement must also be specified in the FUNCTION, SUBROUTINE, or ENTRY statement referenced before the executable statement.

DUPLICATE COMMON DEFINITION "*name*"
Variable or array appears in common more than once.

DUPLICATE CONTROL OPTION IN LIST
An option is specified more than once in an I/O statement control list.

DUPLICATE DIMENSION "*name*"
Dimensions cannot be declared more than once.

DUPLICATE TYPE DEFINITION "*name*"
Variables cannot be given more than one type. This message is issued when a type statement redefines the variable type with a type established by a DATA, POINTER, or previous type statement.

DYNAMIC BLOCK "name" NOT IN PREVIOUS COMMON

Dynamic name must be declared as a common block previous to its appearance in a DYNAMIC compiler directive.

†EBCDIC NOT IMPLEMENTED

The current version of CFT allows only ASCII characters.

EMBEDDED COMMENTS ARE NONSTANDARD

A comment is embedded in a line of source code following an exclamation point. Embedded comments are a CFT extension to the ANSI FORTRAN Standard.

†EMPTY PARENTHESES ILLEGAL IN FORMAT

CFT found an empty set of parentheses nested in a format specifier list. Only the outermost set of parentheses can be empty.

ENCODE/DECODE MAY NOT BE LIST DIRECTED

The format identifier in an ENCODE or DECODE statement must not specify list-directed I/O; it cannot be an *.

ENTRY NAME ILLEGAL

ENTRY name not a function or subroutine name.

ENTRY "*name*" USED AS DUMMY ARGUMENT

"*name*" is an entry point (that is, "*name*" appeared in a FUNCTION, SUBROUTINE, ENTRY, or BLOCK DATA statement) and appears as a dummy argument.

ENTRY STATEMENT ILLEGAL IN DO LOOP OR BLOCK IF

The ENTRY statement must not be used in a DO loop or a block IF.

ENTRY STATEMENT ILLEGAL IN MAIN PROGRAM

The ENTRY statement must not be used in a main program. It is used only in a subroutine or function.

†EQUIVALENCE EXTENDS COMMON BLOCK BASE

Common block storage is illegally extended by adding storage units preceding the first storage unit specified in the COMMON statement.

EQUIVALENCE OF "*name*" IN DIFFERENT COMMON BLOCKS

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit.

ERROR IN CONSTANT

Illegal characters in constant, or constant out of range

EXECUTABLE CODE IN BLOCK DATA SUBPROGRAM

Executable statements appear in a block data subprogram. This is prohibited by the ANSI FORTRAN Standard.

†EXPRESSION ILLEGAL IN INPUT LIST

Input list item is not a variable name, array element name, or array name.

EXPRESSION TYPE MUST BE INTEGER

Expressions in alternate RETURN statement must be type integer.

EXTENDED RANGE DO-LOOP IS NONSTANDARD

CFT detects a branch into the range of a DO loop or a possible branch using an ASSIGN, END=, or ERR= branch to a label defined in the range of a DO-loop. Extended range DO loops are a CFT extension to the ANSI FORTRAN Standard.

†EXTRA CHARACTERS AFTER END OF STATEMENT

Characters are specified after the syntactic end of a statement.

†EXTRA CHARACTERS AFTER END OF STATEMENT IN EQUIVALENCE AT SEQUENCE "*number*"

Characters are specified after the syntactic end of an EQUIVALENCE statement.

†EXTRA COMMA OR MISSING PARAMETER

Either the statement contains an extra comma or a parameter or list item has been omitted.

†FEWER SUBSCRIPTS USED THAN DECLARED

A reference to an actual array element has fewer subscript expressions in its subscript than dimension declarators in the corresponding array declarator. The missing subscript expressions are assumed rightmost in the subscript and are each assigned the value 1 by the compiler.

†FEWER SUBSCRIPTS USED THAN DECLARED IN EQUIVALENCE AT SEQUENCE "*number*"

A reference to an actual array element in an EQUIVALENCE statement has fewer subscript expressions in its subscript than dimension declarators in the corresponding array declarator. The missing subscript expressions are assumed rightmost in the subscript and are each assigned the value 1 by the compiler.

†FIELD WIDTH MUST NOT BE ZERO

The field width following a format edit descriptor is zero, for example, F0.2 or G20.8E0.

†FIELD WIDTH VALUE TOO SMALL

The field width value of a format edit descriptor is too small to print as specified, for example, F3.5.

FORMAT MUST BE CHARACTER EXPRESSION

A FORMAT specifier can be an expression only if the expression is a character expression.

FUNCTION "*name*" ALREADY DECLARED EXTERNAL
The symbolic name appearing in an INTRINSIC statement has already appeared in an EXTERNAL statement.

FUNCTION "*name*" ALREADY DECLARED INTRINSIC
The symbolic name appearing in an INTRINSIC statement has already appeared in an INTRINSIC statement.

FUNCTION "*name*" MORE THAN 6 CHARACTERS
An attempt was made to declare a function with a name greater than six characters as having a vector call-by-value version (CDIR$ VFUNCTION). Rename the function with a shorter name.

FUNCTION "*name*" MUST BE DECLARED IN INTRINSIC OR EXTERNAL STATEMENT
A function passed to another subprogram as an actual argument must be declared in an INTRINSIC statement (intrinsic functions) or an EXTERNAL statement (user-supplied functions).

FUNCTION "*name*" NOT DECLARED
One of the following conditions exists with IMPLICIT NONE or IMPLICIT SKOL specified.

1. Function "*name*" has not appeared in an EXTERNAL statement.
2. "*name*" was intended to be an array but did not appear in an array declarator.

FUNCTION OR CALL "*name*" REFERENCES ITSELF
A reference to the function or subroutine subprogram being compiled is encountered with that subprogram.

†FUNCTION USED WITH INCORRECT NUMBER OF ARGUMENTS
The number of arguments in the function reference does not agree with the number of arguments in the function definition.

GROUP NAME DEFINED PREVIOUS TO NAMELIST
A group name may be defined only in NAMELIST.

†H,L,R COUNT < OR = ZERO
In an *n*H, *n*L, or *n*R specification of a Hollerith value, *n* is less than or equal to 0.

†H,L,R COUNT PAST END OF STATEMENT
In an *n*H, *n*L, or *n*R specification of a Hollerith value, *n* specifies more characters than are provided, or an apostrophe terminating a Hollerith string is missing.

HEXADECIMAL CONSTANT IS NONSTANDARD
The ANSI FORTRAN Standard does not provide for hexadecimal constants.

†HOLLERITH CONSTANT > EIGHT CHARACTERS

A Hollerith constant of more than eight characters is specified in other than H- or L-form and in other than an actual argument list or a DATA statement constant.

†HOLLERITH CONSTANTS ARE NONSTANDARD

A Hollerith constant was used in a statement other than a FORMAT statement. The ANSI FORTRAN Standard only allows Hollerith constants in FORMAT statements.

IDENTIFIER "*name*" MORE THAN 6 CHARACTERS

The identifier contains 7 or 8 characters. The ANSI FORTRAN Standard provides for a maximum of 6 characters in identifier names.

IF BLOCK LEVEL NOT = ZERO AT END STATEMENT

An ENDIF statement is missing.

ILLEGAL ARGUMENT TO TSKSTART

One or more arguments being passed to TSKSTART is inconsistent with the expected arguments. Check the arguments to ensure that the first argument is an integer array with a minimum length of 2. The second argument must be a declared external. The remaining arguments must be local variables or variables declared in common. See the Multitasking User's Guide, CRI publication SN-0222 for more information.

ILLEGAL ARITHMETIC EXPRESSION

An operand in an arithmetic expression is of type logical. This is prohibited by the ANSI FORTRAN Standard.

ILLEGAL BY VALUE CALL

A by-value function call requires more than seven S or V registers to pass its arguments. A by-value call cannot use more than seven registers. Reduce the register number to a number less than eight or pass the arguments by address instead of by value.

†ILLEGAL CHARACTER

A nonstandard FORTRAN character, misplaced character, or syntax error has been encountered.

ILLEGAL CHARACTER EXPRESSION

A character assignment expression (for example, *v*=*e*) is illegal because *v* is used in expression *e*.

†ILLEGAL CHARACTER OR MISSING DIMENSION

Either the statement contains an illegal character or an array element has not been defined by a DIMENSION statement.

†ILLEGAL CHARACTERS IN NAME FIELD

Illegal characters are in a field that must contain a symbolic name.

†ILLEGAL CHARACTERS IN NAME FIELD IN EQUIVALENCE AT SEQUENCE "*number*"
Illegal characters are in an EQUIVALENCE field that must contain a symbolic name.

†ILLEGAL CHARACTERS IN STATEMENT NUMBER FIELD
Non-numeric characters appear in what should be a numeric field.

†ILLEGAL COMMON BLOCK NAME
The specification of a common block name does not conform to the rules for constructing symbolic names.

†ILLEGAL COMPILER DIRECTIVE
CDIR$ omitted or misspelled in columns 1 through 5, 6 not blank or zero, compiler directive not in columns 7 through 72.

ILLEGAL CONDITIONAL STATEMENT
The conditional statement go logical IF must not be a logical IF statement or a block statement IF.

ILLEGAL CONTINUATION
More than 19 consecutive continuation lines encountered or the first line of a program unit is a continuation line.

†ILLEGAL CONTROL OPTION
An option in the control list of an I/O statement is incorrect.

ILLEGAL CONVERSION IN DATA STATEMENT
The types of a variable and an associated constant in a DATA statement differ. The type conversion required is illegal or undefined.

†ILLEGAL DO INDEX
DO variable is not an integer, real, or double-precision variable.

†ILLEGAL DO TERMINATOR
DO loops must not terminate on unconditional transfer statements.

ILLEGAL DO VARIABLE OR PARAMETER TYPE
The DO-loop variable or parameter is not type integer, real, double-precision, or Boolean.

†ILLEGAL FORMAT NAME
A format identifier cannot be recognized as a statement label or the name of an array.

†ILLEGAL IMPLICIT STATEMENT ARGUMENTS
IMPLICIT statement argument is not an alphabetic character or the range of characters specified is illegal.

ILLEGAL LOGICAL EXPRESSION
An operand in a logical expression is not of type logical. This is prohibited by the ANSI FORTRAN Standard.

ILLEGAL MASKING OR BOOLEAN EXPRESSION
One or both operands in a masking or Boolean expression is of type double precision or complex. Masking and Boolean expression operands must be single word entities.

ILLEGAL MIX OF CHARACTER AND NONCHARACTER IN COMMON BLOCK "*name*"
It is illegal to mix character and noncharacter entities in the same common block.

ILLEGAL MIX OF CHARACTER AND NONCHARACTER IN EQUIVALENCE AT SEQUENCE "*number*"
It is illegal to mix character and noncharacter entities in the same EQUIVALENCE statement.

ILLEGAL MIXED MODE OR CONVERSION
The types of two operands in an expression are incompatible or the type of array element or variable being defined is incompatible with the type of expression being evaluated.

†ILLEGAL NUMBER IN NAME FIELD
A symbolic name must not begin with a number.

†ILLEGAL OR DUPLICATE PARAMETER DEFINITION
Symbolic name of type integer, real, double precision, or complex not followed by an arithmetic expression. Symbolic name of type logical not followed by a logical expression. A symbolic name has been assigned more than once in the same program unit.

ILLEGAL PLACEMENT OF ALIGN, DIRECTIVE IGNORED
The ALIGN compiler directive was not placed immediately before a DO statement, a statement with a referenced statement label, a PROGRAM statement, a SUBROUTINE statement, a FUNCTION statement, or an ENTRY statement. The directive will be ignored.

ILLEGAL POINTEE "*name*"
A pointee cannot be a dummy argument or a pointer. It cannot be equivalenced or be specified in a common block statement.

ILLEGAL POINTER VARIABLE "*name*"
A pointer must be a simple variable. It cannot appear in an EQUIVALENCE statement. If defined in a PARAMETER or DATA statement, the definition must not precede its definition as a pointer.

ILLEGAL RELATIONAL EXPRESSION
One or both of the operands in a relational expression is an illegal type for a relational expression. The most common error is comparing logical values with .EQ. or .NE. The logical operators .EQV. or .NEQV. should be used in these cases.

ILLEGAL STATEMENT LABEL IN IO CONTROL LIST
A 1- to 5-digit statement number is missing after END= or ERR=.

ILLEGAL STATEMENT SEQUENCE

An improper sequence of statement types has been encountered (for example, a GO TO statement followed by a DIMENSION statement).

ILLEGAL STATEMENT TYPE

A statement keyword is misspelled (for example, DIMENSOIN) or is otherwise unidentifiable.

ILLEGAL STATEMENT TYPE IN BLOCK DATA SUBPROGRAM

A statement appears in a block data subprogram which is not provided for by the ANSI FORTRAN Standard, that is, an INTRINSIC or EXTERNAL statement.

ILLEGAL SUBSCRIPT TYPE "*name*"

A subscript expression is not of type integer or contains a constant that exceeds 2^{24}-1.

ILLEGAL SUBSTRING

A substring for a character item is incorrectly formed or an attempt is made to use a substring with an entity which can not have a substring (such as a character constant).

ILLEGAL SYNTAX IN NAMELIST

Illegal element found in NAMELIST statement.

ILLEGAL TYPE FOR ASSIGNED VARIABLE

A variable reference in an ASSIGN statement is not of type integer.

†ILLEGAL TYPE LENGTH

Length specified is not allowed for this data type.

ILLEGAL UNIT SPECIFIER

The unit specifier for INQUIRE must be an integer expression.

†ILLEGAL USE OF ** IN CONSTANT EXPRESSION

A constant expression specifies exponentiation to a non-integer power.

ILLEGAL USE OF ASSUMED CHARACTER LENGTH

A character entity with a length of * must be a dummy argument, the symbolic name of a constant, or an external function whose name appears in a FUNCTION or ENTRY statement within the same program unit.

ILLEGAL USE OF ASSUMED SIZE ARRAY "*name*"

An array with an asterisk for the last dimension cannot be used without subscripts in an I/O statement.

ILLEGAL USE OF COLON

A colon can only be used in a FORMAT statement or to separate the lower and upper dimensions in a declarative.

ILLEGAL USE OF DUMMY ARGUMENT "*name*"
A dummy argument in a procedure subprogram cannot be named the same as a local variable or another dummy argument.

ILLEGAL USE OF DUMMY ARGUMENT "*name*" IN EQUIVALENCE AT SEQUENCE "*number*"
Dummy arguments may not appear in an EQUIVALENCE statement.

ILLEGAL USE OF FUNCTION "*name*"
A function name cannot be used as an array name.

ILLEGAL USE OF FUNCTION "*name*" IN EQUIVALENCE AT SEQUENCE "*number*"
A function name cannot be used as an array name in an EQUIVALENCE statement.

ILLEGAL USE OF "*name*" IN I/O LIST
External, function, or program name not permitted in an I/O list.

ILLEGAL USE OF "*name*"
Group name referenced previous to its definition in a NAMELIST statement.

ILLEGAL USE OF NAMELIST GROUP "*name*"
A namelist group "*name*" can be used only as a group name in a NAMELIST read or write.

ILLEGAL USE OF TASK COMMON
The named common block was declared as both a task common block and a regular common block in the same subprogram.

ILLEGAL USE OF TASK COMMON VARIABLE
A task common variable was used illegally in a DATA, NAMELIST I/O, or SAVE statement.

ILLEGAL VALUE IN CONSTANT EXPRESSION
The evaluation of a constant expression yields a result that is out of range.

IMPLICIT NONE MUST BE ONLY IMPLICIT STATEMENT
IMPLICIT NONE or IMPLICIT SKOL appear in the same program unit as another IMPLICIT statement.

IMPROPERLY NESTED DO LOOP
Inner DO loop is not contained entirely within the outer DO loop range.

INCORRECT ARGUMENT TYPE
Actual argument is of the wrong type in a function reference.

INPUT FILE EMPTY

An end-of-file record was encountered as the first record of the source input dataset.

INTEGER*2=24 BIT INTEGER

INTEGER*2 is implemented as a 24-bit integer by CFT.

†INTEGER CONSTANT EXPECTED WHERE "*char*" OCCURS

When a format edit descriptor field is parsed, "*char*" appears where an integer constant is expected.

INTEGER CONSTANT EXPRESSION REQUIRED

The subscript or substring expression is not an integer constant expression.

INTRINSIC FUNCTION "*name*" CANNOT BE ACTUAL ARGUMENT

Certain intrinsic functions cannot be passed to subprograms as actual arguments.

INTRINSIC FUNCTION "*name*" IS NONSTANDARD

The specified intrinsic function is a CFT intrinsic function and is not provided for in the ANSI FORTRAN Standard. CFT uses the intrinsic version unless the function is declared external. This message is NON-ANSI if ANSI is specified on the CFT control statement and "*name*" is confirmed as an intrinsic function in an INTRINSIC statement.

INTRINSIC FUNCTION USED WITH ILLEGAL ARGUMENT TYPE

The actual argument(s) to the intrinsic function is an improper type.

INT24 IGNORED WITH EMA OPTION

When the EMA (extended memory addressing) CPU characteristic is in effect, the INT24 compiler option is ignored; 64-bit integers are used whenever the EMA option is used.

IO CONTROL LIST SPECIFIER MUST BE CHARACTER EXPRESSION

The I/O control list specifier must be evaluated to a character value.

IO CONTROL LIST SPECIFIER MUST BE CHARACTER VARIABLE OR ARRAY ELEMENT

The I/O control list specifier can be a character variable or an array element.

IO CONTROL LIST SPECIFIER MUST BE INTEGER EXPRESSION

The I/O control list specifier must be evaluated to an integer value.

IO CONTROL LIST SPECIFIER MUST BE INTEGER VARIABLE OR ARRAY ELEMENT

The I/O control list specifier can be an integer variable or an array element.

IO CONTROL LIST SPECIFIER MUST BE LOGICAL VARIABLE OR ARRAY ELEMENT

The I/O control list specifier can be a logical variable or an array element.

LAST ARRAY ONLY PARTIALLY INITIALIZED
The last element in a DATA statement variable list is an unsubscripted array and not enough constants are specified to completely fill the array. Remaining elements of the array are not initialized.

†LEFT PARENTHESIS EXPECTED
A required opening parenthesis was omitted.

LEFT PARENTHESIS EXPECTED IN EQUIVALENCE AT SEQUENCE "*number*"
A required opening parenthesis was omitted in an EQUIVALENCE statement.

LINE LENGTH > 133 CHARACTERS
One or more lines exceeds 133 characters during FORMAT statement editing.

LIST DIRECTED IO ILLEGAL FOR INTERNAL FILE
List-directed reads and writes all illegal operations on internal files. Internal file IO must be formatted.

LOGICAL OPERATOR MUST END IN PERIOD
A period does not follow an otherwise correct logical operator.

LOSS OF PRECISION IN TYPE CONVERSION
The type of a variable and the type of the associated constant in a DATA statement differ. The constant is converted to the type of the variable and precision is lost.

LOWERCASE CHARACTERS IN KEYWORDS OR IDENTIFIERS ARE NONSTANDARD
At least one lowercase alphabetic character which is not part of a character constant or comment appears in a program unit. This lowercase alphabetic character may be a keyword, identifier, or control character such as a Hollerith character constant descriptor. Lowercase characters are not provided for in the ANSI FORTRAN Standard. This message is issued only once in a program unit that contains lowercase characters.

LOWERCASE CHARACTERS USED AS EDIT DESCRIPTORS ARE NONSTANDARD
When a format specifier list was parsed, CFT encountered at least one lowercase character used as an edit descriptor. The ANSI FORTRAN Standard character set does not include lowercase characters.

MAIN PROGRAM MUST BE NAMED FOR FLOW TRACE
The main program must be named if flow trace is enabled by using the ON=F control statement option or by using a CDIR$ FLOW directive in the source program.

MASKING OR BOOLEAN EXPRESSION IS NONSTANDARD

A masking or Boolean expression was detected by CFT. These expressions are not provided for in the ANSI FORTRAN Standard.

MAXIMUM CODE SIZE EXCEEDED

The code size for the current program unit has exceeded 4 million words (16 million parcels).

MAXIMUM DATA BLOCK SIZE EXCEEDED

The current program unit local data block size, in Extended Memory Addressing (EMA) mode, has exceeded 16 million words.

MAXIMUM LEGAL ITERATION COUNT EXCEEDED

A DO-loop trip count is larger than the allowable maximum of $2^{23}-1$.

MINIMUM ONE PASS DO-LOOPS ARE NONSTANDARD

The control statement option ON=J was selected, causing all DO loops to execute at least once. This is a CFT extension not provided for in the ANSI FORTRAN Standard. This message is issued for all DO loops in a program compiled with ON=J.

†MISSING =

An equal sign is missing in a PARAMETER or statement function definition statement.

MISSING = IN CONTROL LIST

There is no equal sign after an option in an I/O statement control list.

MISSING COLON

A required colon has been omitted in a substring expression.

MISSING END STATEMENT

The last or only program unit being compiled lacks an END statement in its last line.

MISSING OR ILLEGAL CONSTANT LIST

A PARAMETER or DATA statement has not specified a constant list, or a list has a missing separator.

†MISSING OR ILLEGAL STATEMENT NUMBER IN DO

The statement number is missing or it contains illegal characters in a DO statement.

†MISSING OR ZERO COUNT FOR HOLLERITH STRING

The count field for a Hollerith edit descriptor is missing or zero in a format specifier list.

MISSING RIGHT PARENTHESIS OR UNEXPECTED END OF FORMAT

When a format specifier list was parsed, CFT unexpectedly reached the end of the format statement. This can occur when the parentheses are unmatched or when a Hollerith string count is too large and contains the closing parenthesis at the end of a FORMAT statement.

MISSING STATEMENT NUMBER IN ASSIGN

An ASSIGN statement lacks a statement label reference.

MISSING TO IN ASSIGN STATEMENT

An ASSIGN statement requires the keyword extension TO.

MODIFICATION OF DO CONTROL VARIABLE WITHIN LOOP IS NONSTANDARD

CFT has detected the modification of the DO-loop control variable inside the DO loop. This is allowed by CFT but not allowed by the ANSI FORTRAN Standard.

MORE THAN 8 MILLION WORDS OF LOCAL VARIABLES DECLARED IN SUBROUTINE

A subroutine may not use more than 8 million words for local variables. If possible, use common blocks for some of the variables.

MORE THAN 312 DUMMY ARGUMENTS IN PROGRAM UNIT

CFT does not accept more than 312 dummy arguments in a subroutine or function subprogram. Each argument for an entry point in a program unit represents a separate argument when computing the number of arguments used in a program unit.

MORE THAN 511 DISTINCT CHARACTER LENGTHS DECLARED IN THIS PROGRAM UNIT

There cannot be more than 511 distinct character lengths declared in a program unit. These lengths include character variables, character constants, and character temporaries.

MORE THAN ONE ELSE STATEMENT AT THIS IF LEVEL

Only one ELSE statement is permitted per IF-level.

MORE THAN ONE UNNAMED BLOCK DATA SUBPROGRAM IS NONSTANDARD

More than one unnamed block data subprogram appears in a compilation. CFT allows a maximum of 26 unnamed block data subprograms per compilation, but the ANSI FORTRAN Standard allows only one unnamed block data subprogram per compilation.

†NAME LONGER THAN EIGHT CHARACTERS

A symbolic name must not contain more than eight characters.

NO BLOCK IF ASSOCIATED WITH ELSE STATEMENT

An ELSE statement must follow a block IF statement and precede an END IF statement of the same level.

NO BLOCK IF ASSOCIATED WITH END IF STATEMENT

An END IF must be uniquely associated with an IF(*e*)THEN statement of the same IF-level.

NO PATH TO THIS STATEMENT

The previous statement is an unconditional transfer and this statement has no statement number.

NONSTANDARD "*name*" SPECIFIER

A CFT extended form of a unit or format specifier appears in an I/O control list. This form is not allowed in the ANSI FORTRAN Standard.

†NONSTANDARD "*name*" STATEMENT SYNTAX

An extended form of an ANSI FORTRAN Standard statement type indicated by "*name*" was used in a program.

NONSTANDARD ARITHMETIC EXPRESSION

An arithmetic or relational expression is formed with operand types not provided for in the ANSI FORTRAN Standard. An example is adding an integer variable to a Hollerith constant, which is a CFT extension.

NONSTANDARD BLOCK DATA STATEMENT SYNTAX

Parameters appear on a BLOCK DATA statement. These are CFT extensions to the ANSI FORTRAN Standard.

†NONSTANDARD DIMENSION DECLARATOR

A dimension declarator expression contains noninteger constants or variables, or function references. The ANSI FORTRAN Standard specifies that only integer variables and constants can be used in a dimension declarator expression.

†NONSTANDARD EDIT DESCRIPTOR FIELD

An edit descriptor not provided for in the ANSI FORTRAN Standard or an extended form of a standard edit descriptor was used in a format specifier list.

†NONSTANDARD OPERATOR "*name*"

An operator not provided for in the ANSI FORTRAN Standard, such as the .XOR. or .X. operator, was used. .XOR. and .X. are CFT extensions to the ANSI FORTRAN Standard.

NONSTANDARD RELATIONAL EXPRESSION

A relational expression compares a pair of operands in a way not provided for in the ANSI FORTRAN Standard. An example is comparing a character constant to an integer variable, which is a CFT extension. Some nonstandard relational expressions may receive the message "NONSTANDARD ARITHMETIC EXPRESSION" because of operator conversion during compilation.

†NONSTANDARD STRING DELIMITER

CFT allows string constants to be delimiters by using quotation marks in place of apostrophes. Asterisks can also be used in format specifier lists. Quotation marks and asterisks are not provided for in the ANSI FORTRAN Standard.

†NONSTANDARD TYPE DECLARATION
A TYPE * BYTE COUNT type declaration or nonstandard IMPLICIT statement, such as an IMPLICIT NONE statement, appears in a program unit, or a double declaration type statement is used in place of a DOUBLE PRECISION type statement. These are CFT extensions to the ANSI FORTRAN Standard.

NOT ENOUGH DO PARAMETERS
Fewer than two arguments have been encountered after the equal sign in a DO statement.

NOT ENOUGH MEMORY TO COMPILE
The program unit is too long to compile in the available memory.

†OCTAL CONSTANT IS NONSTANDARD
Octal constants were used, and they are not provided for in the ANSI FORTRAN Standard. These are CFT extensions to the ANSI FORTRAN Standard.

OPTIMIZATION BLOCK BROKEN AT THIS POINT
The code size forced CFT to terminate an optimization block at this point. A new optimization block begins with the next statement.

PARAMETER USED TWICE IN STATEMENT FUNCTION PARAMETER LIST
A given symbolic name can appear only once in a single dummy argument list.

†PARENTHESES NESTED TOO DEEPLY
The number of nested parentheses allowed by CFT in a format specifier list exceeded the maximum limit of nine nested parentheses.

PASS TWO SKIPPED BECAUSE OF FATAL PASS ONE ERRORS
Pass two of CFT's compilation is skipped for this program unit due to fatal pass one errors.

†PERIOD EXPECTED WHERE "*char*" OCCURS
When a format edit descriptor field is parsed, "*char*" appears where a period is expected.

PLEASE RERUN WITH SMALLER VALUE FOR MAXBLOCK
This message follows a compiler error or an internal compiler error message if the value for the MAXBLOCK control statement parameter is greater than the system default value.

POINTER MUST BE TYPE INTEGER
A pointer variable must not be assigned a type other than integer.

POSSIBLE BRANCH INTO BLOCK IF VIA ASSIGN OR END=/ERR= WITH LABEL "*l*"
Label "*l*" is a FORTRAN statement number defined in an IF-block, ELSE IF-block, or ELSE-block and appears in an ASSIGN statement or in an END= or ERR= branch of an I/O statement. Branches into IF-blocks, ELSE IF-blocks, or ELSE-blocks are not provided for in the ANSI FORTRAN Standard.

POSSIBLE BRANCH INTO INACTIVE DO LOOP; STATEMENT LABEL "*nnn*"

CFT detected a branch to a labeled statement inside the range of a DO loop from a branch statement outside the range of the DO loop.

POSSIBLE BRANCH INTO INACTIVE DO LOOP VIA ASSIGN OR END=/ERR= WITH LABEL "*n*"

"*n*" is a statement label defined within the range of a DO loop. It has appeared in an ASSIGN statement or in the END= or ERR= branch of an I/O statement in the program unit. Verify that the branches to the statement label occur only within the innermost DO loop where the label is defined.

†PREVIOUS IMPLICIT REFERENCES THIS CHARACTER

Only one IMPLICIT reference is permitted per character.

PREVIOUS REFERENCES TO "*name*"

An ENTRY name has been used before its declaration as an ENTRY.

PROGRAM UNIT TOO LARGE TO COMPILE

One of CFT's internal tables has overflowed because there is too much code in a program unit.

REAL*8 = SINGLE PRECISION

REAL*8 is implemented as single-precision by CFT.

RECURSIVE SUBROUTINE OR FUNCTION REFERENCE OF "*name*"

The function, subroutine, or entry name was referenced within the same program unit that defined it.

RECURSIVE SUBROUTINE REFERENCE "*name*" USED AS AN ARGUMENT

The main subroutine name was used as an argument to another function or subroutine call.

REFERENCES TO ARRAY "*name*" WITH NO SUBSCRIPTS

The array named was referenced without subscripts in a statement that required them.

RELATIONAL EXPRESSION WITH DOUBLE PRECISION AND COMPLEX IS NONSTANDARD

A relational expression was detected with a complex and double-precision operand. CFT converts the double-precision operand to a complex operand. The ANSI FORTRAN Standard does not provide for these conversions. Some nonstandard relational expressions with double-precision and complex operands may receive the message ARITHMETIC EXPRESSION WITH DOUBLE PRECISION AND COMPLEX IS NONSTANDARD because of operator conversion during compilation.

†REPETITION COUNT ILLEGAL FOR "*name*"

A repetition count appears before the nonrepeatable edit descriptor "*name*" in a format specifier list.

†REPETITION COUNT MUST BE > ZERO

The repetition count before a repeatable edit descriptor in a format specifier list is zero.

†REPETITION COUNT TOO LARGE

The value of *n* in the *n*X edit descriptor field moves the next character position to the left of the first position.

RETURN ILLEGAL IN MAIN PROGRAM

A RETURN statement is encountered in a main program unit.

†RIGHT PARENTHESIS EXPECTED

A required closing parenthesis was omitted.

RIGHT PARENTHESIS EXPECTED IN EQUIVALENCE AT SEQUENCE "*number*"

A required closing parenthesis was omitted in an EQUIVALENCE statement.

SCALAR DUMMY ARGUMENT "*name*" USED AS FORMAT IDENTIFIER

The integer variable named appears both as a format identifier and as an entry in a dummy argument list in this program unit. The cause might be a missing DIMENSION statement.

SCAN STOPPED, TOO MANY ERRORS IN FORMAT

CFT attempts recovery of up to three errors before abandoning the FORMAT statement.

SPECIFIER RECL LEGAL IF AND ONLY IF ACCESS IS DIRECT

In an OPEN statement, the RECL control list option is supported only if direct access is specified.

STATEMENT FUNCTION "*name*" IN COMMON OR ARGUMENT LIST

Statement function must not appear as a variable in a common block or an argument list.

STATEMENT FUNCTION "*name*" REFERENCES ITSELF

A statement function definition statement cannot be recursive.

†STATEMENT FUNCTION PARAMETER MUST NOT BE ARRAY

The names of variables appearing as dummy arguments of a statement function have a scope of that statement only.

STATEMENT LABEL IGNORED

Statement label is ignored because transfer to this statement is prohibited.

STATEMENT LENGTH EXCEEDED

The statement, when arithmetic statement functions have been expanded, exceeds CFT's limit on size of statements.

STATEMENT NUMBER ILLEGAL ON DECLARATIVE
CFT does not allow a statement number on ENTRY statements.

STATEMENT NUMBER ON BLANK CARD IGNORED
Blank lines cannot contain statement labels.

SUBROUTINE "*name*" NOT DECLARED
IMPLICIT NONE or IMPLICIT SKOL has been specified but "*name*" did not appear in an EXTERNAL statement.

SUBSCRIPT OUT OF DIMENSION BOUNDS IN EQUIVALENCE AT SEQUENCE "*number*"
Subscript exceeds the value given in the dimensions.

SUBSTRING EXPRESSION OUT OF BOUNDS
In a substring expression "(C1:C2)", the relations 1 ≤ C1 ≤ C2 ≤ LEN do not all hold (where LEN is the declared length of the character entity).

SYNTAX ERROR
Illegal element, name where number required, or extra or missing punctuation.

SYNTAX ERROR IN ENCODE OR DECODE STATEMENT
Illegal element in ENCODE or DECODE statement.

†SYNTAX ERROR IN IMPLIED DO
An implied-DO list specified in a DATA statement is of improper syntactical form, references a variable that is not an implied-DO variable, or references an array element that does not specify the implied-DO variable for this implied-DO list in its subscript.

SYNTAX ERROR IN IO CONTROL LIST
Illegal element in I/O control list.

†SYNTAX ERROR IN UNIT IDENTIFIER
A syntax error was found in a unit identifier.

†TAB COUNT MUST NOT BE ZERO
A tab edit descriptor (T, TL, or TR) appears in a format specifier list followed by a tab count of 0.

TASK COMMON BLOCK "*name*" IS STATICALLY ALLOCATED
A task common block was declared when the allocation specification was defined as STATIC.

TASK COMMON IS NONSTANDARD
A task common block is not provided for in the ANSI FORTRAN Standard.

†TASK COMMON MUST BE NAMED
A blank common block was declared with the CFT extension task common block.

TEST EXPRESSION MUST BE LOGICAL

Expression type in a logical IF must be logical or Boolean.

TEST EXPRESSION MUST NOT BE CHARACTER

Cannot have character expression type in a logical or arithmetic IF statement.

TEST EXPRESSION MUST NOT BE LOGICAL

Expression type in an arithmetic IF must not be type logical.

TOO MANY COMMON BLOCKS DECLARED

More than 120 distinct common blocks were declared in a single program unit.

TOO MANY DO PARAMETERS

More than three arguments have been encountered after the equal sign in a DO statement.

TOO MANY DOS ON STATEMENT

More than 15 DO loops ended on the same statement.

TOO MANY POINTERS DECLARED

More than 312 pointers were declared in a single program unit.

TOO MANY SUBSCRIPTS

An array reference contains more subscripts than the subscripts declared.

TOO MANY SUBSCRIPTS IN EQUIVALENCE AT SEQUENCE "*number*"

An array reference in an EQUIVALENCE statement has more subscripts than were declared.

TOO MANY UNNAMED BLOCK DATA SUBPROGRAMS

The ANSI FORTRAN Standard allows only one unnamed block data subprogram to be used in a program. CFT allows a maximum of 26 unnamed block data subprograms. More than 26 block data subprograms appeared during the compilation.

TWO BRANCH IF STATEMENT IS A NONSTANDARD STATEMENT

A 2-branch arithmetic or logical IF statement appears in a program. These statements are CFT extensions to the ANSI FORTRAN Standard.

TYPE CONVERSION IN DEFINITION

A constant in a PARAMETER statement was not converted to the type of the corresponding symbolic name.

TYPE OF "*name*" NOT DECLARED

"*name*" was declared in an EXTERNAL statement, but did not appear in an explicit type statement.

TYPE MISMATCH BETWEEN "*name*" AND EXPRESSION

The expression being assigned to *name* is not of the same type as *name* (for example, a logical variable being assigned a character string in a PARAMETER statement).

TYPE STATEMENT IGNORED FOR INTRINSIC FUNCTION "*name*"

Type statements do not change the type of an intrinsic function and are ignored.

UNBALANCED PARENTHESIS

Opening and closing parentheses do not match; required parenthesis not present.

UNDEFINED ITEM IN CONSTANT EXPRESSION

A constant expression in a PARAMETER or DATA statement is specified with other than constants or the symbolic names of constants. A constant expression in a DATA statement is specified with other than constants, the symbolic names of constants, or the names of implied-DO variables.

UNDEFINED STATEMENT NUMBER "*number*"

A referenced statement label is not defined.

UNDEFINED SUBSCRIPT

An equivalence subscript must be constant.

UNEXPECTED END OF STATEMENT

A statement encountered is syntactically incomplete.

UNIT=* ILLEGAL FOR DIRECT ACCESS

[UNIT=]* appeared in a direct access READ or WRITE statement.

UNIT=* ILLEGAL FOR UNFORMATTED IO

[UNIT=]* appeared without a format identifier in a READ or WRITE statement.

UNIT=* LEGAL ONLY IN READ OR WRITE

[UNIT=]* appeared in an auxiliary I/O statement.

UNKNOWN LOGICAL OPERATOR

The characters following a period do not represent a logical operator.

UNRECOGNIZED COMPILER DIRECTIVE

The compiler directive is misspelled or does not exist for CFT.

UPPER DIMENSION < LOWER DIMENSION

The lower dimension must be less than or equal to the upper dimension.

USE OF END ILLEGAL IN WRITE CONTROL LIST

END= may not be specified in a WRITE statement.

VALUE NOT ASSIGNED TO FUNCTION NAME
Function subprogram is missing value assignment for the function.

VARIABLE DIMENSION ARRAY "*name*" MUST BE DUMMY ARGUMENT
A variably dimensioned array must appear as a dummy argument at some entry point.

VARIABLE DIMENSION ILLEGAL FOR ARRAY IN COMMON
An attempt was made to put a variably dimensioned array into COMMON.

VARIABLE LIST LONGER THAN CONSTANT LIST
Constants and variables must correspond one-to-one in a DATA statement.

VARIABLE "*name*" USED AS ARRAY OR FUNCTION
A simple variable is referenced with either subscripts or an argument list.

VERY LARGE LOCAL DATA BLOCK; USE EXTENDED MEMORY COMMON BLOCK
Very large local arrays were declared, causing the generated code to end at more than 4 million words of memory.

VERY LARGE OFFSET ENCOUNTERED; USE EXTENDED MEMORY ADDRESSING
A calculated offset greater than 4 million words was detected with a nonextended memory variable. An extended memory variable must be declared in a common block.

ZERO SUBSCRIPT INCREMENT
A CIV subscript must have a nonzero increment.

ZERO TO NEGATIVE POWER
Raising zero to a zero or negative power produces unpredictable results in an executable program.

D.2 LOGFILE MESSAGES

The following messages appear in the logfile following the CFT statement if the indicated condition occurs. Some of the conditions cause compiler execution to terminate after processing the CFT statement. Control statement processing resumes with an EXIT statement if there is one in the control statement file; otherwise, the job terminates.

CF007 - BAD PARAMETER TO KEYWORD *keyword = parameter*
The parameter for the keyword is out of range or undefined.

CF008 - NULL INPUT FILE ILLEGAL
I=0 is an illegal input.

CF009 - B=0 and ON=Z INCOMPATIBLE OPTIONS
B must specify a file if the Z option is on.

CF010 - ON = *character* PARAMETER NOT ALPHA
CF010 - OFF = *character* PARAMETER NOT ALPHA
All characters in the strings for ON and OFF must be alphabetic.

CF011 - *letter* OPTION NOT IMPLEMENTED
No existing ON/OFF option is associated with the letter.

CF012 - *nn* CFT CONTROL CARD ERRORS
Gives count of control statement errors.

CF013 - WARNING: *string* WILL BE SET TO OFF
Options listed in *string* appear in both the ON= and OFF= keyword parameter lists.

CF014 - DOUBLY DEFINED OPTION FOR OPT= KEYWORD
An option set by an OPT= keyword parameter was defined two times or redefined in the parameter list.

CF015 - HEAP BASED ALLOCATION NOT YET IMPLEMENTED
ALLOC=HEAP was specified on the CFT control statement. Heap Memory management is not implemented by CFT.

CF016 - CPU TYPE UNKNOWN - MAY EFFECT GENERATED CODE
The CPU type from the Job Communication Block is unknown to CFT and optimizations may be effected.

CF017 - 1 WARNING
CF017 - *n* WARNINGS
Warning errors were encountered during compilation.

CF018 - WARNING: MAXBLOCK WILL BE SET TO 1
When compiling with DEBUG on the COS CFT control statement or with -e D specified on the UNICOS CFT command, MAXBLOCK is set to 1.

CF019 - WARNING: Z WILL BE SET TO ON
When compiling with DEBUG on the CFT control statement, Z is forced on.

CF020 - WARNING: I WILL BE SET TO ON
When compiling with DEBUG on the CFT control statement, I is forced on.

CF023 - 1 NON-ANSI MESSAGE ISSUED
CF023 - *n* NON-ANSI MESSAGES ISSUED
Nonstandard FORTRAN was detected when compiling with ANSI specified on the CFT control statement.

CF025 - ILLEGAL MAINFRAME TYPE TO KEYWORD CPU=*mainframe type*
An illegal mainframe type was specified on the CPU keyword or the COS TARGET control statement.

CF026 - ILLEGAL VALUE *value* FOR PARAMETER *param* TO KEYWORD CPU=
An illegal value was entered for a numeric trait of the CPU keyword or the TARGET control statement.

CF027 - DUPLICATE OR CONFLICTING PARAMETER TO KEYWORD CPU= *string*
A parameter to the CPU keyword was defined twice or was redefined.

CF028 - WARNING:[NO]CI AND [NO]GS WILL BE REMOVED IN CFT 1.16; USE [NO]CIGS INSTEAD
[NO]CI and [NO]GS will not be supported in CFT 1.16.

CF029 - *lmdn* ALREADY LOCAL
The LOOPMARK option on the CFT compiler call uses dataset *lmdn* as a scratch dataset. *lmdn* was a local dataset at the start of compilation. Modify the job so that *lmdn* is not a local dataset at the start of compilation.

D.3 NOVECTOR MESSAGES

NOVECTOR messages indicate the reason an innermost loop did not vectorize. If LOOPMARK or LOOPMARK=MSGS is specified on the CFT compiler call, a NOVECTOR message is printed in the source listing on the line immediately following each innermost scalar loop.

NOVECTOR messages are also printed as part of the Table of Loops Encountered.

'*name*' IN RECURRENCE AT S.N. *sn* USED AS A SUBSCRIPT OR IS OF TYPE INT24
Using a reduction variable as a subscript inhibits vectorization. Reductions involving 24 bit integers do not vectorize. For example:

```
      DIMENSION IARRAY1(100),IARRAY2(100),IARRAY3(100)
      DO 10 I = 1,N
        ISUM = ISUM + IARRAY1(I)
        IARRAY2(ISUM) = IARRAY3(ISUM)
   10 CONTINUE
```

'*name*' REFERENCED AT S.N. *sn* BEFORE BEING DEFINED AT S.N. *sn*

A loop will not vectorize if the first reference to a variable in the loop occurs before the first definition of the variable in the loop. For example:

```
      DIMENSION A(100)
      II = 1
      DO 10 I = 2,N
           A(II) = 3.0
           II = I
      10 CONTINUE
```

'*name*' STATEMENT AT S.N. *sn*

STOP statements inhibit vectorization. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
        IF (A(I) .EQ. 5.0) STOP
   10 CONTINUE
```

ASSIGNED 'GOTO' AT S.N. *sn*

Assigned GOTOs inhibit vectorization. For example:

```
      DIMENSION A(100)
      ASSIGN 20 TO JUMPOUT
      DO 10 I = 1,N
        IF (A(I) .EQ. 5.0) GO TO JUMPOUT
   10 CONTINUE
   20 CONTINUE
```

BACKWARD BRANCH FROM S.N. *sn* TO LABEL '*label number*' AT S.N. *sn*

Backward branches within a loop inhibit vectorization. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
   20   CONTINUE
        A(I) = A(I) + 1.0
        IF (A(I) .LE. 5.0) GO TO 20
   10 CONTINUE
```

BRANCH BETWEEN DEFINITION OF '*name*' AT S.N. *sn* AND
OCCURRENCE AT S.N. *sn*

A branch statement between the definition of a vector temporary and a use or redefinition of the vector temporary inhibits vectorization. For example:

```
      DIMENSION A(100),B(100),C(100)
      DO 10 I = 1,N
        VECTTEMP = A(I)
        IF (MOD(I,2) .EQ. 0) VECTTEMP = B(I)
        C(I) = VECTTEMP
   10 CONTINUE
```

CDIR$ 'NEXTSCALAR'

The compiler directive NEXTSCALAR preceded this loop. For example:

```
      DIMENSION A(100)
CDIR$ NEXTSCALAR
      DO 10 I = 1,N
        A(I) = 0.0
      10 CONTINUE
```

CDIR$ 'NORECURRENCE'

The compiler directive NORECURRENCE preceded this loop and this loop has an otherwise vectorizable recurrence. For example:

```
      DIMENSION IARRAY(100)
CDIR$ NORECURRENCE
      DO 10 I = 1,N
        ISUM = ISUM + IARRAY(I)
   10 CONTINUE
```

CDIR$ 'NOVECTOR'

The compiler directive NOVECTOR is in effect for this loop. For example:

```
         DIMENSION A(100)
   CDIR$ NOVECTOR
         DO 10 I = 1,N
           A(I) = 0.0
      10 CONTINUE
```

CHARACTER VARIABLE OR FUNCTION REFERENCE '*name*' AT S.N. *sn*

Character variables and character function references inhibit vectorization. For example:

```
      CHARACTER CHARARAY*1(100)
      DO 10 I = 1,N
        CHARARAY(I) = ' '
   10 CONTINUE
```

CII '*name*' DEFINED MORE THAN ONCE

Defining a CIV more than once in a loop inhibits vectorization. For example:

```
         DIMENSION A(100)
         J = 0
         DO 10 I = 1,N
           J = J + 1
           A(J) = 1.0
           J = J + 1
           A(J) = 2.0
      10 CONTINUE
```

CII '*name*' EQUIVALENCED WITH AN ARRAY

Equivalencing a CIV with an array inhibits vectorization. For example:

```
   DIMENSION A(100)
   EQUIVALENCE (I,A(50))
   DO 10 I = 1,N
     A(I) = I
10 CONTINUE
```

COMPUTED 'GOTO' AT S.N. *sn*

Computed GOTOs inhibit vectorization. For example:

```
DIMENSION A(100)
   DO 30 I = 1,N
     GOTO (10,20) I
     A(I) = I + 2
10   CONTINUE
     A(I) = I + 1
20   CONTINUE
     A(I) = I
30 CONTINUE
```

DEPENDENCY INVOLVING ARRAY '*name*' AT S.N. *sn*

A dependency that inhibits vectorization was encountered. See the NOVECTOR message on the following line of the listing for a further explanation. For example:

```
   DIMENSION A(100)
   DO 10 I = 1,N
     A(J) = A(I)
10 CONTINUE
```

DEPENDENCY INVOLVING ARRAY '*name*' AT S.N. *sn* AND *sn*

A dependency that inhibits vectorization was encountered. See the NOVECTOR message on the following line of the listing for a further explanation. For example:

```
   DIMENSION A(100)
   DO 10 I = 1,N
     B(I) = A(I-1)
     A(I) = C(I)
10 CONTINUE
```

DO CONTROL VARIABLE '*name*' MODIFIED AT S.N. *sn*

Assigning a value to a DO control variable, other than in a DO statement, inhibits vectorization. For example:

```
   DIMENSION A(100)
   DO 10 I = 1,N
      I = I + 3
      A(I) = 3.0
10 CONTINUE
```

'ELSEIF' OR NESTED 'IF' AT S.N. *sn*

ELSEIF statements and nested IF statements inhibit vectorization. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
        IF (A(I) .LE. 10.0) THEN
          A(I) = 10.0
        ELSEIF (A(I) .LE. 5.0) THEN
          A(I) = 5.0
        ENDIF
   10 CONTINUE
```

EXTERNAL REFERENCE '*name*' AT S.N. *sn*

Subroutine calls and external function references inhibit vectorization. For example:

```
      DO 10 I = 1,N
        CALL SUB1(I,X,Y,Z)
   10 CONTINUE
```

FAILED VECTOR TEMPORARY

An apparent vector temporary was not used.

I/O NOT VECTORIZABLE - '*name*' AT S.N. *sn*

Input and output statements inhibit vectorization. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
         WRITE (*,*) A(I)
   10 CONTINUE
```

I/O NOT VECTORIZABLE - IMPLIED DO AT S.N. *sn*

Input and output statements inhibit vectorization. For example:

```
      DIMENSION A(100,100)
      DO 10 I = 1,N
        WRITE (*,*) (A(I,J),J=1,N)
  10  CONTINUE
```

NO REDUCTION NOR VARIANT ARRAY DEFINITION ENCOUNTERED

If a loop does not have a reduction nor variant array definition, there is no reason to vectorize it. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
        X = A(I)
   10 CONTINUE
```

'OFF=V' SPECIFIED ON CFT CONTROL STATEMENT

When OFF=V is specified on the CFT control statement, all vectorization is suppressed.

OPTIMIZATION BLOCK BROKEN AT S.N. *sn*

CFT can only vectorize loops that fit into one optimization block. The size of intermediate text for this loop exceeds CFT's optimization block size.

OUTER LOOP

A loop is vectorizable only if it is an innermost loop or if all the loops nested in it are unrolled. For example:

```
C     DO 20 does not vectorize because DO 20 is not an innermost
C     loop and DO 10 does not unroll.

      DIMENSION A(100,100)
      DO 20 I = 1,N
        DO 10 J = 1,N
          A(I,J) = 0.0
   10   CONTINUE
   20 CONTINUE

C     DO 40 will vectorize if DO 30 is unrolled.
C     DO 30 will be unrolled by default or if UNROLL=n is
C     specified on the CFT control statement and n is
C     greater than or equal to 3.

      DO 40 I = 1,N
        DO 30 J = 1,3
          A(I,J) = 0.0
   30   CONTINUE
   40 CONTINUE
```

REASON FOR NOT VECTORIZING UNKNOWN

The reason for not vectorizing this loop is unknown. This is abnormal but does not affect the correctness of the compiled program. Please show your listing to a Cray Research systems analyst.

RECURRENCE OF '*name*' AT S.N. *sn* BEFORE BRANCH OUT OF LOOP
AT S.N. *sn*

Recurrences before branches out of loops inhibit vectorization. For example:

```
      DIMENSION A(100)
      DO 10 I = 1,N
        SUM = SUM + A(I)
        IF (SUM .GE. 100.0) GO TO 20
   10 CONTINUE
   20 CONTINUE
```

RECURRENCE OF _'name' AT S.N. *sn* USED AS AN OPERAND AT S.N. *sn*

Referencing the result of a recurrence elsewhere in the same loop inhibits vectorization. For example:

```
     DIMENSION A(100)
     DO 10 I = 1,N
       SUM = SUM + A(I)
       IF (MAX .LT. SUM) MAX = SUM
  10 CONTINUE
```

REDUCTION LOOP WITH CONSTANT ITERATION COUNT <= *n*

A reduction loop with a constant iteration count of less than or equal to *n* will not vectorize. The default value for *n* is 2 since reduction loops with iteration counts of 1 or 2 generally execute faster as scalar loops than as vector loops. The value of *n* can be changed with the compiler directive NORECURRENCE=*n*. For example:

```
     DIMENSION A(100)
CDIR$ NORECURRENCE=15
     DO 10 I = 1,10
       SUM = SUM + A(I)
  10 CONTINUE
```

SEARCH LOOP TOO COMPLICATED

This loop has more than eight branches out or 2 * double precision + 2 * complex + real + integer + logical number of stores is greater than eight.

```
**************************************************************
*** THIS MESSAGE AND EXPLANATION WILL BE IMPROVED FOR THE  ***
*** NEXT REVISION.  AN EXAMPLE LOOP WILL ALSO BE INCLUDED. ***
**************************************************************
```

SECOND ARGUMENT TO 'SHIFT' AT S.N. *sn* IS VARIANT

The intrinsic function SHIFT will not vectorize if the shift count varies within the loop. For example:

```
     DIMENSION IARRAY1(100),IARRAY2(100)
     DO 10 I = 1,N
       IARRAY1(I) = SHIFT(IARRAY2(I),I)
  10 CONTINUE
```

SUBSEQUENT PLUS GATHER/SCATTER

Loops with subsequent plus dependencies involving nonlinear subscripts do not vectorize. For example:

```
     DIMENSION IARRAY1(100)
     DIMENSION A(100),B(100)
     DO 10 I = 1,N
       A(IARRAY1(I)) = X
       C(I) = A(IARRAY1(I)+1)
  10 CONTINUE
```

TOO MANY INDIRECT ADDRESSES, NO GATHER/SCATTER HARDWARE

Code is being generated for a Cray computer system with no gather/scatter instructions. Gather/scatter instructions are found on certain CRAY X-MP Computer Systems, but are not found on CRAY-1 S Computer Systems.

On machines without gather/scatter instructions, indirect addressing will vectorize only if the number of operations is greater than the number of array references. If the number of operations is less than the number of array references, CFT generates scalar code since scalar code will execute faster than pseudo-vectorized code in this instance. For example:

```
     DIMENSION A(100),B(100),IARRAY(100)
     DO 10 I = 1,N
       A(I) = B(IARRAY(I))
  10 CONTINUE
```

UNVECTORIZABLE ASSIGNMENT TO SCALAR '*name*' AT S.N. *sn*

A CIV in a conditionally executed block must be of the form I = I ... in order to vectorize. If a CIV in a conditionally executed block is of any other form, for example, J = I ..., where I is a CIV, the loop will not vectorize. For example:

```
     DIMENSION A(100),B(100)
     DO 10 I = 1,N
       IF (A(I) .LT. 0.0) THEN
         J = I + 5
         B(J) = 2.0
       ENDIF
  10 CONTINUE
```

UNVECTORIZABLE RECURRENCE AT S.N. *sn*

See section 9 for a discussion of the types of recurrences that CFT vectorizes. This loop contains a recurrence that is not of a vectorizable form. For example:

```
     DIMENSION A(100)

     DO 10 I = 1,N
       X = A(I) / X
  10 CONTINUE

     DO 20 I = 1,N
       X = A(I) - X
  20 CONTINUE

     DO 30 I = 1,N
       X = X * A(I) + Y
  30 CONTINUE

     DO 40 I = 1,N
       X = SQRT(A(I)*X)
  40 CONTINUE
```

UNVECTORIZABLE TYPE FOR PRODUCT REDUCTION - '*name*' AT S.N. *sn*

Product and division reductions will not vectorize unless the operands are real. For example:

```
     DIMENSION IARRAY(100)
     DO 10 I = 1,N
        ISUM = ISUM * IARRAY(100)
  10 CONTINUE
```

UNVECTORIZABLE TYPE FOR REDUCTION - 'name' AT S.N. sn

Only real and integer reductions vectorize. For example:

```
     DOUBLE PRECISION DARRAY(100)
     DO 10 I = 1,N
       DSUM = DSUM + DARRAY(100)
  10 CONTINUE
```

VARIANT BASE POINTER '*name*' AT S.N. *sn*

A pointer that is variant within a loop points to an array that is defined or referenced in the loop. This is not vectorizable. For example:

```
     POINTER (IP,P(100))
     DO 10 I = 1,N
       IP = IP + N
       P(I) = 3.0
  10 CONTINUE
```

VECTOR TEMPORARY '*name*' AT S.N. *sn* IS IN A CONDITIONALLY EXECUTED BLOCK

Vector temporaries in conditionally executed blocks inhibit vectorization. For example:

```
     DIMENSION A(100),B(100)
     DO 10 I = 1,N
       IF (I .EQ. 3) THEN
         VECTTEMP = A(I)
         B(I) = VECTTEMP + 2.7
       ENDIF
  10 CONTINUE
```

ZERO INCREMENT CII

Zero increment CIVs inhibit vectorization. For example:

```
     DIMENSION A(100),B(100)
     DO 10 I = 1,N
       J = J + 1 - 1
       A(J) = B(I)
  10 CONTINUE
```

D.4 INFORMATIVE DEPENDENCY MESSAGES

When a dependency message is issued, another message also appears explaining why the dependency exists. The following list contains all the informative messages with examples of DO loops causing the message to be issued. Examples of the first two messages can be found in part 3, section 2.

PREVIOUS PLUS WITH A DECREMENTING SUBSCRIPT

PREVIOUS MINUS WITH AN INCREMENTING SUBSCRIPT

POTENTIAL PROBLEM WITH EQUIVALENCED ARRAYS

```
         DIMENSION E(100),D(50)
         EQUIVALENCE (E,D)
           .
           .
           .
         DO 10 I = M,N
         E(I+1) = 2.0
         D(I) = 3.0
    10   CONTINUE
```

If E and D are dimensioned to 100 elements, no dependency is detected.

```
         DIMENSION A(100),B(100)
         EQUIVALENCE (A(50),B)
           .
           .
           .
         DO 15 I = 1,100
         A(I) = X
         B(I) = Y
    15   CONTINUE
```

REFERENCE MADE TO AN ARRAY THAT IS NOT SUBSCRIPTED

```
         DO 20 I = 2,N
         A(I) = SASUM(N-I,A,1)
    20   CONTINUE
```

ARRAY USED AS AN ARGUMENT TO A SUBROUTINE/FUNCTION

```
         DO 40 I = M,N
         A(I) = 2.0
         CALL SB(A(I))
    40   CONTINUE
```

DEFINITION AND REFERENCE HAVE A DIFFERENT NUMBER OF SUBSCRIPTS

```
       DO 50 I = M,50
       A(I,K) = A(I)
    50 CONTINUE

       DO 30 I = 1,100
       B(I) = A(2,I)
       A(I) = 2.0
    30 CONTINUE
```

AMBIGUOUS OR CONFLICTING SUBSCRIPTS

```
       DO 55 I = 1,N
       B(I) = A(3)
       A(I) = 3.0
    55 CONTINUE

       DO 60 I = 1,N
       K = IA(I)
       A(K) = A(I)
    60 CONTINUE
```

CONFLICTING SUBSCRIPTS

```
       DO 10 I = 1,200
       A(I) = A(2)
    10 CONTINUE
```

NULL DEPENDENCY WITH CIV MODIFIED BETWEEN DEFINITION AND REFERENCE

```
       DO 10 I = 1,N
       B(I) = A(J)
       J = J+1
       A(J) = 1.0
    10 CONTINUE

       DO 5 I = 1,N
       A(J) = 3.0
       J = J-2
       A(J) = B(I)
    5  CONTINUE
```

Null means the difference between the subscripts is zero. If the subscript difference is not equal to zero, vectorization is possible.

AMBIGUOUS INCREMENT OF CIV

```
       DO 20 I = M,N,K
       A(I) = A(I+1)
    20 CONTINUE

       DO 30 I = 1,100
       A(J) = 1.0
       B(J) = A(J-1)
       J = J-K
    30 CONTINUE
```

DEPENDENCY POSSIBLE WITH ZERO INCREMENT

```
       DO 90 I = 1,M
       A(J) = A(J)+B(I)
       J = J+N
    90 CONTINUE
```

This message is only issued if OPT=ZEROINC is specified on the CFT control statement.

NO CIV WAS FOUND IN ARRAY REFERENCE

```
       DO 55 I = 1,N
       A(I) = 1.0
       B(J) = B(K)
    55 CONTINUE
```

E. OUTMODED FEATURES

This appendix describes non-ANSI features CFT supports but have generally been outmoded by alternatives meeting the standard and enhancing the portability of CFT programs. These outmoded features and their preferred alternatives are as follows.

Obsolete Feature	Preferred Alternative
Hollerith data	Character data
Two-branch arithmetic IF	Arithmetic IF or block IF
Indirect logical IF	Logical IF
ENCODE and DECODE	Internal files
Asterisk editing	Quotation mark editing
[-*b*]X editing	TL editing
DOUBLE declaration type statement	DOUBLE PRECISION declaration type statement
DOUBLE declaration FUNCTION statement	DOUBLE PRECISION declaration FUNCTION statement
DATA statement *nlist/clist* logical/Hollerith correspondence	*nlist/clist* correspondence both logical or both character
PUNCH statement	WRITE statement
Type statements with **n*	Standard type statements
DATA statement with declaratives	DATA statement after other declaratives
EOF, IEOF, and IOSTAT functions	End-of-file specifier (END=) or status specifier (IOSTAT=)

E.1 HOLLERITH DATA

Hollerith data is a sequence of any characters capable of internal representation as specified in appendix A. Its length is the number of characters in the sequence, including blank characters. Each character occupies a position within the storage sequence identified by one of the numbers 1, 2, 3, ... indicating its placement from the left (position 1). Hollerith data must contain at least one character.

A *Hollerith constant* is expressed in one of three forms. The first of these is specified as a nonzero integer constant followed by the letter H and as many characters as equal the value of the integer constant. The second form of Hollerith constant specification delimits the character sequence between a pair of apostrophes followed by the letter H.

The third form is like the second, except quotation marks replace apostrophes.

Example:

CHARACTER SEQUENCE	Form 1	Form 2	Form 3
ABC 12	6HABC 12	'ABC 12'H	"ABC 12"H

Two adjacent apostrophes or quotation marks appearing within the bounds of two delimiting apostrophes or quotation marks are interpreted and counted as a single apostrophe within the sequence. The character sequence, DON'T USE "*" would be specified with the apostrophe delimiters as 'DON''T USE "*"'H, and with the quotation mark delimiters as "DON'T USE ""*"""H.

Each character of a Hollerith constant character sequence is represented internally by its unique 8-bit code (see Appendix A) with up to eight such codes contained in a single 64-bit Cray computer word. The codes corresponding to character positions 1 through 8 of a Hollerith constant are sequentially represented from left to right in a Cray computer word. Successive groups of eight codes are similarly represented in as many successive Cray computer words. When the last position of a sequence is not an even multiple of 8, the unused portion of the computer word it occupies is to its right and contains up to seven blank character codes (040_8).

When the number of characters in a character sequence is fewer than eight, the single Cray computer word used can contain up to seven null character codes (000). The null character codes can be produced by substituting the letter L for the letter H in the Hollerith forms described above.

When fewer than eight characters appear in a Hollerith constant, the unused portion of a single Cray computer word can contain up to seven null character codes (000) to the left of the one or more codes representing the character sequence. The null character codes can be produced by substituting the letter R for the letter H in the first form of Hollerith constant expression or by suffixing the second apostrophe or quotation mark delimiter with the letter R in the second form.

All of the following Hollerith constant expressions yield the same Hollerith constant and differ only in specifying the content and placement of the unused portion of the single Cray computer word containing the constant.

Hollerith Constant	Internal Representation (64-bit Cray computer word)							
(bit position)	(0-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63)
6HCRAY-1	C	R	A	Y	-	1	(040_8)	(040_8)
'CRAY-1'H	C	R	A	Y	-	1	(040_8)	(040_8)
"CRAY-1"H	C	R	A	Y	-	1	(040_8)	(040_8)
6LCRAY-1	C	R	A	Y	-	1	(000)	(000)
'CRAY-1'L	C	R	A	Y	-	1	(000)	(000)
"CRAY-1"L	C	R	A	Y	-	1	(000)	(000)
6RCRAY-1	(000)	(000)	C	R	A	Y	-	1
'CRAY-1'R	(000)	(000)	C	R	A	Y	-	1
"CRAY-1"R	(000)	(000)	C	R	A	Y	-	1

A Hollerith constant is limited to a maximum of eight characters except when specified in a CALL statement, a function argument list, or a DATA statement. In a DATA statement, Hollerith constants longer than eight characters are permitted only when an array is being initialized in its entirety. For example,

```
DIMENSION IARRAY (2)
DATA IARRAY /16HABCDEFGHIJKLMNOP/
```

is permitted, but

```
DIMENSION JARRAY (2)
DATA JARRAY(1),JARRAY(2) /16HABCDEFGHIJKLMNOP/
```

is illegal. All Hollerith constants with R suffixes are limited to a maximum of eight characters. A zero word follows the last word containing a Hollerith constant specified as an actual argument in an argument list.

The forms E'*string*' and E"*string*" are reserved for future EBCDIC constants.

NAMELIST Hollerith constants are specified by the following forms.

```
    nH...

    nL...

    nR...

          H
    '...' L
          R

          H
    "..." L
          R
```

If the R form is used, the string must contain eight or less characters. Within the ' or " delimited format, a ' or " is specified as '' or "", respectively.

E.1.1 HOLLERITH EXPRESSIONS

Hollerith expressions contain no operators and only a single operand. A Hollerith expression is evaluated to yield a sequence of characters. Its value is that sequence. The forms of a Hollerith expression appear below.

- A Hollerith constant
- Name of a variable containing Hollerith data
- Name of an array element containing Hollerith data
- Name of a function providing Hollerith data when referenced

A Hollerith constant comprising a Hollerith expression is limited to eight characters.

The data type of the name referencing a variable or array element containing Hollerith data can affect its evaluation during program execution. A variable or array element of type integer or real contains eight Hollerith characters. A variable or array element of type complex or double precision cannot contain Hollerith characters. A variable or array element of type logical cannot contain Hollerith characters except when it has been initialized in a DATA statement.

Hollerith data provided when a function is referenced contains as many characters as a variable or array element of corresponding type.

When used in arithmetic or relational expressions, Hollerith expressions are considered to be type Boolean.

E.1.1.1 Hollerith relational expressions

The form of a Hollerith relational expression is

$$\boxed{e_1 \; \textit{relop} \; e_2}$$

e_1 and e_2
Hollerith expressions

relop Relational operator

A Hollerith relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator; false if they do not.

The Hollerith expression e_1 is considered less than e_2 if its value precedes the value of $e(2)$ in the collating sequence or is considered greater if its value follows the value of e_2 in the collating sequence.

Examples:

The following are evaluated as true if the integer variable LOCK contains the Hollerith characters K, E, and Y in that order and left-justified with five trailing blank character codes.

```
3HKEY.EQ.LOCK
'KEY'.EQ.LOCK
LOCK.EQ.LOCK
'KEY1'.GT.LOCK
'KEY0'H.GT.L0CK
```

Two Hollerith expressions are equivalent if their values are equal for all possible values of their specification.

E.1.2 HOLLERITH FORMAT SPECIFICATION

A format specification can be an array name of type integer, real, or logical.

The leftmost characters of the specified entity must contain Hollerith data constituting a format specification when the statement is executed.

The format specification must begin with a left parenthesis and end with a right parenthesis. Data can follow the right parenthesis ending the format specification and have no effect. Blank characters can precede the format specification.

E.2 TWO-BRANCH ARITHMETIC IF STATEMENTS

The form of a two-branch arithmetic IF statement is as follows.

Format:

$$\boxed{\text{IF}(e)s_1,s_2}$$

e An integer, real, or double-precision expression

s_1,s_2 Statement labels of executable statements appearing in the same program unit as the two-branch arithmetic IF statement

Execution of a 2-branch arithmetic IF statement causes evaluation of the expression e. Control is transferred to the statement identified by s_1 if e is nonzero or to the statement identified by s_2 if e is zero.

Example:

```
    IF (I+J*K) 100,101
```

E.3 INDIRECT LOGICAL IF STATEMENTS

The form of an indirect logical IF statement is as follows.

Format:

IF(*e*)s_1,s_2

e Logical expression

s_1,s_2 Statement labels of executable statements appearing in the same program unit as the indirect logical IF statement

Execution of an indirect logical IF statement causes evaluation of the expression *e* for a logical value followed by a transfer of control. If the value of *e* is true, the statement identified with statement label s_1 is executed next. If the value of *e* is false, the statement identified with statement label s_2 is executed next.

Example:

```
IF(X.GE.Y)148,9999
```

E.4 FORMATTED DATA ASSIGNMENT

Formatted data assignment operations define entities by transferring data between input/output list items and internal records. Like other assignment statements, formatted data assignment statements only perform internal data transfers. Like formatted input/output statements, formatted data assignment statements specify an input/output list and invoke format control during their operations.

The two formatted data assignment statements are ENCODE and DECODE.

E.4.1 ENCODE AND DECODE STATEMENTS

The forms of the ENCODE and DECODE statements are:

ENCODE (*n*,*f*,*dent*)[*elist*]

DECODE (*n*,*f*,*sent*)[*dlist*]

n Number of characters to be processed, specified by a nonzero integer expression not to exceed 152

f FORMAT identifier, except for an asterisk

dent Symbolic name of a destination variable, array element, or array where the *n* characters of *elist* are packed (eight per word) by ENCODE

sent Symbolic name of the source variable, array element, or array where characters are unpacked and stored into *dlist* by DECODE

elist and *dlist*
Lists specified the same as for formatted input/output statements. *elist* is the list of items written to the destination entity; *dlist* is the list of items receiving the source entity.

E.4.1.1 The ENCODE statement

The ENCODE statement produces a sequence of *n* characters (packed eight per word) from values contained in the input list items specified in *elist* under control of the format specification identified by *f*. The character sequence is stored into a variable, array element, or array identified by *dent*.

If *n* is not an integer multiple of eight, the last word in each record is padded with spaces to a word boundary. In effect, *n* is rounded up to be a multiple of eight.

Example:

elist:	array ZD(5):	ZD(1) = 'THISbbbb'
		ZD(2) = 'MUSTbbbb'
		ZD(3) = 'HAVEbbbb'
		ZD(4) = 'FOURbbbb'
		ZD(5) = 'CHARbbbb'
f:	FORMAT (5A4)	
n:	20	
dent:	array ZE(3)	

The sequence

```
      ENCODE (20,1,ZE)ZD
  1   FORMAT (5A4)
```

produces

```
    dent =        ZE(1) = 'THISMUST'
                  ZE(2) = 'HAVEFOUR'
                  ZE(3) = 'CHARbbbb'
```

E.4.1.2 The DECODE statement

The DECODE statement processes a sequence of *n* characters (packed eight per word) contained in the variable, array element, or array identified by *sent* under control of the format specification identified by *f*. The resulting values define the input list items specified in *dlist*.

If *n* is not an integer multiple of eight and the DECODE format calls for more than one DECODE record, the second and all subsequent DECODE records begin on a word boundary. In effect, *n* is rounded up to be a multiple of eight.

Example:

```
    sent:         ZE:               ZE(1) = 'WHILETHI'
                                    ZE(2) = 'SbHASbbF'
                                    ZE(3) = 'IVEbbbbb'

       n: =       20

       f:         FORMAT (5A5)
```

The sequence

```
        DECODE (20,2,ZE)ZD
    2   FORMAT (4A5)
```

produces

```
    dlist =       ZD(1) = 'WHILEbbb'
                  ZD(2) = 'THISbbbb'
                  ZD(3) = 'HASbbbbb'
                  ZD(4) = 'FIVEbbbb'
```

E.5 EDIT DESCRIPTORS

The formats of obsolete edit descriptors are:

$*h_1h_2\ldots h_n*$ (asterisk)

```
 ________
|        |
| [-b]X  |
|________|
```

* or X   Asterisk and the X indicate the manner of editing

h_i Any ASCII character listed in appendix A as capable of internal representation

b   Any nonzero, unsigned integer constant

Examples:

AN ASTERISK EDIT DESCRIPTOR

-55X (moves current position 55 spaces to left)

E.6 DOUBLE DECLARATION STATEMENTS

The form of the double declaration type statement is:

```
 __________________
|                  |
| DOUBLE v[,v]...  |
|__________________|
```

DOUBLE   Specifies the desired data type

v   Constant, variable, array, function, or dummy procedure name, or an array declarator

The form of the double declaration FUNCTION statement is:

```
 ___________________________________
|                                   |
| DOUBLE FUNCTION fun[([d[,d]...])  |
|___________________________________|
```

fun   Symbolic name of the function subprogram in which the FUNCTION statement appears

d   Dummy argument representing a variable, array, or external procedure name

E.7 DATA STATEMENT FEATURES

An *nlist* entity of type logical can correspond to a *clist* constant of type Hollerith.

One constant must exist for each element of an array whose name appears in the list without subscripting unless named as the last item of an *nlist*. In this case, the values in *clist* can specify any number of consecutive array element values, beginning with the first.

A character constant can be specified to correspond to entities of any type except logical.

If a variable, an array element, or an entity associated with either is defined by a DATA statement more than once in an executable program, the one nearest the end of the program is the only definition to apply.

E.8 PUNCH STATEMENT

The PUNCH statement is a data transfer output statement.

Format:

PUNCH *f* [,*iolist*]

f	Format identifier
iolist	Input/output list specifying the data to be transferred

E.9 TYPE STATEMENT DATA LENGTH

The forms of the type statements with data length included are:

type [*n] v[,v]...

IMPLICIT *type*[*n](a_1[-a_n][,a_1[-a_n]]...)[,*type*[*n](a_1[-a_n]
[,a_1[-a_n]]...)]...

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

*n Specifies the data length as shown in table E-1 (INTEGER*2 should be used with caution since it implies 24-bit integers)

v Constant, variable, array, function, or dummy procedure name, or an array declarator

a Single letter or is a range of single letters denoted by the first and last letter of the range separated by a hyphen. Writing a range of letters (a_1 through a_n) has the same effect as writing a list of the single letters (a_1,a_2,...a_n) and where a_1 precedes a_n in this alphabetically ordered sequence.

Table E-1. Data Length

data \ *type* \ *n*	1	2	4	8	16
INTEGER		24-bit integer	64-bit integer		
REAL			64-bit real single precision		128-bit real d'ble prec.
COMPLEX			64-bit complex single precision		
LOGICAL		64-bit logical			
DOUBLE PRECISION					128-bit real d'ble prec.
Any other data length gives a fatal error.					

E.10 EXTENDED RANGE OF A DO LOOP

Transfer of control out of the range of a DO loop does not inactivate the DO loop. However, the DO loop becomes inactive if the DO variable becomes undefined or is redefined while outside the range. If the DO loop remains active, control can be transferred to any statement in the loop.

E.11 NONCHARACTER ARRAYS FOR FORMAT SPECIFICATION

No FORMAT statement is required if the format identifier in a formatted input/output or formatted assignment statement is a noncharacter array name. The initial and following elements of that array must be defined with character data that constitute a format specification when the input/output statement is executed. The opening parenthesis must be in the first or ninth character position (the first character in the first or second word). If in the ninth character position, the content of the first word has no effect on program execution.

E.11.1 EOF, IEOF, AND IOSTAT FUNCTIONS

If an end-of-file condition occurs during a READ statement not containing a end-of-file specifier (END=) or an I/O status specifier (IOSTAT=), execution of the program continues. A reference to the EOF, IEOF, or IOSTAT function must occur before the next attempted read from that dataset.

The ANSI FORTRAN Standard does not provide continued execution of a program following an end-of-file encountered during a READ statement not containing an end-of-file specifier or an I/O status specifier.

F. CREATING NON-FORTRAN PROCEDURES

Function and subroutine subprograms written in languages other than FORTRAN can be used with CFT programs. Cray Assembly Language (CAL), Cray C, and Cray Pascal can be used with FORTRAN programs.

F.1 CAL

Specific macros are available to aid the CAL programmer in writing routines to be used with CFT. These macros maintain compatibility with different versions of CFT. The following is a list of some of the CAL linkage macros and their purpose:

- DEFARG defines argument transmission
- DEFB defines B register use
- DEFT defines T register use
- ENTRY defines FORTRAN-callable entry points
- ARGADD allows argument retrieval
- LOAD allows local variable reference
- STORE allows local variable updates

See the Macros and Opdefs Reference Manual, CRI publication SR-0012, for more information on linkage macros.

F.2 CRAY PASCAL

There are several important differences between the Pascal and FORTRAN languages that must be taken into account in order to use Pascal routines with CFT.

- Because all FORTRAN arguments are passed by address, all arguments in Pascal must be declared in VAR statements.
- Pascal and FORTRAN store arrays differently. Multidimensional arrays defined in one language are transposed in the other.

- Cray Pascal is a stack-based language. The Cray Pascal Compiler uses the standard Cray stack management routines, so FORTRAN routines also using the stack will not have a problem. A FORTRAN program executing in static mode that calls a Pascal routine will have a stack allocated for the Pascal routine.

- If a Pascal routine performs any I/O with the default input or output files and the main routine is not a Pascal routine, the Pascal routine must reset (for input) or rewrite (for output) appropriately.

- A reference to a Pascal procedure name is a level of indirection more than a reference to a FORTRAN subroutine name, when the name is specified as an argument. For example, the following FORTRAN and Pascal programs pass different information to B.

```
SUBROUTINE A
EXERNAL P
CALL B(P)
END
SUBROUTINE B(P)
EXTERNAL P
CALL P
END


program A;
     procedure B(procedure P);
     begin;
     end;
     procedure P;external;
begin
     B(P)
end;
```

In FORTRAN, the parcel address of P is passed to subroutine B. In Pascal, the parameter descriptor containing the parcel address of P is passed to procedure B.

F.3 CRAY C

There are several important differences between the C and FORTRAN languages that must be taken into account in order to use C language routines with CFT.

- Because all FORTRAN parameters are passed by address, all parameters in the C routines must be declared as "pointer to." This is true for all single-word items. Arrays and structures declared in C are correctly passed by address.

- FORTRAN arrays and C arrays have different storage allocation. Multidimensional arrays defined in one language are transposed in the other.

- The C character pointer is incompatible with the FORTRAN CHARACTER type. It is not recommended that character values be passed between C and FORTRAN routines.

- Cray C is a stack-based language. The Cray C Compiler uses the standard Cray stack management routines, so FORTRAN routines also using the stack will not have a problem. A FORTRAN program executing in static mode that calls a C routine will have a stack allocated for the C routine.

See the Cray C Differences Reference Manual, CRI publication SR-2024, for more information about using C routines in FORTRAN programs.

G. SYMBOLIC DEBUG PACKAGE

The symbolic debug package provides, upon request, a symbolic memory dump. This dump gives variable names and values in a format appropriate to the variable type. It is invoked by specifying the CFT Z option, ON=Z, in the CFT control statement and by including the DEBUG control statement or by calling the library routine SYMDEBUG.

The DEBUG control statement is conventionally used after EXIT and DUMPJOB control statements as an aid in determining the cause of a job abort.

Example:

```
    JOB .... .
    CFT,ON=Z.
    SEGLDR,GO.
    EXIT.
    DUMPJOB.
    DEBUG.
```

The library routine SYMDEBUG is callable from a running program and provides a debug printout of the job's memory. SYMDEBUG has one argument, a Hollerith string, which can contain any of the parameters that DEBUG accepts.

Example:

```
    CALL SYMDEBUG('O=OUT,PAGES=17.')
```

The string must terminate with a period.

See the Symbolic Debugging Package Reference Manual, CRI publication SR-0112, for details of the optional DEBUG parameters.

H. UNBLOCKED DATASETS

Unblocked datasets do not conform to standard Cray operating system COS dataset format. An unblocked dataset must be explicitly declared with the U parameter on a COS ASSIGN statement.

Formatted input/output is prohibited on an unblocked dataset. For unformatted I/O, an implied DO loop in the input/output list (*iolist*) is not permitted. For synchronous I/O, each item in the *iolist* must be an array name without subscripts. In addition, each array dimension must be a multiple of 512.

ENDFILE and BACKSPACE requests are illegal for an unblocked dataset. All other auxiliary I/O operations are permitted on an unblocked dataset.

See the COS Version 1 Reference Manual, publication SR-0011, for detailed descriptions of the ASSIGN control statement, unblocked and blocked dataset structures, and logical I/O for both structures.

I. REPRIEVE PROCESSING

Reprieve processing suspends normal system error processing and allows the user to attempt to recover from what normally would be an abort condition. The user selects the conditions under which recovery occurs. (See the COS Version 1 Reference Manual, publication SR-0011, for a complete list of reprievable abort conditions and selection codes.) Reprieve processing allows recovery from a time limit, for example, and can save important data on disk before the job aborts.

I.1 REPRIEVE INITIATION

Reprieve processing is set up by a call to the library routine SETRPV. This call can appear anywhere and any number of times in a program but must be executed before the abort condition occurs. The call typically appears at the beginning of a main program.

Format:

```
CALL SETRPV(recname,xpsave,class)
```

recname Name of the external subroutine to be called if a reprievable error occurs. *recname* must be declared in an EXTERNAL statement.

xpsave Array with dimensions of at least 40. On entry to *recname*, *xpsave* contains a copy of the Exchange Package at the time of the abort and information about the type of the abort. (See the COS Version 1 Reference Manual, publication SR-0011, for a description of *xpsave*.) *xpsave* must be in a common block in the routine that calls SETRPV and in *recname*.

class Mask that defines recoverable errors. Valid mask values are listed in appendix F of the COS Version 1 Reference Manual, publication SR-0011.

Within *recname*, any FORTRAN statements except RETURN or END can be executed.

I.2 REPRIEVE TERMINATION

The normal FORTRAN method of terminating reprieve processing is to call the subroutine ENDRPV from the external subroutine set up by SETRPV.

Format:

```
 _______________
|               |
| CALL ENDRPV   |
|_______________|
```

The execution of this call ends the job step. Job processing resumes at the next EXIT control statement or it terminates if no EXIT is present.

A STOP or CALL EXIT can terminate the job step and resume execution at the next job control statement. Using END or RETURN to terminate the job step gives unpredictable results.

J. FTREF UTILITY

The FTREF utility generates a report about common block variable usage in the subroutines of a user application on a global basis. FTREF also provides tabular information, which includes entry names, calling routines, and the called routines for each subroutine. FTREF displays this information as a static calling tree. If the user program is multitasked, FTREF states whether a common variable is locked or unlocked when it is referenced or redefined.

FTREF is invoked by specifying ON=XS in the CFT control statement and by including the FTREF control statement. The input file to FTREF should contain as many modules used by the application as possible for the best results.

Example:

```
    JOB, .... .
    ACCOUNT, .... .
    CFT,ON=XS,L=OUT.
    CFT,ON=XS,L=OUT.
    CFT,ON=XS,L=OUT.
    FTREF,I=OUT,CB=FULL,TREE=FULL.
        .
        .
        .
```

See the COS Version 1 Reference Manual, publication SR-0011, for details of the FTREF control statement.

INDEX

INDEX

- -a parameter in UNICOS CFT command, 1-13
- -A parameter in UNICOS CFT command, 1-16
- A (alphanumeric) editing, 8-27
- A edit descriptor, 8-27
- Abort
  - compilation, 1-19
  - condition, recovery from, I-1
  - job, 1-11,18
  - NAMELIST, reasons for, 7-30
- ABORT system function, C-2
- ABS function, B-2
- Absolute value function, B-2
- ACCESS specifier, 7-21
  - for OPEN, 7-26
- Access
  - direct, 7-7
  - sequential, 7-7
- ACOS function, B-4
- Active DO loop, 6-8
- Actual argument, 2-3,6,8,21, 4-12
  - for external procedures, 2-22
- Actual array, 4-4
  - declarator, 4-8
  - element, 4-4
- Addition, 5-5
- Address
  - field, 1-22
  - of current record, return, C-4
  - of pointee, 3-14
- Adjustable array, 4-8,13
  - declarator, 4-7,8
- Adjustable-size array, 4-4
- AIDS=*aids* control statement parameter, 1-2
- AIMAG function, B-3
- AINT function, B-2
- ALIGN compiler directive, 1-43
- ALLOC=*alloc* control statement parameter, 1-2
- Allocation of variables to storage, 1-10
- Allowed
  - assignment statements, 5-4
  - Logical and Masking Operations and Result Types, 5-21
- ALOG function, B-6
- ALOG10 function, B-6
- Alphanumeric editing, 8-27
- Alternate return, 2-14,18
- AMAX0 function, 2-25, B-6
- AMAX1 function, 2-25, B-6
- Ambiguous vectors, 1-27
- AMIN0 function, 2-25, B-6
- AMIN1 function, 2-25, B-6
- AMOD function, B-2
- AND function, B-9
- ANINT function, B-2
- ANSI control statement parameter, 1-2
- Apostrophe edit descriptor, 8-12
- Apostrophe editing, 8-12
- Arccosine function, B-4
- Arcsine function, B-4
- Arctangent function, B-4
- ARGADD, CAL linkage macro, F-1
- ARGPLIMQ directive, 1-39
- Arguments, 2-20
  - number of, function, B-11
  - type character, 3-12
- Arithmetic
  - assignment statement, 5-3
  - constants, 4-1
  - constant expression, 5-3
  - expressions, 5-2,6,7
  - IF statement, 6-3,11
  - IF statement, two-branch, E-6
  - operand, 5-6
  - operator, 5-4
  - relational expressions, 5-16
  - value, 3-1
- Arrays, 3-1, 4-3
  - as arguments, 4-12
  - bounds checking directive, 1-41
  - declarator, 3-3, 4-3,5
  - differences between C and CFT, F-3
  - dimension, 3-3, 4-3,9
  - element, 3-1,2, 4-3,9
    - in vector and scalar modes, 9-7
    - name, 4-3,21
    - reference, 5-6
    - substring, 4-4
    - and subscripts, 4-7
  - name, 3-3, 4-3,21
    - in an I/O list, 7-12
  - size, 4-11
  - specification and size, 4-8
  - storage, 4-17
  - stored differently in Pascal than CFT, F-1
  - subscripts and storage sequence, 4-9
- ASCII character, 3-10
  - codes, 1-20
  - set, A-1
- ASIN function, B-4
- ASSIGN to connect a dataset for random acess, 7-39
- Assigned GOTO
  - statement, 6-3,11
  - inhibits vectorization, D-31

- Assignment, 5-1
  - statements, 1-33, 5-1,4
- Association, 4-1
  - of arguments, 2-21
  - of entities, 4-19
- Associative law, 5-6
- Assumed-size array, 4-4,5
  - declarator, 4-8
  - dummy array, 4-11
- Asterisks written on field overflow, 8-18
- ATAN function, B-4
- ATAN2 function, B-4
- Auxiliary character set, A-1

- -b parameter in UNICOS CFT command, 1-14
- B registers, 1-26
- B=*bdn* control statement parameter, 1-2
- BACKSPACE statement, 7-17, 7-18
  - unblocked datasets, H-1
- Backward branches, 9-2
  - inhibit vectorization, D-31
- Basic real constant, 3-6
- Bidirectional memory, 1-3, 1-17, 9-20
  - hazards, 1-34
- Binary
  - load modules, 1-2
  - object file, 1-18
  - output, 1-14
  - program, 1-1
- BL compiler directive, 1-40
- BL optimization option, 1-8, 1-14
- BLANK specifier
  - for INQUIRE, 7-22
  - for OPEN, 7-26
- Blanks
  - ignore, 8-18
  - in list-directed input, 8-3
  - treat as zeros, 8-18
- $BLD, 1-13
- Block begins messages, 1-21
- BLOCK compiler directive, 1-42
- BLOCK DATA statement, 2-20
- Block data subprogram, 2-9, 2-19
- Block field, 1-24
- Block IF statement, 6-6
- Blocked files, I/O for, 7-2
- BN edit descriptor, 8-18
- BN editing, 8-18
- Boolean
  - functions, B-9
  - results, 3-8
  - type, 3-8
  - values, 3-8
  - variable, 5-4
- Bottom loading, 1-40,41
- Bounds checking, 1-41
- BOUNDS compiler directive, 1-12, 1-41
- Branch instruction, 1-8, 1-40
- Branch statement inhibiting vectorization, D-31
- Breakpoints, 1-4
- BTREG optimization option, 1-8, 1-14
- BUFFER IN, BUFFER OUT on COS random datasets, 7-40
- BUFFER IN statement, 7-33
- BUFFER OUT statement, 7-33
- Buffered I/O with formatted records, 7-3
- BZ edit descriptor, 8-18
- BZ editing, 8-18

- -c parameter in UNICOS CFT command, 1-14
- -C parameter in UNICOS CFT command, 1-16
- C procedures, F-2
- C=*cdn* control statement parameter, 1-3
- CABS function, B-2
- CAL, F-1
  - assembler, 1-3
  - listing, 1-14
  - procedures, calling, F-1
- CALL statement, 2-14
  - and Hollerith constants, E-3
- Call-by-value sequence, 1-35, 1-46
- Carriage control, modify, 8-15
- Categories of compiler directives, 1-29
- CAUTION message, definition, D-1
- CCOS function, B-4
- CEXP function, B-5
- CFT
  - command, 1-13
  - compiler, 1-1
  - control statement, 1-1
  - input/output statements, 7-1
  - revision level, 1-21
  - utilities, C-2
  - utility procedures, C-2
- Changing I/O buffer lengths
  - using SEGLDR directives, 7-41
  - by regenerating $IOLIB, 7-41
- CHAR function, 2-25, B-8
- Character
  - array, 3-12
  - assignment statement, 5-13
  - constant, 3-10
  - dummy argument, 3-13
  - expression, 5-13,16
    - evaluation, 5-14
  - function, 3-10, B-7
  - information edit descriptor, 8-12
  - items, list-directed input, 8-3
  - limit in I/O list and format specification, 7-16
  - output constants, 8-4
  - print control, 7-16
  - relational expressions, 5-16
  - set, A-1,2
  - storage unit, 4-17
  - string, 3-10
  - substring, 3-12
  - -to-integer function, B-8
  - type statement, 3-11
  - value, 3-10
  - variables, 3-12
  - variables and function references, inhibiting vectorization, D-32
- CHARACTER statement, 3-11
- CHARACTER type, 3-10
- Cilist parameter, I/O statements, 7-10

Circular shift function, B-10
CIV see Constant increment variables
CLEARFI subroutine, C-3
CLEARFIS subroutine, C-3
Cllist parameter, CLOSE statement, 7-24
CLOCK function, B-11
Clock register content, return, C-4
Clock, real-time, function, B-11
CLOG function, B-6
CLOSE Specifiers and Their Meanings, 7-27
CLOSE statement, 7-24
CMPLX function, 2-25, B-8
CODE compiler directive, 1-31
Code, 1-11
    generation block, 1-18
    movement, 1-40
    scheduler, 1-40
Colon editing, 8-15
Comment between input NAMELIST group records, 7-29
COMMENT message, definition, D-1
Common block, 1-2,24, 2-3, 3-2, 4-3,17,22
    extended memory, 4-25
    names, 1-11,18, 4-24
    size, 4-25
    storage sequence, 4-24
    variable usage, reporting, J-1
COMMON keyword, 4-26
Common logarithm function, B-6
COMMON statement, 4-5,19,23
COMMONS SEGLDR directive, 7-41
Compilation errors, 1-18
Compile-time messages, D-1, D-2
Compiler directive lines, 1-11,18,29
Compiler directives, 1-28
Compiler options, 1-6,21,28
    under COS, 1-12
    under UNICOS, 1-18
Compiler-generated variables, 1-10,11
COMPL function, B-9
Complement function, B-9
Complex
    constant, 3-7
        as NAMELIST variables, 7-30
    editing, 8-25
    expression, 5-16
    items, list-directed input, 8-3
    output constants, 8-4
    type conversion, 5-12
    value, 3-7
COMPLEX type, 3-3,7
Compressed index gather/scatter hardware, 1-3,17
Compressed index hardware, 1-3,17
Compressed index reference, 9-19
Computed GOTO statement, 6-2
    inhibits vectorization, D-33
Concurrent I/O, 7-33
Conditional
    block, 6-5, 6-11
    GOTO statement, 6-1
    replacement statements, 1-6,7,15,44
    statements, 9-16
    vector loops, 1-14, 9-12
Conditions inhibiting vectorization, 9-2
CONJG function, B-3
Conjugate of complex value function, B-3
Connecting a file, 7-24
Considerations in evaluating arithmetic expressions, 5-12
Constant array declarator, 4-8
Constant increment variables, 1-6,15, 9-3
    defined more than once in a loop, D-32
    equivalencing with an array, D-33
    zero increment, D-38
Constant, 3-1,3, 4-1, 5-1
    Hollerith, E-2
Constant-size array, 4-4
CONTINUE statement, 6-2, 6-4, 6-11
Control
    characters, print, 7-16
    information list, 7-10
        examples, 7-11
    statement, 1-1
Conversion
    to complex function, B-8
    to double-precision function, B-8
    to integer function, B-7
    to real function, B-8
COS, 1-1
    CFT control statement, 1-1
    dataset, 7-4
    error messages, 1-20
    function, B-4
    select, C-2
COSH function, B-5
Cosine function, B-4
COT function, B-4
Cotangent function, B-4
CPU time for job function, C-4
CPU=*cputype*:*cpuchar* control statement parameter, 1-3
Cray Assembly Language (CAL), 1-3
CRAY C, F-2
Cray FORTRAN, 1-1
    programming, 9-1
CRAY Pascal, F-1
CRAY scalar merge function, B-12
CRAY X-MP Computer System, 1-1
CRAY-1 Computer System, 1-1
Creating a dataset
    for random access, 7-38
    to be randomly accessed, 7-38
Creating non-FORTRAN procedures, F-1
Cross-reference information, 1-27
CSIN function, B-4
CSMG (CRAY scalar merge) function, B-10,12
CSQRT function, B-5
Current interrupt mode, determine, C-3
Current record, return address, C-4
CVL compiler directive, 1-45
CVL optimization option, 1-9,14
CVMGM function, B-12
CVMGN function, B-12
CVMGP function, 9-15, B-12
CVMGT function, 9-15, B-12
CVMGZ function, B-12

- -d parameter in UNICOS CFT command, 1-14
- D edit descriptor, 8-23
- D editing, 8-23
- DACOS function, B-4
- DASIN function, B-4
- Data
  - length, E-12
    - of type statement, E-11
    - table, E-12
  - specification statements, 3-2
  - statement, 1-2,3, 4-14
    - and Hollerith constants, E-3
    - features, E-11
  - structures, 4-1
  - transfer operations, 7-14
  - transfer statement, 7-9
  - transfer, formatted, 7-15
  - transfer, unformatted, 7-15
  - types, 3-1,2
    - in DATA statement, 4-16
    - in Exponentiation, 5-10
    - in Relational Operations, 5-15
    - of arithmetic expressions, 5-8
    - with correct edit descriptors, 8-9
  - using and storing, 4-1
- DATAN function, B-4
- DATAN2 function, B-4
- Dataset, 7-4
  - connection, 7-39
  - for random access, creating, 7-38
  - names, 1-1
  - unblocked, H-1
- DATE function, B-11
- DBLE function, 2-25, B-8
- DCOS function, B-4
- DCOSH function, B-5
- DCOT function, B-4
- DDIM function, B-3
- DEBUG compiler directive, 1-45
- DEBUG control statement parameter, 1-4
- DEBUG control statement, G-1
- Debug Symbol Table, 1-4,13,18,19
- Debugging directives, 1-45
- Debugging instructions, 1-22
- Decimal normalization control on I/O, 8-17
- DECODE statement, E-7, E-9
- DEFARG, CAL linkage macro, F-1
- Defaults, controlling NAMELIST input, 7-30
- Defaults, controlling NAMELIST output, 7-33
- DEFB, CAL linkage macro, F-1
- Defined array element, 4-18
- Defined entity, 4-17
- Defined variable, 4-18
- DEFT, CAL linkage macro, F-1
- Delimiter, NAMELIST ($ or &), 7-30
- Dependencies, 9-5
- Dependency information combinations, 9-8
- Dependency messages, 9-9, D-38
- DEXP function, B-5
- Difference, positive, function, B-3
- DIM function, B-3
- Dimension bound expression, 4-6
- Dimension declarators, 3-3, 4-6
- DIMENSION statement, 3-15, 4-5
- DINT, B-2
- Direct access of external files, 7-6
- Direct access, 7-7
- DIRECT specifier for INQUIRE, 7-22
- Directive Are Used, 8-10
- Division, 5-5
- DLOG function, B-6
- DLOG10 function, B-6
- DMAX1 function, 2-25, B-6
- DMIN1 function, 2-25, B-6
- DMOD function, B-2
- DNINT function, B-2
- DO control variable, 1-10
- DO loop, 1-5,7,12,27,32, 6-7
  - active, 6-8
  - extended range of, E-13
  - transfer out, E-13
  - vectorizable, 9-1
- DO statement, 6-7, 6-8, 6-11
- DO variable, 6-7,8
  - in an implied DO list, 7-15
- DO-Loop Table, 1-11,18
- DO-loop unrolling, 1-10,16,46
- Dollar sign editing, 8-15
- Double declaration statements, E-10
- Double precision, 1-12,19
  - constant, 3-7
  - editing, 8-23
  - exponent, 3-7
  - items, list-directed input, 8-3
  - octal output, 8-25
  - output constants, 8-4
  - product function, B-3
  - type conversion, 5-12
- DOUBLE PRECISION type, 3-3,7
- DPROD function, B-3
- DSIGN function, B-2
- DSIN function, B-4
- DSINH function, B-4
- DSQRT function, B-5
- DTAN function, B-4
- DTANH function, B-5
- Dummy and actual arrays, 4-4
- Dummy argument, 1-24, 2-4,13,14,21,23, 4-4
- Dummy array, 4-12
  - declarator, 4-8
- Dummy procedure, 2-10,24
- Dump, symbolic memory, G-1
- DUMPJOB subroutine, C-4
- Dynamic common block, 1-40
- Dynamic common block directive, 1-40
- DYNAMIC compiler directive, 1-40

- -e parameter in UNICOS CFT command, 1-14
- -E parameter in UNICOS CFT command, 1-17
- E edit descriptor, 8-21
- E editing, 8-21
- E=*eml* control statement parameter, 1-4
- Edit descriptor, 8-6, E-9
  - A, 8-27
  - apostrophe, 8-12
  - BN, 8-18
  - BZ, 8-18
  - colon, 8-15

- Edit descriptor (continued)
  - D, 8-23
  - dollar sign, 8-15
  - E, 8-21
  - F, 8-19
  - G, 8-23
  - H, 8-12
  - I, 8-18
  - L, 8-26
  - O, 8-25
  - P, 8-16
  - quotation mark, 8-12
  - R, 8-28
  - S, 8-18
  - slash, 8-14
  - SP, 8-18
  - SS, 8-18
  - Z, 8-25
- Edit Descriptors with Data Types, 8-9
- Edit descriptors, obsolete, E-9
- Editing input data, 7-6
- EDN=*edn*, 1-5
- EJECT compiler directive, 1-30
- ELSE statement, 6-7
- ELSE-block, 6-5
- ELSEIF statement, 6-6
- ELSEIF statements, inhibiting vectorization, D-34
- ELSEIF-block, 6-5
- Embedded format specification, 8-10
- Empty record, write, 8-14
- ENCODE statements, E-7, E-8
- END statement, 2-3, 2-9, 2-13, 6-13
- End-of-data record, 7-4
- End-of-file
  - condition, 7-16
  - identifier, 7-11
  - record, 7-2
- END= parameter, 7-11
- Endfile record, 7-2
- ENDFILE statement, 7-3, 7-17, 7-18
- ENDFILE with unblocked datasets, H-1
- ENDIF statement, 6-6
- ENDRPV subroutine, C-4, I-2
- Enter message in logfile, C-2
- Entity categories, 9-3
- Entry point address, 2-11
- ENTRY statement, 2-16, 4-3
- ENTRY, CAL linkage macro, F-1
- EOD record, 7-4
- EOD, detect, C-3
- EODW, C-3
- EOF
  - condition after reading an endfile record, 7-17
  - function, C-3, E-13
  - detect, C-3
  - written by ENDFILE, 7-17
- EQUIV directive of SEGLDR, 8-10
- Equivalence function, B-9
- EQUIVALENCE statement, 4-4,19,20
- EQV function, B-9
- ERR specifier, 7-20
  - for CLOSE, 7-27
  - for OPEN, 7-25
- ERR= parameter, 7-11
- ERREXIT subroutine, C-2
- Error
  - condition, 7-16
    - during data transfer, 7-16
    - during INQUIRE, 7-23
  - during I/O, 7-40
  - exit processing, resume, C-4
  - identifier, 7-11
  - listing, 1-5,17,18
  - message, definition, D-1
  - messages, 1-4,20
  - recovery, I/O, 7-40
- Establishing a format, 7-14
- Exchange Package, saving a copy, I-1
- Executable program, 2-1,3
- Executing a DO statement, 6-8
- Execution speed, 9-1
- EXIST specifier, 7-21
- EXIT statement, with CFT job, D-28
- EXIT subroutine, C-2
- EXP function, B-5
- Exponent function, B-5
- Exponential functions, B-5
- Expressions, 5-1
- Extended memory
  - addressing, 1-3,17, 4-25
  - common blocks, 4-25
- Extended range of DO loop, E-13
- External file ,opening, 7-23
- External file, 7-2
  - access methods, 7-6
  - identifiers, 7-9
  - attribute, determining status, 7-19
- External
  - function, 2-2,5,8,11
    - references, inhibit vectorization, D-34
  - routines, 1-46
  - subprogram, 1-42
- EXTERNAL statement, 2-8, 2-10, 2-24, 2-26

- F edit descriptor, 8-19
- F editing, 8-19
- Factor, 5-7
- False logical value, 3-10
- Fast integer multiply and divide algorithms, 1-7
- FASTMD
  - compiler directive, 1-36
  - optimization option, 1-7,15
- Fatal error, 1-4,11,14
  - messages, 1-12,19
- File, 7-4
  - access methods, 7-6
  - and record positioning for buffered I/O, 7-36
  - closing, 7-24
  - identifier, 7-8
  - input/output, 7-4
  - internal, 7-5
  - opening, 7-23
  - parameter, INQUIRE statement, 7-19

- File (continued)
  - position after data transfer, 7-7
  - specifier for OPEN, 7-2
- FLOAT function, 2-25, B-8
- Floating-point
  - arithmetic, 1-5
    - operations, 1-46
  - error, 1-5
  - functional unit, 9-1
  - interrupts, prohibit, C-3
  - interrupts, permit, C-3
  - operations, 1-13,19
  - output format, 8-19
  - results, 1-16
  - values, 3-6
- FLODUMP utility, 1-38
- FLOW compiler directive, 1-37
- Flow trace directives, 1-37
- FLOWLIM directive, 1-39
- FLOWTRACE option, 1-11
- FLOWTRACE utility, 1-18
- FMT parameter, 7-11
- Form and interpretation of logical expressions, 5-19
- FORM specifier
  - for INQUIRE, 7-22
  - for OPEN, 7-26
- Format
  - control, processing, 8-11
  - establishing, 7-14
  - format identifier, 7-9,11
  - of messages, D-1
  - restrictions, lifting, 8-10
  - specifications
    - and I/O lists, 8-10
    - empty, 8-10
    - list-directed output, 8-5
    - using noncharacter arrays, E-13
- FORMAT statement, 8-5
- Formatted
  - and unformatted records on the same file, 7-15
  - data assignment, E-7
  - data transfer, 7-15
  - I/O
    - prohibited on an unblocked dataset, H-1
    - prohibited under random access, 7-40
  - record, 7-2
- FORMATTED specifier for INQUIRE, 7-22
- Formatting, input/output , 8-1
- FORTRAN character set, A-1
- FORTRAN source code, 1-12,19
- FTREF utility, J-1
- FULLDOREP optimization option, 1-7, 1-15
- FULLIFCON optimization option, 1-7, 1-15
- Function, 2-1,4
  - name, 3-3
  - data type of, 3-2
  - external, 2-2
  - intrinsic, 2-2
  - reference, 2-5, 5-6
    - in an I/O statement, 7-40
  - statement, 2-2
- Function (continued)
  - subprogram, 2-9,12
  - value, 2-1, 2-4
- FUNCTION statement, 2-12, 4-3

- G edit descriptor, 8-23
- G editing, 8-23
- Gather/scatter hardware, 1-3, 1-17
- .GE. function, B-7
- General Arithmetic Functions, B-2
- Generated code, 1-26
- Generic Arithmetic Functions, B-2
- GETPOS function, C-4
  - with random access, 7-39
- Global entities, 4-27
- GOTO statement, 6-1
- Group name parameter, NAMELIST, 7-28
- .GT. function, B-7
- Guidelines for vectorization, 9-19

- H editing descriptor, 8-12
- H editing, 8-12
- Header line, 1-21
- Hexadecimal constants as NAMELIST variables, 7-29
- Hexadecimal editing, 8-25
- Hollerith
  - constant, 3-9, 5-14
  - data, E-2
  - expressions, E-4
  - format specification, E-6
  - relational expressions, E-5
  - type, 5-14
- Hyperbolic
  - cosine function, B-5
  - sine function, B-4
  - tangent function, B-5

- -i parameter in UNICOS CFT command, 1-14
- I edit descriptor, 8-18
- I/O
  - error recovery, 7-40
  - list, 7-12
    - and format specifications, 8-10
  - list-directed, 8-2
  - statements, processing, 7-14
  - unformatted, 8-1
  - using a Pascal routine, F-2
- I=*idn* control statement parameter, 1-5
- ICHAR function, 2-25, B-8
- Identifier, 7-8
- Identifying a unit, 7-14
- IDIM function, B-3
- IDINT function, 2-25, B-7
- IDNINT function, B-2
- IEOF function, C-3, E-13
- IF statement, 1-8
  - nested, inhibit, D-34
- IF-block, 6-5
- IF-level, 6-5
- IFIX function, 2-25, B-7

Imaginary portion of complex value
  function, B-3
IMPLICIT NONE statement, 3-5
IMPLICIT statement, 3-2, 3-4, 4-2
Implied-DO list, 4-16, 7-13
  in a DATA statement, 4-15
  in unformatted I/O, 8-2
$IN, unit identifier number, 7-8
Inactive DO loop, 6-8
INDEF control statement parameter, 1-5
INDEX function, B-7
Index of substring function, B-7
Indirect logical IF statement, E-7
Information supplied by FTREF, J-1
Inner DO loops, 9-2
Input
  file, editing, 7-6
  list item, 7-12
  list-directed, 8-3
  lists, examples, 7-12
  statements, inhibit vectorization, D-34
  to CFT, 1-20
Input/output
  file, 7-4
  formatting, 8-1
  record, 7-2
  statements, 7-1
INQUIRE specifiers and their meanings, 7-20
INQUIRE statement restrictions, 7-23
Inquiry
  by file name, 7-19
  by unit, 7-20
Instruction buffer
  boundary, 1-43
  size, 1-3,17
Instruction scheduler, 1-8
INT function, 2-25, B-7
INT24 compiler directive, 1-36
INT64 compiler directive, 1-36
INT=*il* control statement parameter, 1-5
Integer
  arithmetic, 3-8
  constant, 3-5,6
  control directives, 1-36
  editing, 8-18
  nearest, function, B-2
  output constants, 8-4
  type conversion, 5-11
  values, 3-5
Integer-to-character conversion function,
  B-8
INTEGER type, 3-3, 3-5
INTEGER*2 declaration, 1-12, 1-19
Interactive I/O, with $ edit descirptor,
  8-16
Internal file, 7-5
  identifier, 7-8
  position, 7-7
  definition, 7-2
Internal record, 7-5
Interpretation of blanks on input, 8-18
Intrinsic function, 2-2,4,8, B-1
  referencing, 2-8
INTRINSIC statement, 2-25,26

Invariant, 9-3
  array element, 9-3
  expression, 9-3
INVMOV optimization option, 1-7,15
$IOLIB buffers, 7-41
$IOLIB, regenerating to change I/O buffer
  lengths, 7-41
IOSTAT
  function, E-13
  specifier, 7-20
    for OPEN, 7-25
    for CLOSE, 7-27
IOSTAT= parameter, 7-11
IRTC function, B-11
ISIGN function, B-2
Islist parameter, INQUIRE statement, 7-19,20
Iteration count, 1-27,32, 4-16, 6-7, 6-8
IVDEP compiler directive, 1-34
IVDMO compiler directive, 1-34

JDATE function, B-11
Job area image, save, C-4
Job step, terminate, C-2
Julian time function, B-11

KEEPTEMP optimization option, 1-9,15
KILLTEMP optimization option, 1-9,15
Kinds of array declarators, 4-7

-l parameter in UNICOS CFT command, 1-14
L (logical) editing, 8-26
L edit descriptor, 8-26
L=*ldn* control statement parameter, 1-5
Last-in-first-out storage, 4-18
LDR, 1-20
Leading zeros function, B-10
LEADZ function, B-10
.LE. function, B-7
LEN function, B-7
Length
  for I/O lists, maximum, 7-41
  function, 7-38, C-3
  of an unformatted record, 7-3
  of character constant, 3-11
  of code block, 1-17
  of formatted record, 7-3
  of integers, 1-5,14
  of string function, B-7
LGE function, 2-25, B-7
LGT function, 2-25, B-7
Line width ,default for NAMELIST output,
  7-33
Linkage macros, purpose, F-1
LIST compiler directive, 1-30
List options, 1-5,11
List output, 1-5
List-directed
  I/O, 8-2
  input, 8-3
  output, 8-4
Listable output control directives, 1-30
Literal constant, 4-1

LLE function, 2-25, B-7
LLT function, 2-25, B-7
LOAD, CAL linkage macro, F-1
Loader, 1-20
LOC function, B-11
Local entities, 4-27
Local variables, 1-26, 4-18
Location function, B-11
Logarithmic functions, B-6
Logfile messages, D-27
Logfile, enter message, C-2
Logical
  assignment statement, 5-17
  difference function, B-9
  disjunct, 5-19
  editing, 8-26
  entities, 5-16
  expression, 5-16,19
  factor, 5-19
  IF statement, 6-4
    indirect, E-7
  items, list-directed input, 8-3
  operators, 5-16,17,18
  output constants, 8-4
  product function, B-9
  shift function, B-10
  sum function, B-9
  term, 5-19
LOGICAL type, 3-3,10
Loop control processing, 6-9
Loop preamble, 1-7,22
LOOPMARK utility, 1-5,16
LOOPMARK[=*lmmsgs*] control statement parameter, 1-5
.LT. function, B-7

-m parameter in UNICOS CFT command, 1-14
-M parameter in UNICOS CFT command, 1-17
Machine
  characteristics table, 1-12
  language instructions, 9-1
Main
  program, 2-1,3
  usage field, 1-24
$MAIN, 1-23
Mainframe
  characteristics, 1-3,16
  type, 1-3,16
MASK function, B-9
Masking expression, 5-20
MAX function, 2-25, 9-15
MAX/MIN intrinsic functions, 1-6
MAX0 function, 2-25, B-6
MAX1 function, 2-25, B-6
MAXBLOCK=*mb* control statement parameter, 1-6
Maximum
  array size, 4-12
  value function, B-6
Maximum/minimum functions, B-6
Meanings of Logical Operators, 5-18
Memory
  allocation scheme, 1-2,13
  dump, G-1
Memory (continued)
  requirements, data types, 8-1
  speed, 1-3,17
Messages, 1-28, D-1
  compile-time, D-2
  dependency, D-38
  enter into logfile, C-2
  logfile, D-27
  novector, D-29
  write, C-2
MIN function, 2-25, 9-15
MIN0 function, 2-25, B-6
MIN1 function, 2-25, B-6
Minimum value, B-6
Miscellaneous Functions, B-11
MOD function, B-2
Mode identifier, BUFFER IN and BUFFER OUT, 7-35
Modifying a record under random access, 7-40
Movement of invariant code from DO loops, 1-15
Multiple ports to memory, 9-19
Multiplication, 5-5
Multiply/divide directives, 1-36

Name field, 1-22
NAME specifier, 7-21
Named common block, 4-22
NAMED specifier, 7-21
NAMELIST, 4-3
  input, 7-28
    processing, 7-30
    variables, 7-29
  output, 7-32
  statement, 7-27
Natural logarithm function, B-6
NEQV function, B-9
Nested DO loop, 9-13
NEXTREC specifier, 7-21
NEXTSCALAR compiler directive, 1-35
  example, D-32
NINT function, B-2
NO SIDE EFFECTS compiler directive, 1-42
NOBL compiler directive, 1-40
NOBL optimization option, 1-8, 1-14
NOBTREG optimization option, 1-9,14
NOCODE compiler directive, 1-31
NOCVL compiler directive, 1-45
NOCVL optimization option, 1-9,14
NODEBUG compiler directive, 1-45
NODOREP compiler directive, 1-45
NODOREP optimization option, 1-7,15
NOEMA CPU characteristic, 4-25
NOFLOW compiler directive, 1-37
NOIFCON compiler directive, 1-44
NOIFCON optimization option, 1-6,15
NOINVMOV optimization option, 1-7,15
NOLIST compiler directive, 1-30
Non-ANSI features, outmoded, E-1
Non-ANSI messages, 1-18, D-1
Non-FORTRAN procedures, F-1
Noncharacter arrays for format specification, E-13

- Nonrepeatable edit descriptors, 8-8
- Nonstandard FORTRAN, 1-2
- Nonzero real values, 3-6
- NORECURRENCE compiler directive, 1-33
- NORECURRENCE, example, D-32
- NOTE message, definition, D-1
- NOVECTOR compiler directive, 1-32
- NOVECTOR message, definition, D-1
- Novector messages, D-29
- NOVECTOR, example, D-32
- NOZEROINC optimization option, 1-6
- Nozeroinc optimization option, 1-15
- Null
  - character, in Hollerith constant, E-2
  - symbols, 1-19
  - value in input list, 8-3
- NUMARG function, B-11
- NUMBER specifier, 7-21
- Numeric
  - computation, 5-2
  - editing, 8-17
  - storage unit, 4-17

- -o parameter in UNICOS CFT command, 1-14
- O (octal) editing, 8-25
- O edit descriptor, 8-25
- Octal constants as NAMELIST variables, 7-29
- Octal editing, 8-25
- OFF=*string* control statement parameter, 1-6
- Olist parameter, OPEN statement, 7-23
- ON=*string* control statement parameter, 1-6
- One-line DO loop, 9-13
- OPEN specifiers and their meanings, 7-25
- OPEN statement, 7-23
- OPENED specifier, 7-21
- Operand range error, 1-5
- Operand, 5-1
- Operating system routines, 1-20
  - called by CFT Utilities, C-1
- Operator, 5-1
- OPT=*option* control statement parameter, 1-6
- Optimization, 1-6
  - block, 1-6
  - directives, 1-42
  - options, 1-6,14
- Optimized code, 2-8
- Optimized routines, 9-14
- Optionally signed constant, 4-1
- OR function, B-9
- Out-of-bounds subscripts, 1-12,41
- Out-of-range floating-point value, printing, 8-20
- Outmoded features, E-1
- Output
  - control directives, 1-19
  - for numbers, general rules, 8-17
  - from CFT, 1-20
  - list item, 7-12
  - listing control directives, 1-12
  - lists, examples, 7-12
  - list-directed, 8-4
- Output (continued)
  - NAMELIST group record, 7-32
  - statements, inhibit vectorization, D-34
  - to a printer, 7-16
- $OUT, unit identifier number, 7-8

- P editing, 8-16
- Page header lines, 1-21
- PARAMETER statement, 3-4, 4-1
- Parameter, passing between C and CFT, F-2
- Parcel address, 1-27
- Parenthesized expression, 5-1
- PARTIALIFCON optimization option, 1-6,15
- Partially associated entities, 4-19
- Pascal procedures, calling, F-1
- PAUSE statement, 6-12
- Plus signs, control on output, 8-18
- Pointee, 1-25, 3-14
  - array, 4-4,5
    - declarator, 4-8
- Pointer, 3-13
- POINTER statement, 3-14, 4-5
- POPARR function, B-10
- POPCNT function, B-10
- Population count function, B-10
- Population parity count function, B-10
- Position edit descriptors, 8-13
- Positional editing (T, TL, TR, and X), 8-13
- Positioning
  - by format control, 8-11
  - while connected for random access, 7-39
- Pound blocks, 1-25
- Precedence of arithmetic operators, 5-5
- Primary, 5-6
- Print control characters, 7-16
- Printer output, 7-16
- Printing, 7-16
- Procedure, 2-1
  - reference, 4-4
  - subprogram, 2-9, 4-4
- Product, double-precision, function, B-3
- Program
  - block, 1-25
  - control statements, 6-1
  - structure, 2-1
  - unit, 1-28, 2-3
  - Unit Page Table, 1-28
  - variables, dump contents, C-4
- PROGRAM statement, 2-3
- Pseudo CAL file, 1-18
- Pseudo vector, 9-4
- PUNCH statement, E-11
- $PUNCH, unit identifier number, 7-8
- Pushdown storage, 4-18

- Qualifications for vectorization, 9-1
- Quotation mark edit descriptor, 8-12
- Quotation mark editing, 8-12

- R (right-justified) editing, 8-28
- R edit descriptor, 8-28

- Random
  - access dataset, creating, 7-38
  - connection access method, 7-38
  - number function, B-3
  - seed function, B-3
- RANF function, B-3
- RANGET function, B-3
- RANSET function, B-3
- $RBUFLN, 7-41
- READ statement,7-9
  - format, 7-10
- Read-only constants, 1-26
- Real
  - constant, 3-6
  - exponent, 3-6
  - items, list-directed input, 8-3
  - output constants, 8-4
  - type conversion, 5-11
  - function, 2-25, B-8
- REAL type, 3-3,6
- Real-time clock function, B-11
- REC parameter, 7-11
- RECL specifier, 7-21
  - for OPEN, 7-26
- Record, 7-2
  - identifier, 7-11
  - numbers, in direct access of external files, 7-6
  - of an internal file, 7-5
  - end-of-data, 7-4
  - end-of-file, 7-2
  - formatted, 7-2
  - input/output, 7-2
  - internal, 7-5
  - NAMELIST input, 7-29
  - output NAMELIST group, 7-32
  - unformatted, 7-2
- Recurrence relation, 1-33
- Recurrence, referencing the result inhibits vectorization, D-36
- Recursion
  - in subprogram, 2-10
  - reduction variable, 9-4
  - variable as a subscript, D-30
- Reductions, vectorizing, D-38
- Referencing common blocks, 4-23
- Referencing intrinsic function, 2-8
- Relational expression, 5-14,16
- Relational operator, 5-16
- Relocatable binary
  - output, 1-20
  - binary program, 1-1
- Remainder function, B-2
- REMARK subroutine, C-2
- REMARK2 subroutine, C-2
- REMARKF subroutine, C-2
- Repeat specification, 8-5
- Repeatable edit descriptors, 8-7
- Repeated execution of statements, 6-11
- Replacement of one-line DO loops, 1-15
- Reprieve
  - initiation, I-1
  - processing, I-1
  - termination, I-2
- RESUMEDOREP compiler directive, 1-44
- RESUMEIFCON compiler directive, 1-44, 9-16
- RETURN statement, 2-17, 6-8
- REWIND statement, 7-17, 7-19
- $RFDCOM, resizing, 7-41
- Right-justified output, 8-18
- RNLCOMM routine, 7-32
- RNLDELM routine, 7-31
- RNLECHO routine, 7-31
- RNLFLAG routine, 7-31
- RNLREP routine, 7-32
- RNLSEP routine, 7-31
- RNLSKIP routine, 7-31
- RNLTYPE routine, 7-31
- ROLL compiler directive, 1-46
- ROLL/UNROLL directives, 1-46
- Round result on multiply operations, 1-19
- Round-off errors, 5-12
- Rounding results of multiply operations, 1-12
- RTC function, B-11
- Rules for numeric editing, 8-17

- S edit descriptor, 8-18
- S editing, 8-18
- SAFEDOREP optimization option, 1-7,15
- SAFEIF compiler directive, 1-40
- SAFEIF optimization option, 1-8,15
- SAVE statement, 1-10,18, 2-18
- SAVEALL control statement parameter, 1-10
- Scalar
  - loops, 1-32
  - merge function, B-10
  - mode, 1-32,35
  - temporaries, 1-9, 9-14
  - temporary variables, 1-15
- Scale factor, edit descriptor, 8-16
- Scheduler directives, 1-40
- $SCILIB, 1-7,15,27,45,13
- Scope of symbolic names, 4-27
- SDOT library routine, 9-14
- SECOND function, C-4
- Seed, random, function, B-3
- SEGLDR, 1-20
  - directives for changing I/O buffer lengths, 7-41
  - lifting format restrictions, 8-10
- SENSEFI subroutine, C-3
- Separators, list-directed output, 8-4
- Sequence number, 1-18
- Sequential access, 7-7
  - of external files, 7-6
- Sequential file statements, 7-17
- SEQUENTIAL specifier for INQUIRE, 7-22
- SET SEGLDR directive, 7-41
- SETFI subroutine, C-3
- SETFIS subroutine, C-3
- SETPLIMQ directive, 1-39
- SETPOS function, C-4
- SETPOS with random access, 7-39
- SETRPV library routine, I-1
- SETRPV subroutine, C-4
- SHIFT function, B-10

SHIFT does not vectorize, D-36
Shift, circular, function, B-10
SHIFTL function, B-10
SHIFTR function, B-10
Short vector loop, 1-27
SHORTLOOP compiler directive, 1-35
Side effects, 1-42
SIGN function, B-2
Sign, transfer, function, B-2
Signed constant, 4-1
Simple variable, 4-2
Simultaneous I/O, 7-35
SIN function, B-4
Sine function, B-4
SINH function, B-4
Size of an assumed-size array, 4-11
Skip record, 8-10
Skip positions edit descriptor, 8-14
Slash as separator during list-directed input, 8-3
Slash edit descriptor, 8-14
Slash editing, 8-14
Slashes as value separators, list-directed output, 8-5
SLOWMD compiler directive, 1-36
SLOWMD optimization option, 1-7,15
SNGL function, 2-25, B-8
Source
    input, 1-5
    listing, 1-14
    statement listing, 1-20, 1-21
SP edit descriptor, 8-18
SP editing, 8-18
Specification subprogram, 2-9
SQRT function, B-5
Square root function, B-5
SS editing, 8-18
SSUM library routine, 9-14
SSWITCH subroutine, C-2
Stack, 1-2,11,13, 4-18
    allocation, 4-26
    frame, 1-5
    mode, 1-26, 2-3,10
    Space Table, 1-25,26
    storage, 1-24, 4-18
    variable, 1-18, 4-3
Statement
    function, 2-2,4,5
        definition statement, 2-6
    labels, 1-11,19,22
    numbers, 1-22
    sequence, 6-1
Statements
    for using subprograms, 2-16
    I/O, 7-1
Static
    memory, 1-13
    mode, 1-25
    Space Table, 1-25,26
        and Stack Space Table, 1-25
    storage, 1-2,10,11, 4-14,18
    variable, 4-3
Status identifier, 7-11
STATUS specifier for OPEN, 7-26
STATUS specifier for CLOSE, 7-27
Stderr unit identifier number, 7-8
Stdin unit identifier number, 7-8
Stdout unit identifier number, 7-8
STOP statement, 6-8,12
    inhibits vectorization, D-31
Storage, 4-1
    and association, 4-17
    sequence, 4-17
        for a Three-dimensional Array, 4-10
    unit, 4-17
STORE, CAL linkage macro, F-1
String, length of, function, B-7
Subcategories of FORTRAN Terms, 2-2
Subprogram, 2-1,3,9
    entry, 2-3
    recursion in, 2-10
Subroutine, 2-1,13
    calls, inhibiting vectorization, D-34
    subprogram, 2-9,13
SUBROUTINE statement, 2-14
Subscript
    calculations, 4-9
    evaluation, 4-11
    expression, 4-9
    of array element name, 4-8
    value, 4-8
Subscripted variable, 4-2
Substring, index, function, B-7
Subtraction, 5-5
Suspending execution, 6-12
Symbol Table, 1-11,12,13,19
Symbolic debug package, G-1
Symbolic names, 1-23, 4-26, 5-6
SYMDEBUG library routine, G-1
SYMDEBUG subroutine, C-4
Synchronous I/O, with unblocked datasets, H-1
SYSTEM function, C-2

-t parameter in UNICOS CFT command, 1-16
T edit descriptor, 8-13
T editing, 8-13
T registers, 1-8,10,14,26
Table of block names and lengths in octal, 1-25
Table
    of external names, 1-26
    of loops encountered, 1-26
        includes NOVECTOR messages, D-30
    of names encountered, 1-22
    of parameters encountered, 1-25
    of statement numbers, 1-22
TAN function, B-4
Tangent function, B-4
TANH function, B-5
Task common block, 1-24
TASK COMMON statement, 4-25
TASK keyword, 4-26
TASK START statement, 2-19
Temporary variables, 1-26
Term, 5-7
Terminal statement of DO loop, 6-7, 6-11

Terminate job step, C-2
Terminating execution, 6-12
Time and Date Functions, B-11
Time function, B-11
Time limit, recovery from, I-1
TIMEF function, C-4
TL edit descriptor, 8-13
TL editing, 8-13
Totally associated entities, 4-19
TR edit descriptor, 8-13
TR editing, 8-13
Traceback, write, C-2
Transfer of control, 6-1
Transferring data, 7-14
Transportability of programs, 9-1
TRBK function, C-2
Trigonometric functions, B-4
Trip count, 1-10
True logical value, 3-10
TRUNC=*tr* control statement parameter, 1-10
Truncation, 1-10,16
  count, 1-21
  function, B-2
Two-branch arithmetic IF statement, E-6
Type
  Boolean, 5-11
  conversion functions, B-7
  conversion, 5-11
  field, 1-22
  statement, 3-3
    data length, E-11

-u parameter in UNICOS CFT command, 1-16
Unblocked dataset, H-1
  with buffered I/O, 7-36
Unconditional GOTO, 6-11
Unformatted
  and formatted records on the same file, 7-15
  data transfer, 7-15
  I/O, 8-1
    with formatted records, 7-3
    with unblocked datasets, H-1
  record, 7-2
UNFORMATTED specifier for INQUIRE, 7-22
UNICOS, 1-1,13
  CFT command, 1-13
Uninitialized integer variable, 1-5
Unit, 7-7
  identifier, 7-8
    parameter, 7-10
  100, preconnected to $IN or stdin, 7-14
  101, preconnected to $OUT or stdout, 7-14
UNIT
  function, 7-37, C-3
  specifier for CLOSE, 7-27
  specifier for OPEN, 7-25
UNROLL compiler directive, 1-46
UNROLL=*r* control statement parameter, 1-10
Unrolled loop, example, D-35
Unrolling of DO loops, 1-46
UNSAFEIF compiler directive, 1-40
UNSAFEIF optimization option, 1-8,15
Unsigned arithmetic constant, 4-1, 5-2,6
Use of array names, 4-13
Use of Data Types with Arithmetic Operations, 5-9
User control subroutines, 7-30,33
User-supplied code, 2-2
User-supplied function, 2-8
Using compiler directive lines, 1-29
Utility procedures, C-1

-v parameter in UNICOS CFT command, 1-16
Value separators, forms, 8-2
Values Represented in Different Data Types, 3-16
Variable, 3-1, 4-2
  name, 3-3
  reference, 5-6
  usage, common block, J-1
Variables that can contain Hollerith data, E-5
Vector array reference, 9-3,4
Vector block, 1-22
VECTOR compiler directive, 1-32
Vector
  dependency, 9-5
  dot product, 9-13
  functional unit, 9-1
  logical functional units, 1-3,17
  loops, 1-32
  merge, 9-16
  population count functional unit, 1-3,17
  processing, 9-1
  recursion, 1-3,17
  sum, 9-13
  temporaries inhibiting vectorization, D-38
Vectorizable DO loops, 9-1
Vectorization, 1-6,32
  control directives, 1-31
  functions, B-12
  guidelines, 9-19
  inhibition messages, 1-2,16
  with arrays, 9-12
VFUNCTION compiler directive, 1-34

WARNING message, definition, D-1
$WBUFLN, 7-41
$WFDCOM, resizing, 7-41
WNLDELM routine, 7-33
WNLFLAG routine, 7-33
WNLLINE routine, 7-33
WNLLONG routine, 7-33
WNLREP routine, 7-33
WNLSEP routine, 7-33
Write
  a traceback, C-2
  an ASCII message, C-2
  EOD, C-3
WRITE statement, description, 7-9
WRITEDS to create a dataset, 7-38

X edit descriptor, 8-13
X editing, 8-14
XOR function, B-9

Z (hexadecimal) editing, 8-25
Z edit descriptor indicates, 8-25
ZEROINC optimization option, 1-6
Zeroinc optimization option, 1-15
Zeros, leading, function, B-10

24-bit integer, 1-5,36
46-bit integer multiply and divide, 1-15,36
64-bit integer arithmetic, 1-36
64-bit integer multiply and divide, 1-7,15
64-bit integer, 1-5,36

READER'S COMMENT FORM

FORTRAN (CFT) Reference Manual SR-0009 L

Your reactions to this manual will help us provide you with better documentation. Please take a moment tc check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): ____________________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ________________________
Title ________________________
Company ____________________
Telephone ___________________
Today's Date ________________

Address _____________________
City ________________________
State/ Country ________________
Zip Code ____________________

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120

READER'S COMMENT FORM

FORTRAN (CFT) Reference Manual SR-0009 L

Your reactions to this manual will help us provide you with better documentation. Please take a moment to check the spaces below, and use the blank space for additional comments.

1) Your experience with computers: ____ 0-1 year ____1-5 years ____5+ years
2) Your experience with Cray computer systems: ____0-1 year ____ 1-5 years ____5+ years
3) Your occupation: ____ computer programmer ____ non-computer professional
____ other (please specify): ______________________________
4) How you used this manual: ____ in a class ____as a tutorial or introduction ____ as a reference guide
____ for troubleshooting

Using a scale from 1 (poor) to 10 (excellent), please rate this manual on the following criteria:

5) Accuracy _____
6) Completeness ____
7) Organization ____
8) Physical qualities (binding, printing) ____
9) Readability ____
10) Amount and quality of examples ____

Please use the space below, and an additional sheet if necessary, for your other comments about this manual. If you have discovered any inaccuracies or omissions, please give us the page number on which the problem occurred. We promise a quick reply to your comments and questions.

Name ________________________
Title ________________________
Company ____________________
Telephone ___________________
Today's Date _________________

Address _____________________
City _________________________
State/ Country ________________
Zip Code _____________________

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention: PUBLICATIONS
1345 Northland Drive
Mendota Heights, MN 55120